The Use of One-Time Password and RADIUS Authentication in a GSS-API Architecture

XI YANG

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2006

CoS/CCS 2006-09

The Use of One-Time Password and RADIUS Authentication in a GSS-API Architecture

Thesis Report

Xi Yang

2006-04-10

Master of Science thesis performed at
PortWise AB
Stockholm, Sweden

Academic Advisor and Examiner: Prof. Gerald Q. Maguire Jr.
Industrial Advisor: Jon Martinsson

School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

The Generic Security Service Application Program Interface (GSS-API) is an architecture that facilitates applications using distributed security services in a mechanism-independent fashion. GSS-API is supported by various underlying mechanisms and technologies such as Kerberos version 5 and public-key technologies. However, no one-time password based GSS-API mechanism existed.

This thesis focuses on an investigation using one-time passwords together with RADIUS authentication as a protection facility for a GSS-API mechanism. This thesis presents a security architecture using one-time passwords to establish a GSS-API security context between two communicating peers.

The proposed one-time password based GSS-API mechanism could be used to enhance the security of user authentication. Moreover, the mechanism can greatly facilitate static-password based system's transition to stronger authentication.

Keywords: GSS-API, one-time password, RADIUS authentication

Sammanfattning

IETF GSS-API är ett applikationsgränssnitt (API) som tillhandahåller distribuerade säkerhetstjänster för autentisering och datakonfidentialitet oberoende av den underliggande säkerhetarkitekturen. Applikationer som skrivs mot detta API kan på detta sätt flyttas eller porteras utan att västentligen skrivas om. GSS-API stöds av ett flertal undrliggande säkerhetsarkitekturer som tex Kerberos 5, Windows NTLM och PKI. API har också sk bindings för "C" och Java. I dagsläget finns det dock ingen lösning som baseras på engångslösenord.

Denna magisteruppsats har som mål att undersöka möjligheten att använda engångslösenord tillsammans med RADIUS för att implementera en ny GSS-API mechanism. Denna uppsats presenterar ett förslag för hur RADIUS och engångslösenord kan användas för att säkra kommunikationen mellan två GSS-API entiteter.

Den föreslagna mekanismen kan också användas för att förbättra säkerheten för användarautentisering och möjliggöra en övergång från statiska lösenord till stark autentisering.

Acknowledgements

I would like to express my sincere gratitude to my academic advisor, Prof. Gerald Q. “Chip” Maguire Jr., for his inestimable suggestions, comments, and guidance.

I am also highly thankful to my industrial advisor at PortWise, Jon Martinsson, for his advice and valuable suggestions.

I would like to thank my family and friends for their support throughout the duration of this project.

Table of Contents

1. Introduction

When users were simply accessing information on a closed local area network (LAN), simple passwords were usually sufficient to protect information. However, the increasingly distributed computing environment are often operated by multiple parties, for whom there is often lack of strong authentication, that resulted in the rise in identity theft, the need for stronger levels of authentication has become critical. This is especially true, as password-cracking programs are available that can break typical passwords in a short period of time.

Many security mechanisms and technologies have been developed for enhancing the security of user authentication, such as Kerberos [1] and public-key technologies [2]. To facilitate applications working with a range of security systems, and to enable the use of security mechanisms independently of application, the Generic Security Service Application Programming Interface (GSS-API) was introduced (see section 2.1). With the help of GSS-API, a single application could be used by users with different security requirements, rather than requiring multiple version of the application.

The current dominant GSS-API mechanism implementation in use is Kerberos. However, there is currently no one-time password based GSS-API mechanism available. Comparing with reusable passwords, the use of one-time passwords provides greater security for user authentication by eliminating the possibility of "replay attacks".

This Master of Science thesis work was performed at PortWise AB, a secure remote application access platform provider. The goal was to design, implement, and evaluate a security mechanism using one-time passwords to establish a GSS-API security context between two communicating peers. Two-factor authentication is applied in this solution (see section 2.3.2).

Two cryptographic algorithms, HMAC-SHA-1 and AES, are employed for providing message integrity and confidentiality. DES is also supported by our proposed security mechanism to provide message confidentiality. Users are allowed to choose what algorithm they would like to use by specifying the Quality of Protection (QoP) parameter in certain GSS-API routines.

For compatibility with PortWise's existing systems, the proposed security architecture is limited to using RADIUS protocol for user authentication. With the emergence of the Diameter (see section 2.2.5), an advanced Authentication, Authorization, and Accounting (AAA) protocol, the migration to the Diameter protocol would be essential future work as Diameter overcomes the several RADIUS deficiencies.

The focus of this thesis is to investigate the use of one-time password and RADIUS authentication as a protection facility for a GSS-API mechanism. One of the results of this thesis project, should be a one-time password based GSS-API mechanism, to be provided as a shared library in a Unix-like environment.

For performance and conformance testing, SAP's GSS-API test program (see section 5.1) is used to evaluate the implementation. A sample client and server pair using this GSS-API shared library was created to validate the security mechanism and demonstrate how the mechanism works in real life. Those two sample programs were also used for further performance tests.

The results of evaluation indicate that the GSS-API implementation conforms to the GSS-API version 2 specification; the underlying security mechanism works properly. The one-time password based GSS-API mechanism can facilitate the transition to stronger authentication from static-password based systems.

2. Background

2.1 GSS-API

2.1.1 GSS-API Overview

The Generic Security Service Application Programming Interface (GSS-API) is an Internet Engineering Task Force (IETF) standardized API that provides an abstract security service interface for distributed applications. The GSS-API itself does **not** provide any specific security service. Using the GSS-API callers (i.e. applications) acquire security services from the underlying security mechanism(s).

The latest version of GSS-API is version 2, update 1. It is described in RFC 2743 [3]. RFC 2744 specifies C language bindings for the latest version of GSS-API [4], while a Java binding specification of GSS-API is also available [5].

2.1.2 Motivation for GSS-API

The motivation for the emergence of GSS-API is that previously each security mechanism had its own API, thus it was extremely difficult for applications to utilize different security mechanisms because of the differences between each security mechanism's API. However, with a generic API, applications could work with various underlying security systems. GSS-API acts as a framework that facilitates applications using security services in a *mechanism-independent* fashion.

The GSS-API offers four types of portability for applications:

1. Platform independence
The GSS-API is designed to be oblivious to the Operating System that an application is running on.

2. Protocol independence
The GSS-API is completely isolated from any underlying communication protocol.

3. Mechanism independence
As mentioned above, the GSS-API is a generic interface to a security mechanism that has implemented the GSS-API. An application does not need to know which mechanism it is using, as long as a default security mechanism has been specified.

4. QoP independence
Quality of Protection (QoP) is used for specifying the type of algorithm used for providing data integrity and confidentiality. By using the default algorithm provided by the GSS-API, a programmer is allowed to ignore QoP.

The relationship between security mechanisms, applications, and GSS-API is shown in Figure 2-1.

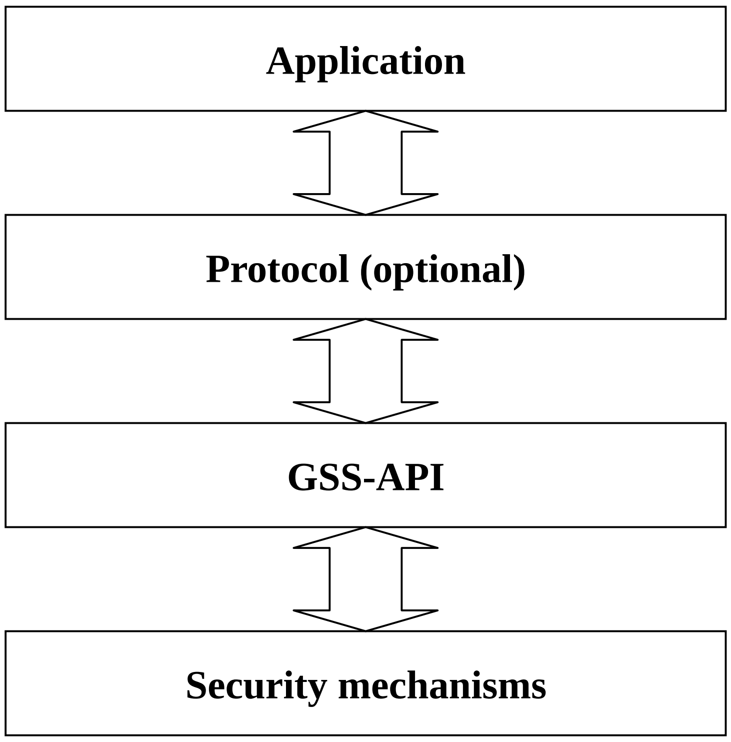

Figure 2-1: Layers of the GSS-API architecture

2.1.3 How GSS-API is used by Applications

GSS-API implementations are usually provided in the form of libraries by security service vendors. These libraries are installed together with some underlying security software. Application writers can write their applications using the GSS-API-compatible interface presented by these libraries. There is no need to rewrite applications when an underlying security implementation needs to be replaced.

Figure 2 illustrates the communication between two GSS-API-aware applications.

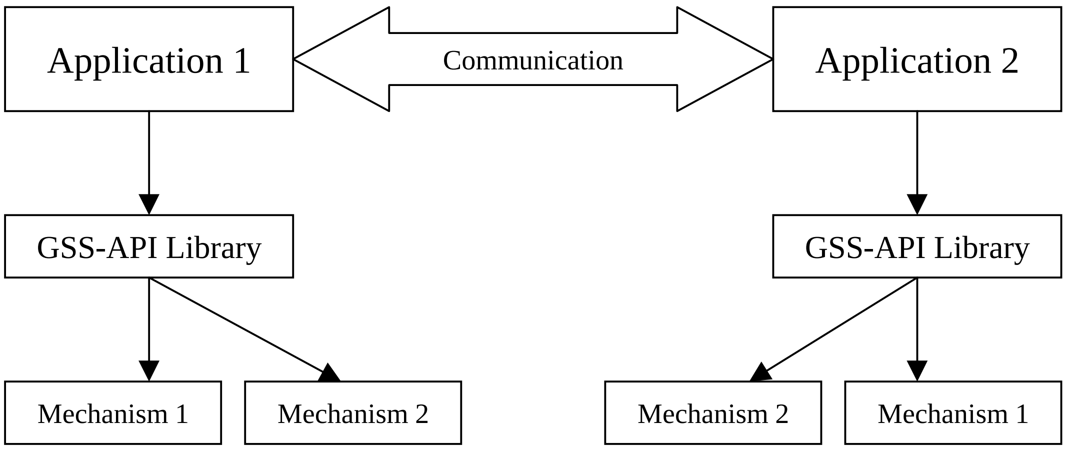

Figure 2-2: GSS-API based communication

The GSS-API libraries on the two hosts are responsible for generating and processing the tokens, i.e. message units used for exchanging information (see section 3.6 for detailed information about tokens used in our proposed security mechanism). The application is responsible for transporting these tokens between the two peers. Thus the underlying security mechanisms provide their security services to the application.

2.1.4 Security Services

There are three kinds of security services provided by GSS-API mechanisms:

1. Authentication
Authentication is the process of verifying an identity. It determines whether someone/something is who/what they declare themselves to be. Authentication is a basic security service offered by GSS-API.

2. Integrity
Integrity refers to the validity of data. GSS-API provides a Message Integrity Code (MIC) that can be attached to data. This MIC is a cryptographic tag that is used to verify that the received data is the same as the data that the sender has transmitted. Integrity is an additional security service, optionally supported by the underlying security mechanisms.

3. Confidentiality
Confidentiality is the protection of the contents of communication from disclosure to unintended parties. Achieving confidentiality is based upon encryption of the data. As with integrity, confidentiality is also an additional security service. The GSS-API

utilizes the underlying security mechanism to perform this data encryption.

2.1.5 A Walk-through of GSS-API Routines

GSS-API routines can be sorted into five categories according to their functionality. These are (1) credential management routines, (2) context-level routines, (3) per-message routines, (4) name manipulation routines, and (5) support routines. Below, we examine these corresponding GSS-API routines.

1. Credential management routines
A credential is the information that proves that a communicating party has a specific identity. It is used to provide the prerequisites that permit GSS-API peers to establish security contexts with each other. Credential management routines deal with operations upon credentials. These routines are:

gss_acquire_cred( )	Acquire a GSS-API credential handle for use.
gss_release_cred( )	Discard a credential handle.
gss_add_cred( )	Add a credential-element to a credential.
gss_inquire_cred( )	Get information about a credential.
gss_inquire_cred_by_mech( )	Get per-mechanism information about a credential.

Because credentials are various in size and structure in different security mechanisms, credential management routines simply deal with the credential via handles, which act as pointers to the actual credentials. However, since the application never deals with the credential itself, the application can be completely independent of how the credential is represented, transferred, etc.

2. Context-level routines
The security context holds state information about each end of a peer communication. GSS-API provides secure communication between two peers with a security context established by an exchange of tokens. Context-level routines are responsible for context establishment, context transfer, and other operations related with security context. These routines are:

gss_init_sec_context( )	Establish a security context between the application and a remote peer.
gss_accept_sec_context( )	Allow a remotely initiated security context between the application and a remote peer to be

	established.
gss_delete_sec_context( )	Delete a security context when no longer needed.
gss_process_context_token( )	Process a received token in the security context.
gss_context_time( )	Determine the number of seconds for which the specified context will remain valid.
gss_inquire_context( )	Get information about a security context.
gss_wrap_size_limit( )	Determine the maximum message size for gss_wrap( ) call under specific security context.
gss_export_sec_context( )	Generate a token for transferring a security context to another process.
gss_import_sec_context( )	Import a security context from a token.

3. Per-message routines

Per-message routines are devoted to the protection of messages within established security contexts. These routines are:

gss_get_mic( )	Calculate a cryptographic Message Integrity Code (MIC) for a message.
gss_verify_mic( )	Verify integrity of a received message by checking the MIC along with the message.
gss_wrap( )	Encapsulate a message and its MIC, optionally encrypt the message content. The conf_req_flag parameter determines if the message is encrypted. The qop_req parameter specifies how the message is encrypted.
gss_unwrap( )	Decapsulate a received token, decrypt message if necessary, verify the message with attached MIC.

4. Name manipulation routines

A name is used to identify an entity. In GSS-API, there are two kinds of representation defined for names:

- **An internal name**. This internal form of names is the GSS-API "native" format for names. The internal names are stored as the implementation-specific `gss_name_t` type, which is opaque to GSS-API callers.

- **A mechanism-specific name**. This is a format that contains only a single version of the name, specific to a certain mechanism. A mechanism-specific name is also called a Mechanism Name (MN). “Mechanism Name” does not represent the name of a mechanism, rather it refers to the name produced by a given mechanism.

Name manipulation routines are used to deal with various name types and naming schemes. These routines are:

gss_import_name( )	Convert a printable name to internal-form.
gss_display_name( )	Translate internal-form name to printable form.
gss_compare_name( )	Compare two internal-form names for equality.
gss_release_name( )	Free storage of a internal-form name.
gss_export_name( )	Convert mechanism-specific name to export form.
gss_duplicate_name( )	Create a copy of name object.
gss_canonicalize_name( )	Translate an internal name to mechanism-specific form.
gss_inquire_names_for_mech( )	Get a list of name-types that supported by the specified mechanism.
gss_inquire_mechs_for_name( )	Get a list mechanisms that support the specified name-type.

5. Support routines

This is a set of routines consists of various supporting functions for GSS-API. These routines are:

gss_display_status( )	Convert GSS-API status codes to printable form.
gss_release_buffer( )	Free storage of a GSS-API allocated buffer.
gss_indicate_mechs( )	Indicate underlying authentication mechanisms that are supported on the local system.

gss_create_empty_oid_set( )	Create an empty object identifier set.
gss_add_oid_set_member( )	Add an object identifier to a set.
gss_test_oid_set_member( )	Determine whether an object identifier is a member of a set.
gss_release_oid_set( )	Free storage former used by a set of object identifiers.

An object identifier is a data type containing ISO-defined tree-structured values. It is used by GSS-API caller to select an underlying security mechanism and to specify name types.

2.1.5 Workflow of GSS-API-aware Applications

Typically, GSS-API-aware applications follow these basic steps.

Step 1: Acquire credentials
As credentials are used for proving an application's identity, the application first acquires credentials, for example, from a successful login.

Step 2: Establish security context
Two communicating applications establish a security context by exchanging tokens. A security context establishes and maintains shared state information in each peer, once established, this state is used to provide a secure channel for communication.

Step 3: Protected data communication
Per-message services are invoked to protect the data to be exchanged by these peers. An application calls the appropriate GSS-API routine (`gss_get_mic()` or `gss_wrap()`) to apply data protection, and to transfer the resulting token to its peer. Upon receiving such a token, the peer application passes it to corresponding GSS-API routine (`gss_verify_mic()` or `gss_unwrap()`) which then removes the protection and validates the data or reject the data as invalid.

Step 4: Delete security context
At the end of a communication session, each application calls a GSS-API routine to delete the security context.

Figure 2-3 shows the basic workflow of two communicating GSS-API peers and the specific GSS-API routines which are called.

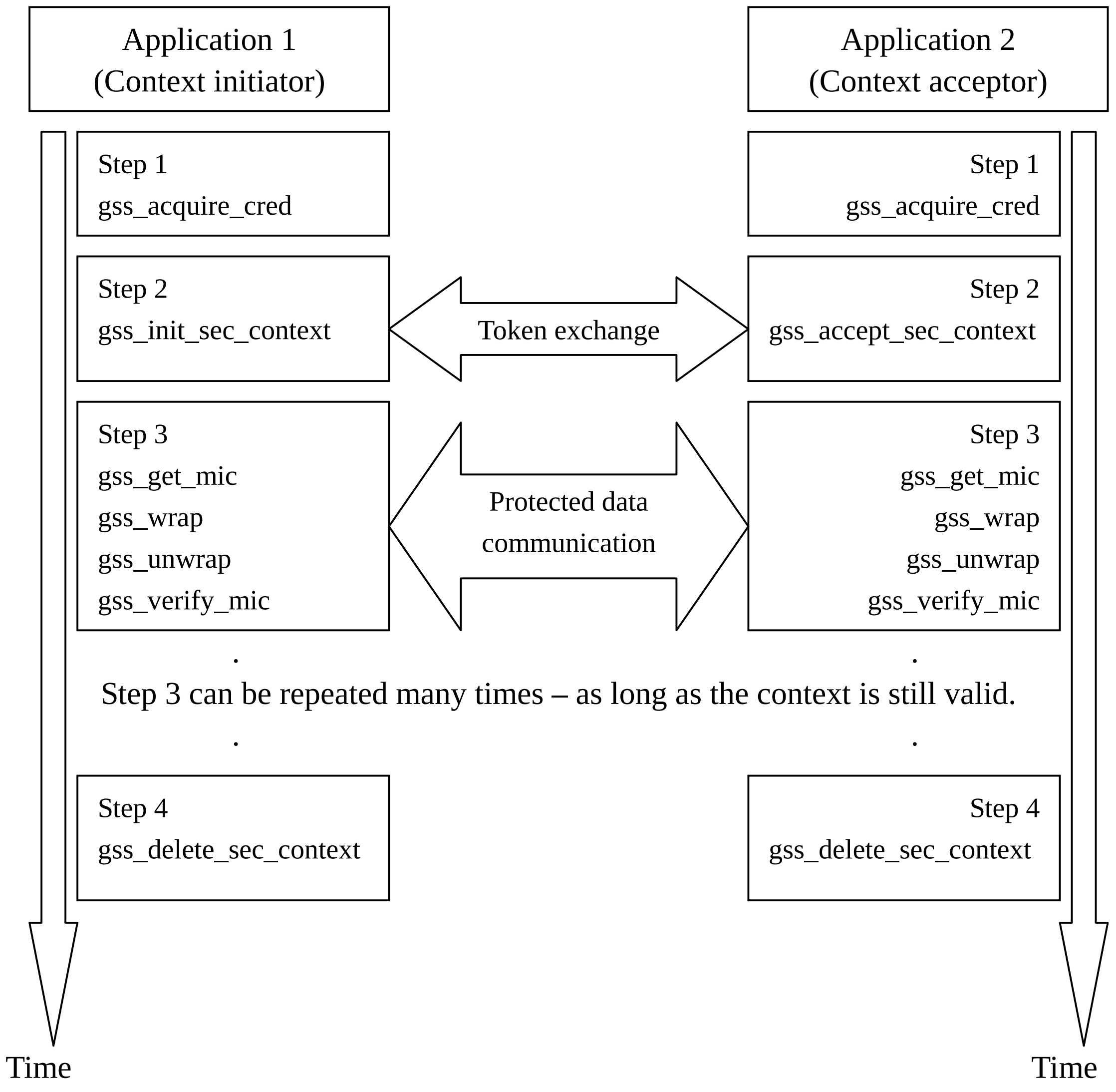

Figure 2-3: Basic GSS-API workflow

2.1.6 Mechanisms Available via GSS-API

The current dominant GSS-API mechanism implemented is Kerberos. The latest version of the Kerberos protocol is version 5, as defined in RFC 4120 [1]. RFC 4121 [6] describes the GSS-API mechanism of Kerberos version 5. MIT Kerberos [7] and Heimdal Kerberos [8] distributions are open-source software that implement the Kerberos GSS-API mechanism.

Sun Microsystems' Sun Enterprise Authentication Mechanism (SEAM) is another Kerberos GSS-API variant [9]. Microsoft Windows 2000 and later also implements a Microsoft specific variant of GSS-API on top of its authentication infrastructure as part of Windows [10].

Based on public-key technologies, the Simple Public-Key Mechanism (SPKM) [2] can be employed by peers implementing the GSS-API. A commercial X.509 Public-Key Infrastructure (PKI) based GSS-API implementation also exists in SECUDE [11].

2.2 RADIUS

2.2.1 RADIUS Overview

The Remote Authentication Dial-In User Service (RADIUS) is an Authentication, Authorization, and Accounting (AAA) protocol. RADIUS can be used on any network that needs a centralized authentication, authorization and/or accounting service.

RADIUS is a transaction-based protocol build on top of the User Datagram Protocol (UDP). UDP was chosen instead of Transmission Control Protocol (TCP) as a transport protocol for technical reasons [12]. The officially assigned port number for RADIUS authentication is UDP port 1812.

RADIUS was originally developed by Livingston Enterprises for their PortMaster series of Network Access Servers (NAS). In 1997, the protocol was published as RFC 2058 [13]. The latest version of the RADIUS protocol is defined in RFC 2865 [12].

The RADIUS protocol is used to authenticate the users to an access network, authorize access to the network, and provide accounting service. As its name implies, RADIUS was originally developed for dial-up remote access. Today RADIUS is widely used for: Digital Subscriber Line (DSL) access, Virtual Private Network (VPN) servers, wireless access points, authenticating Ethernet switches, and other types of network access systems.

2.2.2 Summary of a RADIUS Packet

RADIUS packets are sent as a single User Datagram Protocol (UDP) message. Exactly one RADIUS packet is encapsulated in the UDP Data field [14]. When a reply is generated, the source and destination ports are reversed.

The following Figure 2-4 describes the RADIUS data format, as defined in RFC 2865.

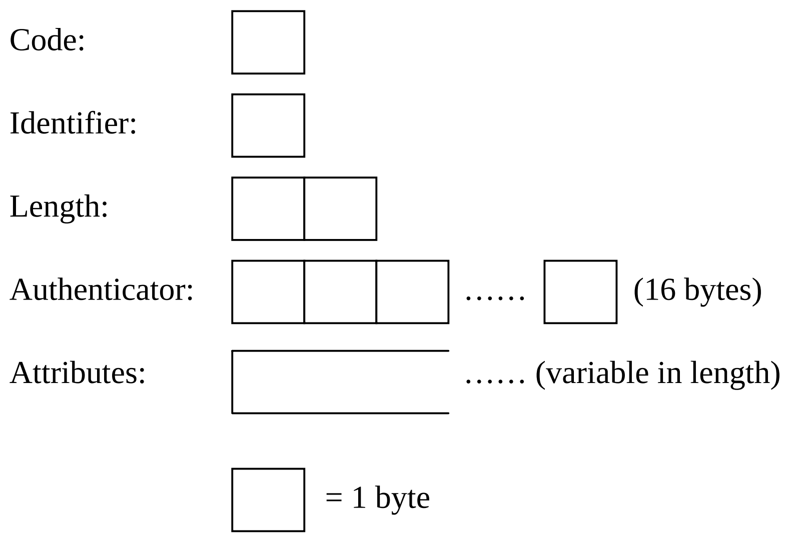

Figure 2-4: RADIUS packet format

Code The RADIUS packet code field indicates the type of a RADIUS packet; it is 1 byte long. Table 2-1 shows the assigned code values and describes each packet type.

Code value	Packet type
1	Access-Request
2	Access-Accept
3	Access-Reject
4	Accounting-Request
5	Accounting-Response
11	Access-Challenge
12	Status-Server (experimental)
13	Status-Client (experimental)
255	(Reserved)

Table 2-1: RADIUS codes [12]

Identifier	The identifier field is 1 octet. Its value is used for matching RADIUS requests to responses. The Identifier value helps a RADIUS server detect duplicate requests. Such duplicates could occur because the underlying UDP packet was replicated or retransmitted.
Length	The length value indicates the length of the whole RADIUS packet in bytes. This field is 2 octets, the minimum length value is 20 and maximum length is 4096.
Authenticator	The Authenticator field is 16 octets. This authenticator is used to authenticate the response from a RADIUS server.
Attributes	The Attributes field contains the list of attributes that are used for the corresponding service (i.e. packet type). This field is variable in length, and is of the length specified via the length field above minus the 20 bytes of header.

2.2.3 How a RADIUS Authentication System Works

A typical RADIUS authentication system involves the following four parties:

User	A user is an entity who wants to have access to network resources.
Client	A client is an entity who asks on behalf of a user to access or use network resources. A client could be a network access server (NAS), a network switch, a VPN server, or a wireless access point.
Server	A server usually is a software program that is listening on specific ports to accept requests. The server accepts access requests coming from clients, checks these requests, and sends an appropriate response.
Database	The database contains data used for authenticating an access request. A database could be a simple flat file or a relational database, such as Microsoft's SQL Server, Oracle, MySQL, etc.

Figure 2-5 illustrates a typical RADIUS authentication system.

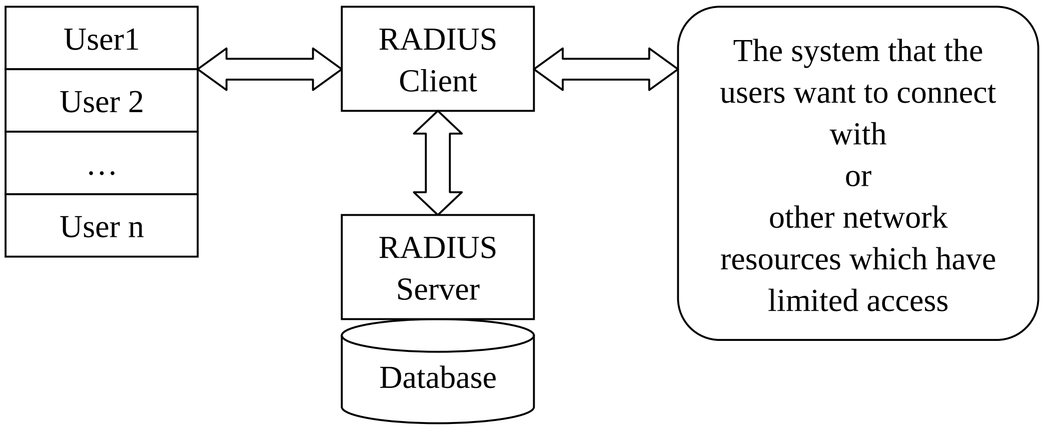
User1
User 2
…
User n
RADIUS Client
RADIUS Server
Database
The system that the users want to connect with or other network resources which have limited access

Figure 2-5: A RADIUS authentication system

A RADIUS client acquires authentication data from user, and supplies its authentication data to the server by sending an access request. The server receives this request and checks in the database whether the information is correct or not, and sends appropriate response to the client. If this user is authenticated, then the client will authorize access to the system that the client wants to connect to.

The server will also be notified when the session starts and stops, so that the user can be billed accordingly to the session time. The data can be also used for statistical purposes.

From a packet level point of view, a common scenario for an RADIUS authentication packet exchange is an Access-Request packet containing a username and password in its attributes field sent from a client, this then followed by an Access-Accept packet or an Access-Reject packet as the response from the server.

2.2.4 Shared Secret

A shared secret is a password used between a RADIUS server and a RADIUS client to mutually verify identity. Both the RADIUS server and the RADIUS client must be configured with the same shared secret, it is required for all RADIUS protocol communications. However, the shared secret itself is never sent across the network.

The shared secret can be up to 128 bytes long, is case sensitive, and can contain alphanumeric and special characters.

2.2.5 Diameter

The RADIUS protocol has been successfully and widely deployed to provide Authentication, Authorization, and Accounting (AAA) services, however, there are several shortcomings of the RADIUS protocol that have limited its ability to handle the ever-expanding requirements of AAA services. To address these limitations, the Diameter protocol was designed as a successor of the RADIUS protocol.

The basic concept of Diameter is to provide a base protocol as a framework that can be extended in order to provide AAA services to applications such as network access or IP mobility. The Diameter base protocol is defined by RFC 3588 [15], which describes the basic message formatting, attribute-value pairs formats, hop-by-hop security, error reporting, etc.

Comparing with the RADIUS protocol, the improvement of the Diameter protocol differs in the following aspects.

Transport protocol
The Diameter protocol operates over Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP). TCP and SCTP are connection-oriented transport protocols with flow control and congestion avoidance mechanisms. Whereas the RADIUS protocol is built on User Datagram Protocol (UDP). UDP is a connectionless transport protocol - without any flow control mechanism and without any built-in reliability.

Size of attribute data
The Attribute Length field in a Diameter message is three octets. Therefore its maximum value allows for over 16,777,000 bytes of data for one attribute. While the Attribute Length field of a RADIUS packet is only one octet. Hence the maximum length of an attribute data is only 255 bytes.

Number of concurrent pending messages
The End-to-End Identifier field in the header of the Diameter packet is four octets, allowing over 4 billion outstanding messages from a Diameter client. As obvious from the header of the RADIUS message, the Identifier field is only one octet, imposing a maximum of 255 outstanding messages between a RADIUS client and a RADIUS server.

Failure detection
By using a connection-oriented transport layer and Diameter keepalive messages, a Diameter node can detect the local failure of a peer. However, when using RADIUS and UDP, when the client does not receive a timely response to a given RADIUS request, the client cannot distinguish the cause of the failure. Specifically it does not

know if the message was lost, if the server has failed, or other reasons caused the failure.

Discarding of packets
The RADIUS protocol specifies that packets are silently discarded for a range of error conditions. In such cases, following a timeout the RADIUS client will assume that the RADIUS server did not receive the packet, and will uselessly retransmit the packet (one or more times) before finally abandoning the request. In contrast, the Diameter protocol returns a response for all but a few error conditions.

Server-initiated messages
Diameter is a peer-to-peer rather than a client-server protocol, hence it supports server-initiated messages. While the RADIUS protocol does not allow a server to send unsolicited messages to clients.

End-to-end security
The Diameter protocol offers not only hop-by-hop security, but also end-to-end security. Data encryption and digital signatures are used to ensure the confidentiality and integrity of selected attribute-value pairs. However, the RADIUS protocol offers only hop-by-hop security, thus there is no facility for securing attribute-value pairs between endpoints.

Support for vendor-specific commands
The Diameter protocol supports both vendor-specific attributes and vendor-specific commands. The RADIUS protocol supports vendor-specific attributes, but not vendor-specific commands.

Alignment requirements
The Diameter protocol requires all attributes to be aligned on 32-bit boundaries. Individual 32-bit fields in the Diameter message header and attribute-value pair header also must be aligned on 32-bit boundaries. The RADIUS protocol does not have alignment requirements, which can add an unnecessary burden on many processors. In a RADIUS packet, all fields within the header and attributes must be treated as byte aligned characters.

Shared secret is not mandatory
The RADIUS protocol requires a shared secret exist between two peers, even if IP Security is employed over a local communication link. The Diameter protocol can provide secure communication between peers by using either Transport Layer Security (TLS) or IP Security.

2.3 Cryptographic Issues

2.3.1 One-Time Password

A one-time password (OTP) is a password that is used only once. After a one-time password used, then it is no longer valid.

A one-time password system generates a series of passwords for logon and other purposes. The method of creating passwords is such that, it is extremely difficult to calculate the **next** password in the series, even given the previous passwords. The logon system always expects a new one-time password at the next logon. In this way, the possibility of "replay attacks" is eliminated.

A variety of cryptographic schemes, such as S/KEY [16] and HOTP [17], can be used to generate one-time passwords from assigned secrets, i.e. seeds or secret keys.

2.3.2 Multifactor Authentication

In a communication system, authentication verifies that messages really come from the stated source. To authenticate a human user, authentication methods are generally classified into three categories:

1. Something the user knows
This is the most common method of authentication today. A password, a pass phrase, or a personal identification number (PIN) is required for this authentication method.

2. Something the user has
This method uses an object that the user **has** to prove its identity, for example, the object could be an ID card, security token, a mobile phone, etc.

3. Something about the user
Fingerprint, retinal pattern, voice pattern, or other biometric identifier could be used in this authentication method.

Using more than one factor of authentication is called multifactor authentication. It is also called “strong authentication”. In some scenarios, a combination of two methods, for instance, a password and a security token are used to provide “two-factor authentication”.

2.3.3 SHA-1

A secure hash function reduces a message of arbitrary length to a fixed length message digest, which is very unlikely to be the same for any other messages. The word “secure” indicates that it is not possible to forge a message which to have given hash value, nor to create two messages with the same hash value.

SHA-1 [18] has been one of the most commonly used cryptographic hash functions. It belongs to the Secure Hash Algorithm (SHA) family. The first member of the family is officially called SHA. However, it is often called SHA-0 to avoid confusion with its successors.

SHA-1, the first successor to SHA, was defined by the United States government standards agency National Institute of Standards and Technology (NIST) in 1995 as a Federal Information Processing Standard (FIPS).

SHA-1 is deployed in a large variety of popular security applications and protocols, including SSL, TLS, SSH, PGP, and IPSec. SHA-1 is considered to be the successor to MD5 [19], an earlier widely-used hash function. The replacement of MD5 was necessary because of weaknesses, which were discovered in MD5.

SHA-1 takes a (long) message (with a maximum size of 2^{64} bits) as input and produces a fixed length (160 bits) digest as output. One important application of SHA-1 is verification of message integrity. The 160-bit message digest generated by SHA-1 can be used to detect whether or not any changes have been made to a message.

The following are some examples of SHA-1 digests:

```
SHA-1("What a nice day") ==
"139bf6917996ab3933bedfa7b55a0e75ea7e398b"
```

Even a small change in the message will result in a completely different hash. For example, let’s change ‘d’ to ‘b’:

```
SHA-1("What a nice bay") ==
"79eaab396f6d1988148dcee6185ec72f567aae81"
```

The hash of the zero-length string is:

```
SHA-1("") ==
"da39a3ee5e6b4b0d3255bfef95601890afd80709"
```

There are four variants were developed in the SHA family, with a slightly different designs and increased output ranges: SHA-224, SHA-256, SHA-384, and SHA-512. Sometimes they are collectively referred to as SHA-2.

Attacks have been found for both SHA-0 and SHA-1, while no attacks have yet been reported on the SHA-2 variants. NIST announced that they planned to phase out the use of SHA-1 by 2010 in favor of the SHA-2 variants [20].

2.3.4 HMAC

A keyed-Hash Message Authentication Code (HMAC) is a type of message authentication code (MAC) calculated using a cryptographic hash function in combination with a secret key.

HMAC may be used to simultaneously verify both the data integrity and the authenticity of a message. Any iterative cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA-1 accordingly.

HMAC is defined in RFC 2104 [21] as follows:

$$HMAC_K(m) = H((K \text{ xor } opad) + H((K \text{ xor } ipad) + m);$$

Where H is the underlying hash function; K is the secret key; m is the message to be authenticated, and + stands for concatenation. *opad* and *ipad* are two one-block-long constants.

The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function and on the size and quality of the key.

2.3.5 DES and AES

The Data Encryption Standard (DES) [22] is a method for encrypting information. It was issued as a Federal Information Processing Standard (FIPS) for the United States in 1976. After this DES was widely adopted in the world.

The DES algorithm is a secret-key encryption scheme. It uses a 56-bit key. This relatively short key length and properties of DES are suspected of offering the U.S. National Security Agency (NSA) a backdoor. Today a 56-bit key is widely considered to be **insufficient**, because it can be cracked by brute force with moderate effort [23] [24].

The Triple-DES (3DES) variant was developed to add more security. 3DES uses a longer key, a total key length of 168 bits, and applies the DES algorithm three times. 3DES has not become popular, as 3DES takes three times as long as DES, for **both** encryption and decryption.

In 2001, the Advanced Encryption Standard (AES) [25] superseded DES as the standard encryption algorithm. AES, also known as Rijndael, is fast in both software and hardware, is relatively easy to implement, and requires little memory. As a new encryption standard, AES is currently being deployed on a large scale.

Strictly speaking, AES is not precisely Rijndael as Rijndael supports a larger range of block and key sizes. AES has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, whereas Rijndael can be specified with key and block sizes in any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits.

Both DES and AES are symmetric ciphers. In a symmetric cipher, both parties must agree on the encryption key in advance. The sender uses that key to encrypt the message. The receiver also uses the **same** key to decrypt the message.

AES and DES are block ciphers, they operate on blocks of fixed length. To encrypt longer messages, several modes of operation may be used. The simplest encryption mode is the electronic codebook (ECB) mode, in which the message is split into blocks and each is encrypted separately. The disadvantage of this method is that identical plaintext blocks are encrypted to identical ciphertext blocks; it does not hide data patterns. In the cipher-block chaining (CBC) mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block is dependent on all plaintext blocks up to that point. Also, to make each message unique an initialization vector (IV) is used.

2.3.6 RSA Algorithm

RSA is an algorithm for public-key encryption. The algorithm was described in 1977 by Ron Rivest, Adi Shamir and Len Adleman at MIT [26]. The letters RSA are the initials of their surnames.

RSA is an asymmetric key algorithm, which involves two keys: public key and private key. The public key is known to everyone and is used to encrypt messages. These messages can only be decrypted by use of the private key. In other words, anybody can encrypt a message, but only the holder of a private key can actually decrypt the message and read it.

3. Design

3.1 System Overview

We consider a system that uses a one-time password based GSS-API mechanism. It consists of three types of parties: the authentication server, the context initiator, and the context acceptor.

An authentication server is an entity providing authentication service. The server-based application running on the authentication server is responsible for managing, and validating the digital identities known to this server. The authentication server is fully RADIUS compliant. By supporting RADIUS the system can be easily integrated with other types of applications and network devices such as VPN, routers, and firewalls.

A context initiator and a context acceptor are entities using GSS-API-aware applications, which acquire security services from underlying security mechanism. As its name implies, a context initiator initiates a security context with a context acceptor in order to establish a secure channel of communication.

Figure 3-1 shows the basic system architecture that contains one authentication server two GSS-API communicating nodes. I use different arrows to represent different relationships.

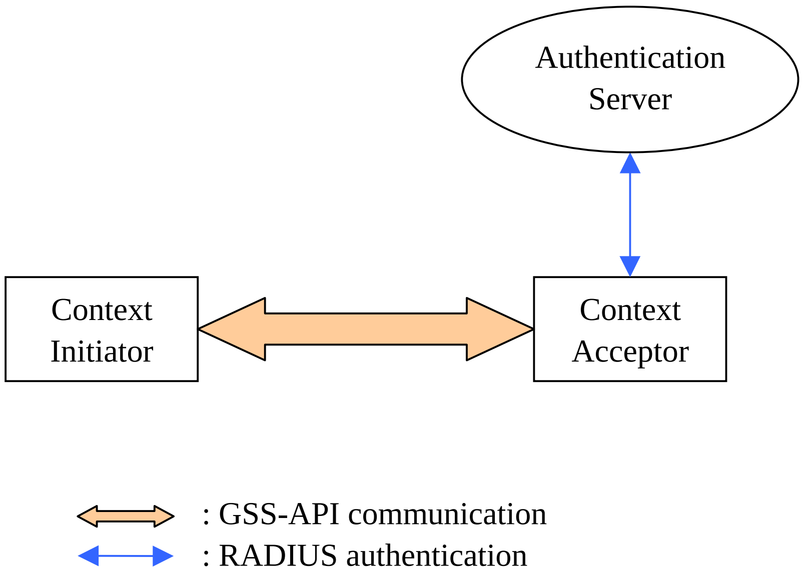

Figure 3-1: Basic system architecture

As shown in Figure 6, the relationship of context initiator and context acceptor is GSS-API peering. Between the authentication server and the context acceptor, there is an authentication relationship. The system can be virtually divided into two subsystems: GSS-API communication subsystem and authentication subsystem.

The GSS-API communication subsystem is a pair of nodes, which establish a common GSS-API security context and can subsequently communicate securely. The authentication subsystem consists of an authentication server and context acceptor(s). Generally speaking, the authentication server acts as a RADIUS server; the context acceptors can be regarded as RADIUS clients. Shared secrets, which are required by the use of RADIUS protocol, are deployed in all context acceptors and authentication servers of the system.

3.2 Applying One-time Passwords

During establishment of a security context, the context initiator is authenticated to the context acceptor by providing a one-time password. From a user's point of view, there are several different ways to acquire one-time passwords. For instance:

- Using a token device, and input a server provided challenge to calculate a one-time password.

- Using a synchronized token device to generate one-time passwords. The token device utilizes a counter and outputs a hashed value based upon this counter. The counter should be synchronized with the server and is controlled by a server-defined offset.

- By providing username and password, a user can request that server generated one-time passwords be sent to a predefined user device such as a mobile phone.

All the methods above are two-factor authentication solutions that require a user to have a device PIN code or a username/password pair (i.e. something the user knows) and be in possession of a specific device (i.e. something the user has).

PortWise have developed web-based Soft Token that is delivered through a web browser to the end user when requested. It uses either ActiveX or Java depending on what is available on the Internet connected device. The Soft Token provides a two-factor authentication solution without being dependent on a specific hardware device.

Our proposed GSS-API mechanism is totally independent of how one-time passwords are generated and/or transferred from authentication server to user (i.e. context initiator). The GSS-API routine gss_acquire_cred are responsible for prompting the user to input a one-time password.

3.3 GSS-API Security Context Establishment

The security context in each peer is a GSS-API data structure that contains shared state information, for instance, cryptographic keys and state flags.

Prior to use of GSS-API routines, each peer should be locally authenticated. It requires that GSS-API callers have access to the keys, which are deployed in each peer.

A context initiator calls the gss_init_sec_context routine, which returns an initial context token to establish the shared information that makes up the security context. The token is an opaque data type that contains state information and a cryptographically protected user-provided one-time password (OTP). The token is encapsulated with a token tag, which is encoded in ASN.1 DER [27] format. The token tag consists of token length and the mechanism's object identifier (OID), it enables the token to be interpreted unambiguously at the GSS-API peer. The caller of the gss_init_sec_context routine (i.e. the context initiator) is responsible for transferring the initial context token to the peer node.

On receiving such a token, the peer application passes it to gss_accept_sec_context routine. The gss_accept_sec_context routine decodes the token - extracting state information as well as the one-time password. The context acceptor then authenticates the context initiator with the help of the authentication server. A RADIUS Access-Request packet containing the context initiator's node name and the one-time password will be sent from the context acceptor to the authentication server. Based upon the response of authentication server, the context acceptor will accept or reject the context initiator. That is, if an Access-Accept packet is sent back from the authentication server, the context initiator is authenticated to the context acceptor. If an Access-Reject packet is sent back, the context initiator will be rejected.

Finally, gss_accept_sec_context yields a response token that will be sent to the context initiator. The token is used to inform the context initiator whether the security context has been successfully established or not. In addition, the context initiator can authenticate the context acceptor by checking the information contained in the response token. On receiving the response token, the context initiator calls the gss_init_sec_context routine again to finalize the context establishment.

Figure 3-2 illustrates the above security context initiator authentication process.

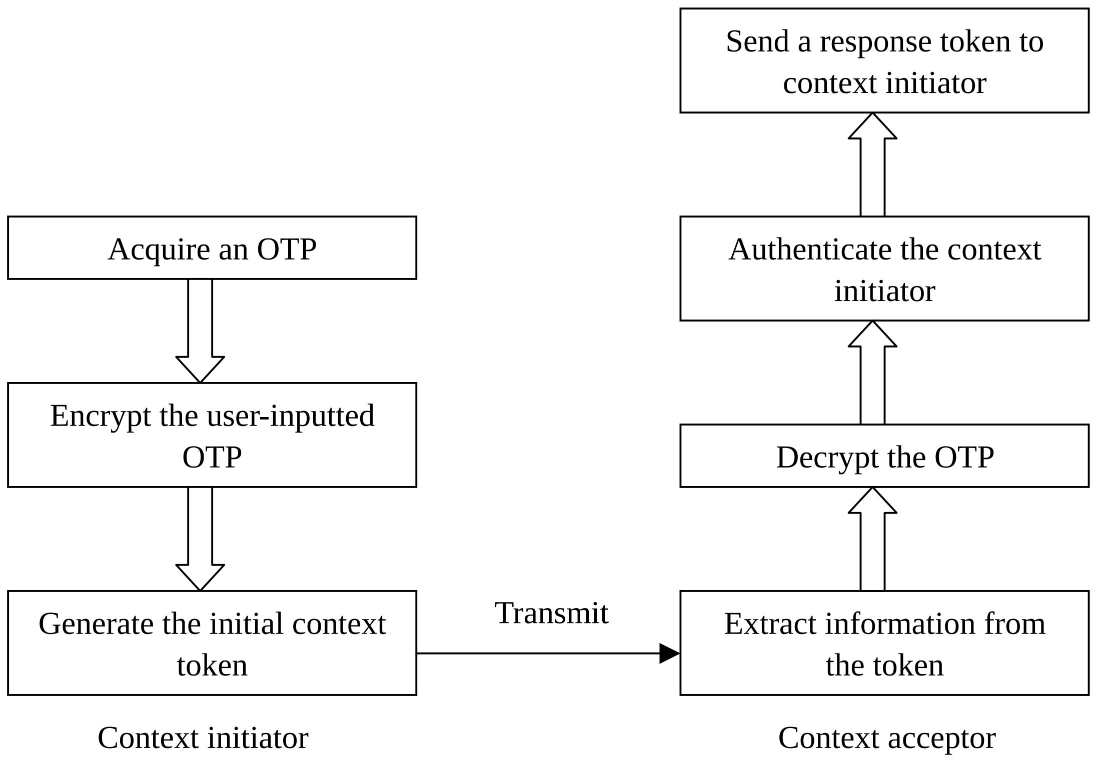
Send a response token to context initiator
Acquire an OTP
Authenticate the context initiator
Encrypt the user-inputted OTP
Decrypt the OTP
Generate the initial context token
Transmit
Extract information from the token
Context initiator
Context acceptor

Figure 3-2: Security context initiator authentication

3.4 Protected Data Communication

After a GSS-API security context is established, peer applications can exchange protected messages each other. Per-message services are invoked to apply data integrity and confidentiality. By specifying the appropriate security context and Quality of Protection (QoP) parameter, gss_get_mic and gss_wrap are called to apply protection. All application data are treated by GSS-API as arbitrary octet-strings.

The gss_get_mic call generates a cryptographic Message Integrity Code (MIC) for supplied message. The token emitted by the gss_get_mic routine, MIC token, will be processed by call to gss_verify_mic routine. The gss_verify_mic call is used to verify that a MIC fits the supplied message. In this routine, a MIC is calculated in the same way as gss_get_mic, and then compared with the MIC contained in the token parameter. If they are the same, then the MIC is verified.

gss_wrap call attaches a MIC and optionally encrypt the specified input message. The conf_req_flag parameter indicates whether gss_wrap is asked to apply confidentiality protection. A non-zero conf_req_flag value means that both confidentiality and integrity services are requested, while a zero conf_req_flag value means that only integrity services is requested.

The application will send the resulting token of gss_wrap, wrap token, to the receiving application. The receiver will pass the received token to the gss_unwrap routine to remove the protection and validate the data. Since the token contains information about the confidential state parameter and the QoP parameter used by gss_wrap, after extract these information from the token, the gss_unwrap routine can properly convert a message previously protected by gss_wrap back to its original form and verifies the embedded MIC.

Figure 3-3 shows the procedure of message protection.

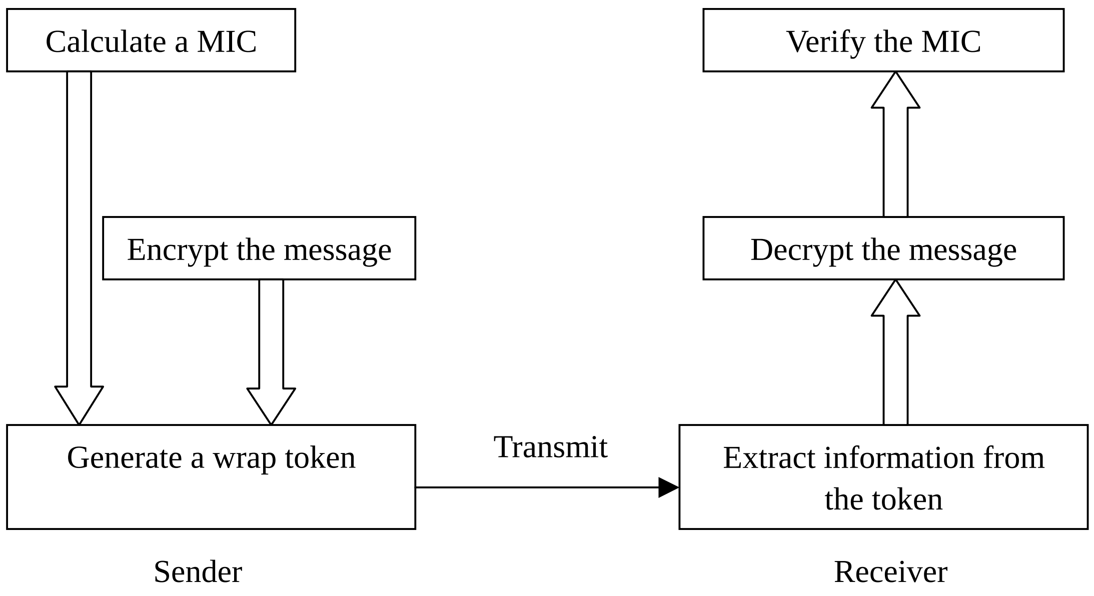

Figure 3-3: Message protection procedure

3.5 Cryptographic Schemes

3.5.1 Protecting One-Time Passwords

As mentioned in section 3.3, the context initiator proves his identity to the context acceptor by providing a one-time password (OTP). The password, contained in an initial context token, must be well protected during transmission in order to prevent user impersonation. Our security mechanism provides two types of encryption schemes to protect the one-time password: public-key mode and secret-key mode.

In public-key mode, the algorithm, which I use for encrypting one-time password, is RSA. The password is encrypted with the context acceptor's public key. The context acceptor decrypts it with his private key. As a one-time password is relatively short, usually a number of 6 to 8 digits, in order to defeat exhaustive search of the password, I put 64-byte long pseudo-random data in front of the password before encryption. In addition, The RSA key size should be at least 1024 bits for the purpose of resistance to brute force attack.

In secret-key mode, I use AES to encrypt the password. The key used in AES algorithm is 128 bits; it made from the first 16 bytes of the shared secret between context initiator and context acceptor. As AES is a symmetric cipher, the same key will be applied for decrypting the one-time password.

3.5.2 Generating Session Keys

A session key is generated, and stored in an established security context of each peer. The session key is used for data protection in current security context. The use of session key comes from the following the considerations:

- The more messages encrypted with a specific key, the easier cryptanalytic attacks are made. The use of session key limits the material processed using a particular long-term key (e.g. the shared secret used in secret-key mode).

- Symmetric cryptography, such as AES, is substantially faster than public-key cryptography [28]. A key, known by both communicating parties, is the prerequisite of using a symmetric cipher.

In public-key mode, the session key is calculated as follows:

Session Key = SHA-1(R + one-time password);

Where + denotes concatenation, R is 64-byte pseudo-random data sent along with the one-time password.

In secret-key mode, the session key is generated as follows, where + stands for concatenation:

Session Key = SHA-1(one-time password + shared secret);

As both peers know the one-time password and R (or shared secret), the same 160-bit session key is calculated in each peer.

3.5.3 Quality of Protection

The Quality of Protection (QoP) parameter of per-message routines allows a choice between several cryptographic algorithms used for encryption data or generating message integrity codes (MICs), if supported by the underlying mechanism. A zero QoP value is used to indicate the default protection, however applications that do not use the default QoP are not guaranteed to be portable across implementations of underlying security mechanisms, or even to inter-operate with different deployment configurations of the same implementation.

In our GSS-API mechanism, the algorithm used for generating a MIC is HMAC-SHA-1. The key used in this algorithm is the session key.

The default algorithm used for message encryption is AES in cipher-block chaining (CBC) mode with a zero initialization vector (IV). The key used in AES algorithm is taken from the first 128 bits of the session key. In the GSS-API specification, the QoP value for using default algorithm, 0, is defined as a symbolic constant: `GSS_C_QOP_DEFAULT`.

Specifying non-default QoP more or less defeats the purpose of using the GSS-API, because it limits the portability of an application. Nonetheless, DES (electronic codebook mode) is also supported by our security mechanism for demonstration purposes. The key used for DES algorithm is taken from the first 64 bits of the session key. Note that there are 8 bits of parity used in the 64-bit key, which means that the effective key length is 56 bits. In my GSS-API implementation, the QoP value for using DES is 1, and is defined as a symbolic constant: `GSS_PORTWISE_QOP_DES`.

3.6 Token Formats

The basic message unit in the GSS-API is the token. Applications using GSS-API communicate with each other by using tokens, both for security context establishment and exchanging data. Tokens are declared as gss_buffer_t data type (see section 4.1.1) and are opaque to applications.

There are two types of tokens: context-level tokens and per-message tokens. Context-level tokens are used when a security context is established. An initial context token is used to initiate a context. A response token is used to reply the context initiator.

Per-message tokens are used to provide protection services on data after a security context has been established. MIC tokens are used to provide integrity service. Wrap tokens are used to provide both integrity service and confidentiality service.

3.6.1 Initial Context Token

An initial context token contains information used to initialize a security context. It is generated by the gss_init_sec_context routine, and is transmitted from context initiator to context acceptor. The following Table 3-1 listed all data fields contained an initial token, in order.

Field Name	Length
Token tag	*Variable*
Token ID	1 byte
Length of initiator’s name	1 byte
Initiator’s name	*Variable (Max. 128 bytes)*
Length of acceptor’s name	1 byte
Acceptor’s name	*Variable (Max. 128 bytes)*
Length of password	2 byte
Password	*Variable*
Context flags	4 bytes
Signature	20 bytes

Table 3-1: Initial context token format

Token tag
The Token tag field contains token length and the mechanism’s object identifier (OID). The encoding format for the token tag is derived from ASN.1 DER. This field consists of the following elements, in order:

- 0x60 (Tag for ASN.1 SEQUENCE)
- Token length (Length of subsequent data)
- 0x06 (Tag for object identifier)
- Object identifier length
- Object identifier

The token tag enables the token to be interpreted unambiguously at the context acceptor. It is a requirement of an initial context token, specified in the GSS-API specification. The ASN.1 structure token tag is not required for other tokens.

Token ID
In public-key mode, this field should contain the value 0x00. While in secret-key mode, the value is 0x01.

Length of initiator’s name
It specifies the length in byte of initiator’s name.

Initiator’s name
This field contains the name of context initiator.

Length of acceptor’s name
It specifies the length in byte of acceptor’s name.

Acceptor’s name
This filed indicates the name of context acceptor.

Length of password
It specifies the length in byte of subsequent Password field. The value is in big-endian form.

Password
The Password field stores an encrypted one-time password that will be used to authenticate context initiator to context acceptor. In public-key mode, the password is left-padded with 64-bytes random data before encryption.

Context flags
The Context flags field is 4 bytes. It is a bit-mask in big-endian form, and contains

various independent flags of the security context. Each flag represents a specific service option supported by the context. All flags are logically-ORed together to form the bit-mask. Flags and corresponding symbolic names are defined in the GSS-API specification.

Signature
This field contains a 20-byte keyed-hash message authentication code (HMAC). This field is used to verify the data integrity and authenticity. The signature is calculated as follows, where + denotes concatenation:

$$\begin{aligned}\text{Signature} = \text{HMAC-SHA-1}_K(&\text{Token ID} + \text{Length of initiator's name}\\ &+ \text{Initiator's name}\\ &+ \text{Length of acceptor's name}\\ &+ \text{Acceptor's name}\\ &+ \text{Length of password}\\ &+ \text{Password}\\ &+ \text{Context flags});\end{aligned}$$

The key used in calculation, K, is calculated the same way as the session key.

3.6.2 Response Token

The response token is transmitted from context acceptor to context initiator. It is generated by the gss_accept_sec_context routine. It contains information used to inform the context initiator whether the security context has been successfully established or not. Data format of a response token is listed in order in the following Table 3-2.

Field Name	Length
Token ID	1 byte
Status	1 byte
Signature (optional)	20 bytes

Table 3-2: Response token format

Token ID
For a response token, this field should contain the value 0x02.

Status
A zero value indicates the context initiator has successfully authenticated to the context acceptor. A non-zero value means the context acceptor rejected the context initiator.

Signature (optional)
This field contains a 20-byte HMAC. It only exists in the response token when context initiator has successfully authenticated to the context acceptor. The signature is calculated as follows, where + stands for concatenation:

$$\text{Signature} = \text{HMAC-SHA-1}_K(\text{Token ID} + \text{Status});$$

The key used in calculation, K, is calculated the same way as the session key.

3.6.3 MIC Token

Message Integrity Code (MIC) tokens are transmitted between GSS-API peers after a security context was established. The MIC token is generated by the gss_get_mic routine. It contains a MIC that proves integrity and data origin of the supplied message buffer. The data format of a MIC token is listed in Table 3-3.

Field Name	Length
Token ID	1 byte
Integrity algorithm	1 byte
Sequence number	4 bytes
MIC	20 bytes

Table 3-3: MIC token format

Token ID
This field in a MIC token should contain the value 0x03.

Integrity algorithm
This field indicates the algorithm used for generating the MIC. The value should consist with the QoP parameter used in the gss_get_mic call. Currently, SHA-1 is the only supported integrity algorithm; hence this field should contain the value 0.

Sequence number
This field stores a 32-bit unsigned integer value in big-endian byte order. The sequence number is used for message replay and out-of-sequence detection.

MIC
This field contains a 20-byte message integrity code. The MIC is calculated as follows, where + denotes concatenation, K stands for the session key:

$$
\begin{aligned}
\text{MIC} = \text{HMAC-SHA-1}_K(&\text{Token ID} + \text{Integrity algorithm} \\
&+ \text{Sequence number} \\
&+ \text{Message buffer});
\end{aligned}
$$

The message buffer consists of 2 parts: message length and content. The first 4 bytes of the message buffer are the length filed; it stores a 32-bit unsigned integer in little-endian form, which specifies the number of bytes of the following message content. The fifth byte of a message buffer is the beginning of the content.

3.6.4 Wrap Token

Wrap tokens are transmitted between GSS-API peers after a security context was established. The wrap token is generated by the gss_wrap routine, and provides integrity services and confidentiality service (optional) to a message buffer. The following Table 3-4 listed all data fields contained a wrap token, in order.

Field Name	Length
Token ID	1 byte
Integrity algorithm	1 byte
Confidentiality algorithm	1 byte
Sequence number	4 bytes
MIC	20 bytes
Message	*Variable*

Table 3-4: Wrap token format

Token ID
This field of a wrap token should contain the value 0x04.

Integrity algorithm
This field indicates the algorithm used for generating the MIC.

Confidentiality algorithm
This field indicates the algorithm used for encrypting the message. The value and corresponding algorithm are listed in Table 3-5:

Value	Algorithm
0x00	AES (CBC mode)
0x01	DES (EBC mode)
0xff	*None*

Table 3-5: Confidentiality algorithms

Specifying 0xff means the message is not encrypted.

Sequence number
This field contains a 32-bit unsigned integer value in big-endian byte order. The sequence number is used for message replay and out-of-sequence detection.

MIC
The MIC field contains a 20-byte Message Integrity Code (MIC) of the message. The MIC is calculated as follows, where + denotes concatenation, K stands for the session key:

$$\begin{aligned}\text{MIC} = \text{HMAC-SHA-1}_K(&\text{Token ID} + \text{Integrity algorithm}\\ &+ \text{Confidentiality algorithm}\\ &+ \text{Sequence number}\\ &+ \text{Message buffer});\end{aligned}$$

Message
The Message field holds the (encrypted) message buffer. Before AES encryption, the original message buffer should be right-padded with null characters (0x00) to a 16-byte boundary. While using DES, the original message buffer should be right-padded with null characters (0x00) to an 8-byte boundary.

4. Implementation

The one-time password based GSS-API mechanism was implemented on Red Hat Linux 9. The security mechanism is provided as a shared library, which contains all the routines defined in the GSS-API version 2 specification. The shared library was written in C and complied with GNU C Compiler (gcc).

4.1 Data Types

4.1.1 Basic Data Types

The following basic data types are described in the GSS-API C-language binding specification.

Integer
GSS-API routines use `OM_uint32` as a platform-independent 32-bit unsigned integer.

Octet-string
Many GSS-API procedures take arguments and return values using contiguous octet-strings. All such data is passed between the GSS-API and the caller using the `gss_buffer_t` data type, which is a pointer to a buffer structure. The buffer structure consists of a length field that contains the total number of bytes in the data, and a value field, which contains a pointer to the actual data.

The following is the definition of the buffer structure `gss_buffer_desc` and the `gss_buffer_t` pointer:

```
typedef struct gss_buffer_desc_struct {
   size_t   length;
   void     *value;
} gss_buffer_desc, *gss_buffer_t;
```

Object identifier
Some GSS-API routines take parameters of the object identifier (OID). The `gss_OID` data type is a pointer to an OID descriptor, which is a type containing ISO-defined tree-structured values. OID is used by GSS-API callers to select underlying security mechanism and to specify name types. A `gss_OID` type has the following structure:

```
typedef struct gss_OID_desc_struct {
   OM_uint32   length;
   void        *elements;
} gss_OID_desc, *gss_OID;
```

Object identifier set
Certain GSS-API routines take parameters of the type `gss_OID_set`, which represent one or more object identifiers. A `gss_OID_set` object has the following structure:

```
typedef struct gss_OID_set_desc_struct {
   size_t   count;
   gss_OID  elements;
} gss_OID_set_desc, *gss_OID_set;
```

4.1.2 Implementation-dependent Data Types

There are three implementation-dependent data types defined in GSS-API header file. They are `gss_name_t`, `gss_cred_id_t` and `gss_ctx_id_t`. In the header file, the following typedef definitions map these types to their respective built-in data structures.

```
typedef struct gss_name_struct *gss_name_t;
typedef struct gss_cred_id_struct *gss_cred_id_t;
typedef struct gss_ctx_id_struct *gss_ctx_id_t;
```

Name
The `gss_name_t` data type is used to refer to a name data structure. The data structure for `gss_name_struct` is defined as follows:

```
typedef struct gss_name_struct {
   size_t length;
   char *value;
   gss_OID type;
} gss_name_desc;
```

The `length` field is a `size_t` object, which stores total number of bytes in the name. The value field contains a pointer to the name string. The `type` of a name is a `gss_OID` constant. Various name type OID constants are specified in the GSS-API C-language binding specification. In my GSS-API implementation, the `gss_OID` constant `GSS_PORTWISE_NT_NODE_NAME` is an implementation-specific name type, which is used to identify the "node name" type.

Credential

The `gss_cred_id_t` data type is used to identify a GSS-API credential data structure. The data structure for `gss_cred_id_struct` is defined as follows:

```
typedef struct gss_cred_id_struct {
   gss_OID mech;           /* Mechanism ID */
   struct _gss_portwise_cred_struct *portwise;
} gss_cred_id_desc;
```

The `mech` field specifies the mechanism OID. The data structure for `_gss_portwise_cred_struct` is defined as follows:

```
typedef struct _gss_portwise_cred_struct {
   gss_name_t cred_name;
   gss_cred_usage_t usage;
   char *otp;
   char *shared_secret;
   RSA *private_key;
} _gss_portwise_cred_desc, *_gss_portwise_cred_t;
```

Below we examine these fields in the structure:

`cred_name`

`cred_name` indicates the name holding this credential. The name type of `cred_name` should be “node name”.

`usage`

`usage` indicates how the credential can be used. There are three vales:

- `GSS_C_BOTH`: The credential may be used either to initiate or accept security contexts
- `GSS_C_INITATE`: The credential will only be used to initiate security contexts.
- `GSS_C_ACCEPT`: The credential will only be used to accept security contexts.

`otp`

This field contains a pointer to the user-provided one-time password, which is used to initiate a security context.

`shared_secret`

This field contains a pointer to the shared secret.

`private_key`

This field contains a pointer to the RSA private key.

Context

The gss_ctx_id_t data type is used to identify one end of a GSS-API security context. The data structure for gss_ctx_id_struct is defined as follows:

```
typedef struct gss_ctx_id_struct {
   gss_OID mech;              /* Mechanism OID */
   struct _gss_portwise_ctx_struct portwise;
} gss_ctx_id_desc;
```

The mech field specifies the mechanism OID. The data structure for _gss_portwise_ctx_struct is defined as follows:

```
typedef struct _gss_portwise_ctx_struct {
   gss_name_t sname;       /* Source name */
   gss_name_t tname;       /* Target name */
   OM_uint32 flags;        /* Context flags */
   ctx_secret secret;      /* Session key */
   unsigned int send_seq;  /* Seq no. of send */
   unsigned int recv_seq;  /* Seq no. of recv */
   time_t expire;          /* Expire time */
   int local;              /* Locally initiated? */
   int open;               /* Context established? */
} _gss_portwise_ctx_desc, *_gss_portwise_ctx_t;
```

Below we examine these fields in the structure:

sname

sname is a gss_name_t object, which represent the name of context initiator. The name type of sname should be "node name".

tname

The tname field is used to store the name of context acceptor. It is a gss_name_t object. The name type of tname is "node name".

flags

The flags field is a bit-mask. It contains various independent flags of the security context. Each flag represents a specific service option supported by the context. All flags are logically-ORed together to form the bit-mask. Flags and corresponding symbolic names are defined in GSS-API specification.

secret

This field contains a 20-byte session key of the context. The data type ctx_secret is defined as follows:

```
typedef unsigned char ctx_secret[20];
```

`send_seq`
This field stores the latest sequence number of generated MIC tokens and wrap tokens. The initial value is 0. After generating a MIC or a wrap token (`gss_get_mic` or `gss_wrap`), this sequence number is incremented by 1.

`recv_seq`
This field indicates the latest sequence number of processed MIC tokens and wrap tokens. The initial value is 0. After processing a MIC or a wrap token (`gss_verify_mic` or `gss_unwrap`), this sequence number is incremented by 1.

`expire`
The `expire` field indicates the time when will the context be expired.

`local`
This field is used to indicate if the GSS-API caller is the context initiator. A non-zero value means the security context is initiated locally, while a zero value indicates the peer is the context acceptor. In order to keep consistent data types used the GSS-API C-language binding specification, I choose to use an integer to represent the status, rather than using a Boolean type object.

`open`
As like `local`, I use an integer to represent the status of context establishment. A non-zero value indicates the security context is fully established. A zero value means the procedure of context establishment is not finished, a token is expected from the peer application.

4.2 Node Configuration

In our demonstration system, each node is configured by placing information in plain text configuration files. Context initiators should be configured properly for GSS-API communication. Context acceptors should be configured as RADIUS clients for authentication.

4.2.1 node.conf

The configuration file in context initiator is `node.conf`. This file contains the following settings.

- "`nodename`" specifies the name of the node with the maximum length of 128 characters.

- "`password`" is used to indicate whether or not the node is going to employ one-time passwords for authentication. A specific character string "O-T-P" is regarded as the "sign" of using one-time passwords by the security mechanism. Other character strings will be regarded as the static password of the node. The mechanism supports a password with the maximum length of 128 characters, which is also the maximum length of user password supported by RADIUS.

- "`authmode`" specifies the authentication mode: "public-key" mode or "secret-key" mode.

Figure 4-1 shows an example of `node.conf`.

```
nodename        client01
password        O-T-P
authmode        public-key
```

Figure 4-1: An example of node.conf

In the above example, "`client01`" is the name of the node. "`O-T-P`" indicates that client01 will use one-time passwords for authentication. The node is going to use public-key authentication mode.

`node.conf` is placed in `.mygss` directory in a user's home directory. This file is owned by the user and only the owner can access this file.

4.2.2 radius-client.conf

A context acceptor should be configured as a RADIUS client. The main configuration file of RADIUS authentication is `radius-client.conf`. This file contains the following settings.

- "`authserver`" specifies the RADIUS server address and port number to use for authentication requests. The RADIUS server address can be in either IP address format or domain name format. This configuration item can appear more than one time. If multiple servers are defined, they are tried in a round robin fashion if one server is not answering. The first RADIUS server is always the first to be tried.

- "`servers`" specifies the file of peers and their associated shared secret used for communication between this RADIUS client and each named server. This file can only be accessed by authorized user in order to keep shared secrets safe.

- "`radius_timeout`" specifies time in second to wait for a reply from the RADIUS server.

- "`radius_retries`" specifies how many attempts to send a request before trying the next server.

- "`bindaddr`" specifies the local address from which radius packets have to be sent.

- "`seqfile`" specifies the name of a file which holds current sequence number for communication with the RADIUS server.

Figure 4-2 shows an example of `radius-client.conf`.

```
authserver        213.100.19.82:1812
authserver        auth.example.com:18120

servers           ./server.conf

radius_timeout    10
radius_retries    3

bindaddr          213.100.19.105

seqfile           /var/run/radius.seq
```

Figure 4-2: An example of radius-client.conf

In the above example, the node will send authentication requests to a RADIUS server 213.100.19.82, port 1812. If this server has not answered after 3 attempts, then authentication requests will be sent to the host with the domain name of “auth.example.com” and port number is 18120.

Figure 4-3 shows an example of `server.conf`.

```
213.100.19.82        Th!sIsAl0ngSh@redSec-ret
auth.example.com     red_SKY@N!gth,UtdDe1ight
```

Figure 4-3: An example of server.conf

`radius-client.conf` should be placed in the `/etc/mygss` directory. Only authorized user(s) can access this file. The file holding the shared secret (`server.conf`) and the file holding the sequence number for RADIUS communication (`radius.seq`) could be placed anywhere as long as the GSS-API caller knows where they are located.

4.3 Key Deployment

4.3.1 Public Key

A context acceptor’s RSA public key is deployed in all its client nodes (context initiators). The key is stored in a file of PEM format [29]: it consists of base64 encoded ASN.1 DER encoded key and additional header and footer lines.

The PEM public key format uses the following header and footer lines:

```
-----BEGIN RSA PUBLIC KEY-----
-----END RSA PUBLIC KEY-----
```

The public key file is named after the name of context acceptor, and placed in the `/etc/mygss/public` directory. Any user can read public key file, though only authorized user(s) can modify it.

4.3.2 Private Key

The RSA private key is deployed in context acceptor. The key is saved in a file of PEM format. The PEM private key format uses the following header and footer lines:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

The private key file is named after the name of context acceptor, and placed in the `/etc/mygss/private` directory. The private key file should be well protected. Only the user who runs the application can access the file.

4.3.3 Shared Secret

In secret-key mode, a shared secret is deployed in both GSS-API communicating peers. The file containing a shared secret is named after the context acceptor, and is placed in the `/etc/mygss/secret` directory. The shared secret file should be well protected. Only the user who runs the GSS-API-aware application can access the file.

4.3 Error Handling

In order to report status information, every GSS-API function returns a GSS status code and a mechanism-specific status code to the callers.

GSS status codes are returned by GSS-API routines as their `OM_uint32` function value. These codes indicate errors that are independent of the underlying mechanism(s). A GSS status code can indicate a single fatal API routine error and a single calling error. Moreover, by setting the supplementary status bits, a GSS status code could also indicate additional status information. The values of calling errors, routine errors, and supplementary info are defined in the GSS-API specification.

Figure 15 shows how a GSS status code is organized.

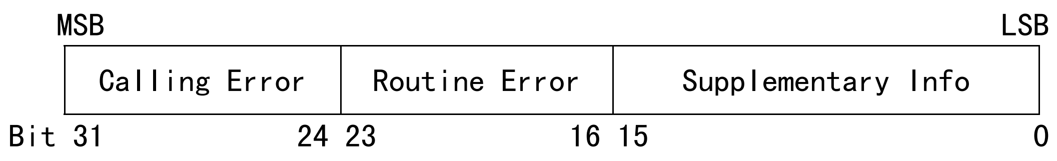

Figure 4-4: The structure of GSS status code

The GSS status code `GSS_S_FAILURE` is used to indicate that the underlying mechanism detected an error for which no specific GSS status code is defined. In such cases, the mechanism-specific status code will provide more detailed information about the error.

GSS-API routines will always set a `minor_status` parameter as mechanism-specific status code. The GSS-API implementation defines the following status codes:

Name	Value	Meaning
`GSS_PORTWISE_NO_ERROR`	0	No mechanism-specific error
`GSS_PORTWISE_AUTH_FAILED`	1	RADIUS authentication was failed
`GSS_PORTWISE_BAD_MSG_LEN`	2	Message is too long to process
`GSS_PORTWISE_BAD_CONF`	3	Fail to acquire node configuration.
`GSS_PORTWISE_PUB_KEY_ERR`	4	Fail to read public key
`GSS_PORTWISE_PRIV_KEY_ERR`	5	Fail to read private key
`GSS_PORTWISE_BAD_RSA_KEY`	6	Invalid RSA key
`GSS_PORTWISE_SECRET_ERR`	7	Fail to read shared secret
`GSS_PORTWISE_RC_CONF_ERR`	8	Fail to read RADIUS client configuration files
`GSS_PORTWISE_RSA_ENC_ERR`	9	RSA encryption error
`GSS_PORTWISE_RSA_DEC_ERR`	10	RSA decryption error

Table 4-1: Mechanism-specific status codes

In the implementation, the `gss_status_code` data structure is used to hold information about a status, including an error value, along with a status name and its description. The data structure is defined as follows:

```
typedef struct gss_status_code {
   OM_uint32 code;
   const char *name;
   const char *text;
} STATUS;
```

The `gss_display_status` routine is responsible for providing textual information (i.e. error description) of a GSS-API status code.

4.3 External Libraries

Four external libraries were used to facilitate the implementation of the proposed GSS-API mechanism:

libcrypto
I used OpenSSL crypto library [30] for RSA and DES encryption/decryption, and generating cryptographically strong pseudo-random data.

libaes
I used Dr. Brian Gladman's AES implementation [31] as a library for AES encryption/decryption. The code implements both AES and Rijndael. It is heavily optimized for AES (fixed block size of 128 bits), especially for the 128-bit key size.

libmhash
libmhash [32] is a free library which provides a uniform interface to a large number of hash algorithms. I used this library to generate SHA-1 message digests.

libradiusclient-ng
libradiusclient-ng [33] is a RADIUS protocol compliant library. It encapsulates the communication between RADIUS clients and server and provides an interface to the operations need to build RADIUS packets.

With the help of these four libraries, the development of the proposed GSS-API mechanism was significantly speed up.

4.4 Compilation and Building

The package of the proposed GSS-API mechanism implementation contains the following main source code files, which are listed in Table 8.

Filename	Size (bytes)	Description
Directory: libmygss/		
gssapi.h	18216	GSS-API header file
Internal.h	654	Definitions of internally used data structures, etc.
asn1.c	654	Functions of ASN.1 DER encoding/decoding
context.c	12234	Context-level routines
cred.c	6106	Credential management routines
error.c	6605	Error handling function
mech.h	4301	Internal symbols
mech.c	1633	Function selection depending on mechanism
misc.c	5496	Supporting routines
msg.c	3601	Per-message routines
oid.c	1633	Definitions of OIDs
name.c	1287	Name manipulation routines
utils.h	1132	Header file of some helper functions
utils.c	1816	Some helper functions
Directory: libmygss/portwise/		
portwise.h	5564	Header file for PortWise-specific routines
pw_internal.h	3234	Definitions of internally used data structures, etc.
context.c	12413	PortWise-specific context-level routines
cred.c	4254	PortWise-specific credential management routines
error.c	2059	PortWise-specific error handling function
msg.c	8083	PortWise-specific per-message routines
name.c	2778	PortWise-specific name manipulation routines
oid.c	902	Definitions of PortWise-specific OIDs
utils.c	6514	Some auxiliary functions

Table 4-2: Main source code files

GNU Make [34] is used to automate the compilation and building process. The source codes are built into a shared library using GNU C Compiler (gcc). All object files are complied with the `-fPIC` option to enable “position independent code” generation; this is a requirement for shared libraries. The following command is used in the `Makefile` to generate the shared library: libmygss.

```
gcc -g -shared -Wl,-soname,libmygss.so.0 \
-o libmygss.so.0.0 $(ALL_OBJS) $(LIBS)
```

5. Evaluation

5.1 SAP GSS Test Tool

GSSTest is a GSS-API interoperability test tool, which was developed by SAP [35]. Although the primary purpose of GSSTest is interoperability tests focusing on SAP distributed applications, it is useful to all implementers of a GSS-API mechanism to verify specification conformance, implementation robustness, consistency in operations, and to measure the performance of one's own GSS-API mechanism implementation.

GSSTest expects a GSS-API mechanism to be supplied as a shared library. This shared library will be dynamically loaded at runtime through the platform-specific runtime linker. All tests and all context establishments are performed locally, i.e. within the same program.

I configured the GSS-API mechanism to use secret-key mode. To make the testing process run smoothly, the mechanism was shifted to use a static password for RADIUS authentication. The mechanism reads the static password from the node configuration file, in this way, the delay of waiting for user input of a one-time password was eliminated. I setup the RADIUS server on the same computer that was running GSSTest in order to eliminate network latency.

The result of GSSTest is listed in Appendix 1. These results indicate that the shared library passed all of the GSS-API result tests. Hence the GSS-API mechanism implementation conforms to the GSS-API version 2 specification. The results also show that the performance of the shared library satisfies the performance requirements of interoperability with SAP's Secure Network Communications (SNC).

In these results, it mentions that the tested GSS-API mechanism requires the use of an external SNC-Adapter. Secure Network Communications (SNC) is a software layer in the SAP system architecture that provides an interface to an external security product. The SNC-Adapter is an intermediate component to attach a third party security software to SNC via a GSS-API version 2 compatible interface. It does **not** change any of the functionality available at the GSS-API level; it only provides a little additional configuration information to SNC about how to deal with this GSS-API mechanism.

5.2 Sample GSS-API Programs

Two sample application programs using this GSS-API shared library were created to demonstrate how the security mechanism works in real life. One program is a client and the other a server. They were developed in C.

5.2.1 mygss-client

The sample client-side program, `mygss-client`, uses a TCP-based socket connection to communicate with its server. It creates a security context with this server, and sends a message string to the server.

In order to test the functionality of GSS-API library, `mygss-client` does the following:

1. Creates a socket connection to the server.
2. Establishes a security context.
3. Wraps a message.
4. Sends the message to sever.
5. Verifies that the message has been signed correctly by the server.

The following GSS-API routines are invoked by `mygss-client`:

```
gss_import_name()
gss_release_name()
gss_init_sec_context()
gss_delete_sec_context()
gss_release_buffer()
gss_verify_mic()
gss_wrap()
gss_get_mic()
gss_display_status()
```

5.2.2 mygss-server

The server-side program, `mygss-server`, performs a security handshake with the client. When the client initiates a security context and sends data, the server accepts the context, verifies the identity of the client, processes the message and finally sends a digital signature for the message back to the client. `mygss-server` is a multi-processed application, once a connection comes in, `mygss-server` forks out a child process to handle the connection and the main server process is able to serve new incoming requests. Hence `mygss-server` can handle multiple connections at the same time.

Specifically, `mygss-server` does the following things:

1. Creates a socket and waits for a connection.
2. Establishes a security context with the client.
3. Get the message from client.
4. Signs the message and sends the signature back.

The following GSS-API routines are invoked by `mygss-server`:

```
gss_import_name()
gss_acquire_cred()
gss_release_name()
gss_accept_sec_context()
gss_release_buffer()
gss_display_name()
gss_delete_sec_context()
gss_get_mic()
gss_unwrap()
gss_verify_mic()
gss_display_status()
```

5.2.3 Program Outputs

The two sample applications can successfully perform tasks as designed. I tested the sample programs and the mechanism with different authentication modes, key sizes:

- Public-key mode with 1024-bit key. The program's outputs are listed in Appendix 3

- Public-key mode with 2048-bit key. The outputs of two sample programs are listed in Appendix 4

- Secret-key mode. The outputs are listed Appendix 5.

In addition, I intentionally produced some error/extreme conditions:

- Wrong password
- Misconfigured configuration file

- Unmatched public/private key pair
- Inaccessible shared secret or public/private key
- “Dead” RADIUS server
- Zero-length message
- Oversized message
- Expired security context

By examining the status codes returned by GSS-API routines and the text message generated by the `gss_display_status` routine, the GSS-API routines invoked by these two sample applications works as expected.

5.2.4 Token Validity Check

In this section, I examine tokens emitted by certain GSS-API routines in order to validate if the mechanism works correctly as designed. With the help of the two sample applications, I inspect an initial context token and a response token exchanged by mygss-client and mygss-server during security context establishment and subsequently a wrap token and a MIC token transferred between the peers. The formats of these tokens are described in Section 3.6. I present these tokens in the form of hexadecimal string.

The following Figure 5-1 illustrates test scenario and sequence of four tokens.

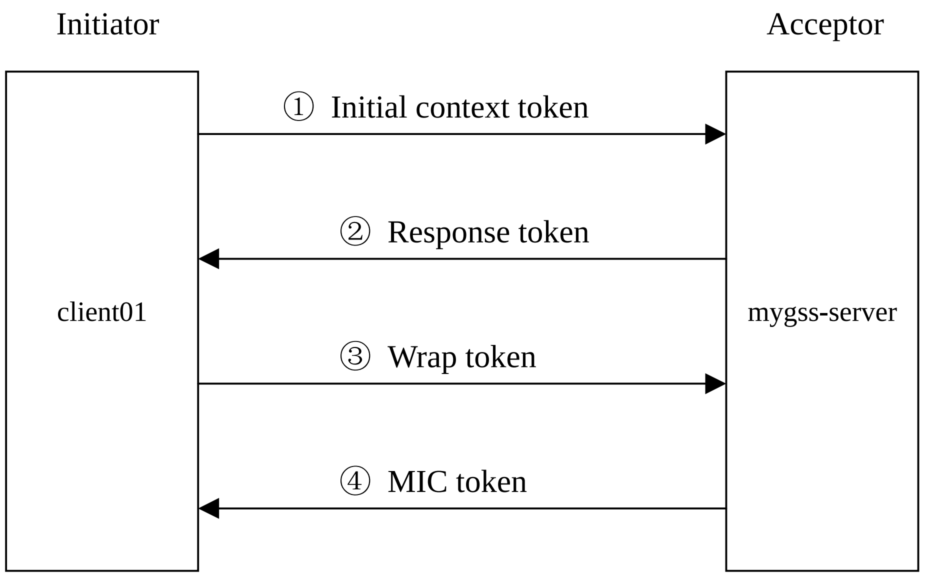

Figure 5-1: Test scenario and token sequence

The configurations and parameters used in this test are listed in Table 5-1.

Item	Value
Name of initiator	“client01”
Name of acceptor	“mygss-server”
Public/private key pair	*See Appendix 2*
One-time password	“990526”
Context flags	0x132
Message	“This is a test”
Confidential state	1 (apply encryption)
QoP state	0 (default protection)

Table 5-1: Configurations and parameters

1. Initial context token

The initial context token generated by the gss_init_sec_context routine is:

```
60 81 bc 06 09 2a 86 48 86 f7 12 01 02 0f 00 08
63 6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d 73
65 72 76 65 72 00 80 46 0e 2c 93 4e a1 88 7a 5f
e5 a9 f1 a8 f8 3a a0 12 41 2c a6 86 52 c3 92 4b
9a e6 62 40 30 db 65 c5 f9 cf a3 56 a3 44 0f 5d
35 70 50 fe 75 60 cb 4e 7a 40 2b 39 7e e7 c4 d2
59 c8 99 52 92 8d 42 d1 71 82 37 eb a1 82 c7 8b
b0 4d d1 9b 77 c9 71 2c b6 22 c4 e8 dd f9 bf b7
70 74 ec 9b 20 e0 e3 f9 34 3e 8b 87 61 6b 6d 6c
0d 1b 86 e9 4c 6e b9 b6 e6 53 ce b3 e0 64 6c 51
9a 97 8e 5a f5 38 0d 00 00 01 32 78 dc 43 a0 c8
8c f8 4b c2 87 86 fe d3 4b 34 bf f4 d9 49 83
```

The length of this token is 191 bytes. Below we examine this token field by field.

Token Tag: `60 81 bc 06 09 2a 86 48 86 f7 12 01 02 0f`
This field is contains the following elements in order:
`60`: The tag for ASN.1 SEQUENCE
`81 bc`: The length of subsequent data is 188 (decimal).
`06`: Tag of object identifier (OID)
`09`: The length of the mechanism’s OID is 9 (decimal).
`2a 86 48 86 f7 12 01 02 0f`: The mechanism’s OID.

Token ID: 00
It means the client uses public-key authentication mode.

Initiator Length: `08`
The length of initiator is 8 (decimal) bytes.

Initiator: `63 6c 69 65 6e 74 30 31`
The node name of initiator is “client01”.

Acceptor Length: `0c`
The length of acceptor is 12 (decimal) bytes.

Acceptor: `6d 79 67 73 73 2d 73 65 72 76 65 72`
The node name of acceptor is “mygss-server”.

Password Length: `00 80`
The length of password field is 128 bytes.

Password: `46 0e 2c 93 … 5a f5 38 0d`
This field contains the encrypted 64-byte random data and the one-time password.

Flags: `00 00 01 32`
The context flag is 0x132. It means mutual authentication is required, replay and out-of-sequence detection is enable, and confidentiality and integrity services are allowed.

Signature: `78 dc 43 a0 … f4 d9 49 83`
The 20-byte signature is presented at the end of the message.

2. Response token

The response token generated by gss_accept_sec_context routine is:

```
02 00 f6 87 02 b1 6b e2 ce e2 a6 08 50 43 e5 9b
90 36 d7 dd c8 0b
```

The token length is 22 bytes. I breakdown this token as follows.

Token ID: `02`
It indicates the token is a response token.

Status: `00`
It means context initiator has successfully authenticated to the context acceptor.

Signature: `f6 87 02 b1 … d7 dd c8 0b`
The 20-byte signature is used to authenticate context acceptor.

3. Wrap token

The wrap token generated by the gss_wrap routine is:

```
04 00 00 00 00 00 01 df 71 b6 9b fa 42 b0 c1 3c
b1 4d cc 88 e3 46 cd 7e e3 15 75 e4 39 db 54 4f
ab 18 d5 9e 0f b6 c8 00 b1 3e 0e a1 dc fe cd c2
45 af ce ac 30 57 af e5 d7 24 3c
```

The token length is 59 bytes. Below we examine this token field by field.

Token ID: `04`
It means the token is a wrap token.

Integrity algorithm: 00
The MIC was calculated by using HMAC-SHA-1.

Confidentiality algorithm: 00
The message buffer was encrypted by using AES.

Sequence number: `00 00 00 01`
The first per-message token emitted by the context initiator.

MIC: `df 71 b6 9b … 7e e3 15 75`
This field contains the 20-byte MIC.

Message: `e4 39 db 54 … e5 d7 24 3c`
The field contains the encrypted message buffer.

4. MIC token

The MIC token generated by the gss_get_mic routine is:

```
03 00 00 00 00 01 5d ef d2 de 2b 5b 52 c9 30 9f
bc 05 b9 84 ec a0 b9 de d7 97
```

The token length is 26 bytes. I breakdown this token as follows:

Token ID: `03`
It means the token is a MIC token.

Integrity algorithm: 00
The MIC was calculated by using HMAC-SHA-1.

Sequence number: `00 00 00 01`
The first per-message token emitted by the context acceptor.

MIC: `5d ef d2 de … b9 de d7 97`
This field contains the 20-byte MIC.

5.3 Performance Tests

5.3.1 Objectives

The objectives of performance tests are:

- Measure the performance of security context establishment in two authentication modes.

- Investigate where the time is being spent in the RADIUS authentication, which is included in security context acceptance.

- Measure the performance of per-message routines.

- Compare the performance of my GSS-API implementation and the Kerberos 5 GSS-API implementation.

5.3.2 Test Bed Setup

In order to evaluate the performance of my GSS-API implementation, I constructed experimental environment as illustrated in the following Figure 5-2.

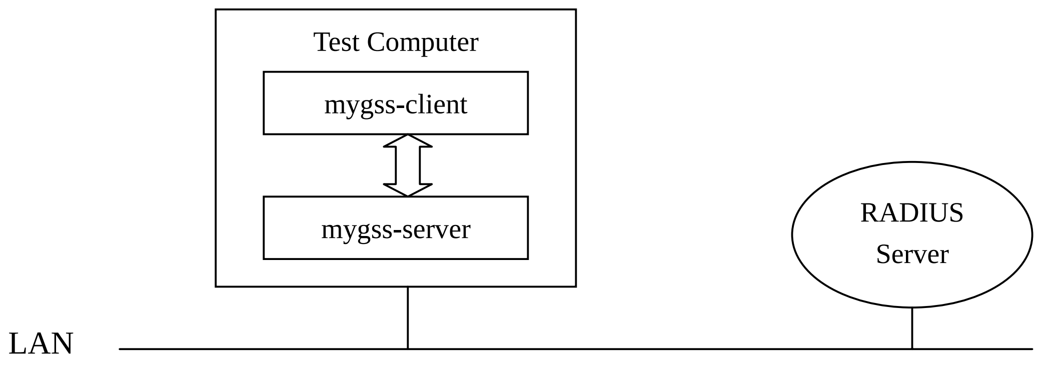

Figure 5-2: Test bed setup

The sample client-server applications were used to perform the tests on one test computer, in order to collect data on same hardware and software environment. I also implemented Kerberos Key Distribution Center (KDC) on the test computer. A RADIUS server running WinRadius [36] was used to process authentication requests. Table 5 shows the hardware and software configurations of computers used in the test bed.

Computer	Hardware	OS	Software
Test Computer	Intel Celeron 2.8GHz 512MB RAM	Red Hat Linux 9 Kernel: 2.4.20-8	mygss-client mygss-server MIT Kerberos 1.4.3
RADIUS Server	AMD Sempron 1.6GHz 512MB RAM	Windows XP	WinRadius 2.01

Table 5-2: Hardware and software configurations

The public/private key pairs required in the tests were generated using the OpenSSL toolkit [30] in version 0.9.7a.

5.3.3 Test Data Collection

In all tests the data was based on the execution time of function calls or code segments of the sample client-server programs. In order to minimize the interference from other processes, all unnecessary services on the test computer are turned off. The function I used for acquiring time is `gettimeofday()`. One of the arguments in this function is a `timeval` structure, as specified in `<sys/time.h>`:

```
struct timeval {
   time_t         tv_sec;     /* seconds */
   suseconds_t    tv_usec;    /* microseconds */
};
```

It gives the number of seconds and microseconds since the Epoch (00:00:00 UTC, January 1, 1970).

The following example shows how I calculate the elapsed time of a `gss_wrap` call:

```
gettimeofday(&start, 0);
maj_stat = gss_wrap(&min_stat,
                     context,
                     conf_state,
                     qop,
                     &in_buf,
                     &state,
                     &out_buf);
gettimeofday(&end, 0);
printf("gss_wrap: %d usecs\n", calc_time(&start, &end));
```

The function `calc_time()` returns the difference (in microseconds) of its two `timeval` structure parameters:

```
int calc_time(struct timeval *start, struct timeval *end)
{
   int usec, sec;

   if (end->tv_usec < start->tv_usec) {
      usec = (1000000 + end->tv_usec) - start->tv_usec;
      end->tv_sec--;
   } else
      usec = end->tv_usec - start->tv_usec;

   sec = end->tv_sec - start->tv_sec;

   return (sec * 1000000 + usec);
}
```

Theoretically, `gettimeofday()` has a maximum resolution of one microsecond. I wrote a simple program to investigate the actual time resolution and average calling overhead of `gettimeofday()` on the test computer. The following code segment shows how time data are collected.

```
struct timeval tv[NUMBER_OF_LOOPS];
for (i = 0; i < NUMBER_OF_LOOPS; i++)
   gettimeofday(&tv[i], NULL);
```

By walking through `timeval` array `tv[]`, maximum time resolution can be determined. It is the smallest non-zero difference between consecutive array elements. The average overhead of calling `gettimeofday()` can be calculated using array `tv[]` by averaging all the non-zero differences. In this experiment, the value of `NUMBER_OF_LOOPS` was 2000. Table 5-3 shows experiment result on the test computer.

`gettimeofday()`	Time (in microseconds)
Maximum time resolution	1
Average calling overhead	0.93

Table 5-3: Experiment result of gettimeofday()

The results reveal that theoretical maximum time resolution of `gettimeofday()` is achieved on test computer. The average calling overhead is less than one microsecond, and is negligible compared to the execution time of GSS-API routines I measured (see the following sections).

5.3.4 Performance of Context Establishment

Two GSS-API routines, gss_init_sec_context and gss_accept_sec_context were measured in different configurations: secret-key mode and public-key mode. In public-key mode, two public-private key pairs were applied, one is a 1024-bit key pair, and the other is a 2048-bit key pair. I repeated the test 10 times in order to calculate the average value and standard deviation. The measurement results are listed in Table 5-4. Standard deviations are presented in parentheses.

GSS-API routines	Average time (in microseconds)		
	Secret-key mode	Public-key mode (1024-bit key)	Public-key mode (2048-bit key)
gss_init_sec_context first call	1148 (232)	3184 (296)	3382 (363)
gss_init_sec_context second call	53 (5)	52 (7)	51 (5)
gss_accept_sec_context	8951 (821)	18083 (1578)	60020 (1894)
In total	10152	21329	63452

Table 5-4: Test results of context establishment

The results indicate that the mechanism has better performance in secret-key mode than public-key mode. The length of RSA key applied in public-key mode significantly affects the performance, mostly of the gss_accept_sec_context routine.

5.3.5 Performance of RADIUS Authentication

The gss_accept_sec_conext routine includes a RADIUS authentication process. The time spent in RADIUS authentication can be roughly broken down into the following steps:

- Step 1: Build and send an Access-Request
- Step 2: Wait for the response
- Step 3: Receive and check the response

As the radiusclient-ng library encapsulates the communication between RADIUS client and server, I modified the source code of the library in order to measure the elapsed time of these steps. The test was repeated 10 times. The results are presented in Table 5-5. Standard deviations are presented in parentheses.

	Average time (in microseconds)
Step 1	2289 (331)
Step 2	833 (15)
Step 3	185 (11)
In total	3307

Table 5-5: RADIUS authentication time

The results indicate that building and sending an Access-Request consumes approximately 69% of the time of RADIUS authentication. It is because in this step the program needs to do the following things:

- Parse the configuration file
- Read shared secret from a file
- Read and update sequence number (Identifier) in a file
- Generate a 16 byte pseudo-random Request Authenticator
- Encrypt user's password
- Construct the whole packet
- Create a socket
- Send the packet

The results also reveal that in my test environment, the network latency and the performance of the RADIUS server have minor impact on the RADIUS authentication time. However, a high-latency connection between the RADIUS client and server can negatively impact authentication times. An extreme scenario is when the RADIUS server is down or inaccessible, this will cause timeouts and retries. The procedure of user authentication and the execution time of gss_accept_sec_context routine will be greatly prolonged. The prolonged time depends on timeout and retry settings specified in the RADIUS client configuration file. With the timeout value of 5 seconds and two retries, I measured the RADIUS authentication time. The test was repeated 10 times. The results are persented in Table 5-7. Standard deviations are presented in parentheses.

	Average time (in microseconds)
Step 1	2302 (395)
Step 2	10000354 (24)
Step 3	N/A
In total	10002656

Table 5-7: RADIUS authentication time (dead server)

The results indicate almost all of the time was spent in waiting for the server's response. This "dead server" problem is caused by the use of RADIUS protocol, as RADIUS operates on UDP, which is a connectionless transport protocol, the client is responsible for providing its own timeouts and retry mechanism.

The use of RADIUS protocol may cause another problem. We consider the following scenario:

- The context acceptor sends an Access-Request packet containing an valid one-time password to the RADIUS server.
- The RADIUS server processes it and sends back an Access-Accept packet.
- The Access-Accept packet is lost due to a network outage.
- The context acceptor retransmits the request.

Thus the RADIUS server will receive the same request twice. However, the password embedded in the second request will be invalid, because the password is already used by the first request. To prevent the problem brought by duplicate requests, a reliable, low-latency link between the context acceptor and the RADIUS server is highly recommended.

5.3.6 Performance of Per-message Routines

My GSS-API implementation employs HMAC-SHA-1 as integrity algorithm. I calculated the execution time of gss_get_mic call (in the server) and gss_verify_mic call (in the client) respectively, I also measured the execution time for calculating MIC. In the test, I had the client and sever applications process a message of 65536 bytes and repeated the test 10 times. The results are shown in Table 5-8. Standard deviations are presented in parentheses.

	gss_get_mic	gss_verify_mic
Calculating MIC (in microsecond)	2609 (278)	2633 (293)
Total time (in microsecond)	2644 (301)	2711 (225)
Throughput (KB/millisecond)	24.02	23.61

Table 5-8: Performance of integrity service (64KB message)

Message confidentiality was provided by using AES (CBC mode). I measured the execution time of gss_wrap call (in the client) and gss_unwrap call (in the server) respectively, I also calculated the time used for calculating MIC and message encryption/decryption in these routines. I had the client and sever applications process a 64KB message and repeated the test 10 times. The test results are shown in Table 5-9. Standard deviations are presented in parentheses.

	gss_wrap	gss_unwrap
Calculating MIC (in microsecond)	2636 (286)	2597 (281)
Encryption/decryption (in microsecond)	3335 (273)	3372 (198)
Total time (in microsecond)	6797 (325)	6889 (337)
Throughput (KB/millisecond)	8.77	8.66

Table 5-9: Performance of integrity and confidentiality service (64KB message)

5.3.7 Comparing with Kerberos

In order to compare the performance of my GSS-API implementation and Kerberos GSS-API implementation, the sample applications needed to work with the Kerberos mechanism.

libgssapi_krb5 is the GSS-API shared library provided by MIT’s Kerberos version 5 distribution [7]. It was very easy to “kerberize” the sample applications. I only needed to slightly modify the Makefile to make both applications link with the Kerberos GSS-API library. There was no need to make any modification to the source code, as that is the purpose of using GSS-API.

I focused on the investigation of 6 key routines in GSS-API as these routines consume most of the time. They are:

- `gss_init_sec_context()`
- `gss_accept_sec_context()`
- `gss_get_mic()`
- `gss_verifiy_mic()`
- `gss_wrap()`
- `gss_unwrap()`

In order to make a relatively fair comparison, I had my GSS-API mechanism use secret-key mode in context establishment as Kerberos is also based on symmetric key cryptography.

In this test, per-message routines were called with default QoP value and a 64KB message. The MIT Kerberos implementation supports various encryption and checksum algorithms. I configured the Kerberos mechanism use AES (128-bit key, CBC mode with cipher text stealing) as its default algorithm for message encryption and HMAC-SHA-1-96 as its default integrity algorithm. Those two algorithms are not exactly the same as the algorithms used in my GSS-API implementation, but are very similar.

The test was repeated 10 times. The results of performance comparisons are presented in Table 5-10. Standard deviations are presented in parentheses.

GSS-API routines	Average time (in microseconds)	
	libmygss.so.0.0	libgssapi_krb.so.2.2
gss_init_sec_context first call	1148 (232)	4843 (394)
gss_init_sec_context second call	53 (5)	1405 (142)
gss_accept_sec_context	8951 (821)	9934 (578)
gss_get_mic	2637 (353)	2586 (262)
gss_verifiy_mic	2755 (287)	2797 (234)
gss_wrap	6822 (303)	6881 (298)
gss_unwrap	6909 (421)	7039 (372)
In total	29275	35485

Table 5-10: Performance comparison between my GSS-API and the Kerberos GSS-API

The results indicate my GSS-API implementation has better performance for context establishment than the MIT Kerberos GSS-API library. The most time-consuming routine of my implementation is gss_accept_sec_context, in which a RADIUS authentication is performed.

5.4 Memory Check

Valgrind [37] is a freely available tool suite for debugging and profiling Linux programs. The Valgrind distribution has multiple tools, the most popular is the memory-checking tool (memcheck) which can detect many common memory errors such as:

- Touching memory you shouldn't (e.g. overrunning heap block boundaries)
- Using values before they have been initialized
- Incorrectly freeing memory, such as double-freeing heap blocks
- Memory leaks

The sample client and server pair along with the GSS-API shared library passed the Valgrind’s memory-checking test.

5.5 Discussions

5.5.1 Key Deployment Schemes

In secret-key authentication mode, shared secrets are deployed in the server (context acceptor) and its client nodes (context initiators). A client can individually authenticate to the server by providing its one-time password, however the server cannot prove its identity using the signature field of the response token, it can only prove that the context acceptor is in possession of the shared secret. Thus every client node that knows the shared secret can impersonate the context acceptor. Unfortunately, this means that there is no real mutual authentication between two peers. In addition, there is a risk that a malicious user who has the shared secret can decrypt other users’ messages by eavesdropping on the network. The more shared secrets are distributed, the more insecure the system is. Therefore the secret-key mode has a poor scalability.

In public-key authentication mode, there is only one single copy of the “secret”, i.e. the private key, which is kept by the context acceptor. No one other than the context acceptor has the right private key, hence they can not generate a proper response token to authenticate to the context initiator. Thus, mutual authentication is easily achieved. Moreover, due to the characteristic of public-key cryptography, the public key can be freely distributed to anyone without fear of causing a security problem, hence the public-key authentication mode has better scalability than secret-key mode.

Compare to public-key mode, the advantage of secret-key mode is that it is fast, as secret-key cryptography is substantially faster than public-key cryptography [28].

5.5.2 Identity Privacy

The identities of two communicating peers are sent in plain text within the initial context token, in addition, the User-Name attribute in a RADIUS Access-Request packet is also presented in cleartext. This may lead to problems in some environments since an eavesdropper is able to identify the communicating parties. This mechanism is not suitable for the scenario that has a requirement of user identity confidentiality.

5.5.3 Perfect Forward Secrecy

The session key is generated based on the information (one-time password and/or random data) transmitted between GSS-API peers. This design does not provide Perfect Forward Secrecy (PFS). That is, if the shared secret (in secret-key mode) or the RSA private key (in public-key mode) is somehow compromised, an attacker can decrypt old conversations (assuming that the attacker knows the algorithm to calculate a session key).

5.5.4 Security of RADIUS Authentication

The security of RADIUS authentication is a part of the security of the whole mechanism. The RADIUS protocol has a set of vulnerabilities that may lead to various attacks on user passwords or shared secret [38]. Since we use one-time passwords, the attacks on user password are negligible. The attacks on the shared secret are based on exhaustive search, the use of strong shared secrets is important to resistance to brute force attack.

6. Conclusions

In this thesis I investigated the use of one-time password and RADIUS authentication as a protection facility for a GSS-API mechanism. I designed, implemented, and evaluated the security mechanism of using a one-time password to establish a GSS-API security context between two communicating peers.

A one-time password based GSS-API mechanism was developed, it was provided as a shared library in a Unix-like environment. For conformance approval, the implementation passed SAP's GSS-API test program. Two sample application programs using this GSS-API shared library were created and demonstrated that the security mechanism works properly.

With the help of these sample applications, I measured the performance difference between secret-key mode and public-key mode in security context establishment, and performance of per-message routines. I measured the performance of RADIUS authentication. I compared the performance of my GSS-API implementation and the Kerberos 5 GSS-API implementation.

The results of performance tests reveal that gss_accept_sec_context is the most time-consuming routine called in the phase of context establishment. In public-key mode, the key length greatly affects the performance of context acceptance. In secret-key mode, while the RADIUS processing is time consuming, the overall process is still faster with one-time passwords and RADIUS than Kerberos.

Using one-time passwords can provide better security to user authentication by eliminating the risk presented by reused passwords. The one-time password based GSS-API mechanism developed and evaluated in this thesis can significantly facilitate the transition to stronger authentication from static-password based systems.

7. Future Work

With the emergence of the Diameter protocol, adding support for Diameter is essential future work. Diameter operates over a reliable connection-oriented transport protocols (TCP or SCTP), it can solve the “dead server” problem. The impact on performance of user authentication when using Diameter should be investigated.

The current implementation uses SHA-1 based HMAC to generate MICs. Since attacks have been found for the full SHA-1 function in last year, the use of a stronger secure hash algorithm, such as SHA256, should be examined for enhancing communication security.

The GSS-API library used a fixed IV to perform AES CBC encryption, the same plaintext will result in the same ciphertext in a communication session. Hence it is desirable to prepend 16 bytes of random information (confounder) at the start of the plaintext message before encryption.

In order to prevent the acceptance of the same message sent back in the reverse direction by an adversary, it is desirable to add a “direction indicator” in per-message tokens. The direction indicator indicates if a token was emitted by a context initiator or a context acceptor.

For the purpose of providing Perfect Forward Secrecy (PFS), the use of Diffie-Hellman key exchange in the security context establishment should be investigated.

References

[1] Neuman, C., Yu, T., Hartman, S. and K. Raeburn, “The Kerberos Network Authentication Service (V5)”, RFC 4120, July 2005.

[2] Adams, C., “The Simple Public-Key GSS-API Mechanism (SPKM)”, RFC 2025, October 1996.

[3] Linn, J., “Generic Security Service Application Program Interface Version 2, Update 1”, RFC 2743, January 2000.

[4] Wray, J., “Generic Security Service API Version 2 : C-bindings”, RFC 2744, January 2000.

[5] Kabat, J. and M. Upadhyay, “Generic Security Service API Version 2 : Java Bindings” RFC 2853, June 2000.

[6] Zhu, L., Jaganathan, K. and S. Hartman, “The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2”, RFC 4121, July 2005.

[7] MIT Kerberos, “Kerberos: The Network Authentication Protocol”, 10 August 2005, <http://web.mit.edu/kerberos/www/> (25 October 2005).

[8] Heimdal, <http://www.pdc.kth.se/heimdal/> (25 October 2005).

[9] Sun Microsystems, “Sun Enterprise Authentication Mechanism”, <http://www.sun.com/software/solaris/8/ds/ds-seam/ds-seam.pdf> (25 October 2005).

[10] Microsoft Corporation, “Windows 2000 Kerberos Authentication”, <http://www.microsoft.com/windows2000/techinfo/howitworks/security/kerberos.asp> (25 October 2005)

[11] SECUDE IT Security, <http://www.secude.com/> (25 October 2005).

[12] Rigney, C., Willens, S., Rubens, A. and W. Simpson, “Remote Authentication Dial-In User Service (RADIUS)”, RFC 2865, June 2000.

[13] Rigney, C., Rubens, A., Simpson W. and S. Willens, “Remote Authentication Dial-In User Service (RADIUS)”, RFC 2058, Januaray 1997.

[14] Postel, J., “User Datagram Protocol”, RFC 768, August 1980.

[15] Calhoun, P., Loughney, J., Guttman, E., Zorn, G. and J. Arkko, "Diameter Base Protocol", RFC 3588, September 2003.

[16] Haller, N., "The S/KEY One-Time Password System", RFC 1760, February 1995.

[17] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and O. Ranen, "HOTP: An HMAC-based One Time Password Algorithm", RFC 4226, December 2005.

[18] Eastlake 3rd, D., and P. Jones, "U.S. Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001

[19] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[20] U.S. National Institute of Standards and Technology, "NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1", 25 August, 2004 <http://csrc.nist.gov/hash_standards_comments.pdf> (25 October 2005)

[21] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997

[22] U.S. National Institute of Standards and Technology, "Data Encryption Standard (DES)", FPIS 46-3, 25 October 1999, <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf> (25 October 2005)

[23] Electronic Frontier Foundation, "EFF: DES Cracker Project" <http://www.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/> (25 October 2005)

[24] Electronic Frontier Foundation, "Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design" July 1998.

[25] U.S. National Institute of Standards and Technology, "AES Home Page", <http://csrc.nist.gov/CryptoToolkit/aes/index1.html> (25 October 2005)

[26] Rivest, R., Shamir, A. and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", <http://theory.lcs.mit.edu/~rivest/rsapaper.pdf> (28 Febrary, 2006)

[27] International Telecommunication Union, "Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T

Recommendation X.690, July 2002

[28] Menezes, A., van Oorschot, P., and S. Vanstone, “Handbook of Applied Cryptography”, CRC Press, October 1996

[29] Linn, J. “Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedure”, RFC 1421, Febrary 1993

[30] OpenSSL, “OpenSSL: The Open Source toolkit for SSL/TLS”, <http://www.openssl.org/> (28 Febrary, 2006)

[31] Gladman, B., “Implementations of AES (Rijndael) in C/C++ and Assembler” <http://fp.gladman.plus.com/cryptography_technology/rijndael/> (18 January 2006)

[32] Mavroyanopoulos, N. and S. Schumann, “Mhash library”, <http://mhash.sourceforge.net/> (15 December 2005)

[33] BerliOS Developer, “RADIUS Client Library”, <http://developer.berlios.de/projects/radiusclient-ng/> (15 December 2005)

[34] Free Software Foundation, Inc., “GNU Make”, <http://www.gnu.org/software/make/> (6 January, 2006)

[35] SAP AG, <http://www.sap.com>, (8 January, 2006)

[36] WinRadius, <http://www.itconsult2000.com/en/product/WinRadius.html> (28 Febrary, 2006)

[37] Valgrind Developers, “Valgrind Home”, <http://valgrind.org/> (8 January 2006)

[38] Hill, J., “An Analysis of the RADIUS Authentication Protocol” <http://www.untruth.org/~josh/security/radius/radius-auth.html> (12 March, 2006)

Appendix 1

```
**********************************************************************
*                                                                    *
*   GSStest Result Summary :                                         *
*                                                                    *
**********************************************************************

  GSSTEST Release   :  Version 1.26    03-Sep-2002  (32-bit)
         built on   :  Jan 19 2006 at 17:37:14

  Command line:
    argv[ 0] = "./gsstest"
    argv[ 1] = "-l"
    argv[ 2] = "libmygss.so.0.0"
    argv[ 3] = "-a"
    argv[ 4] = "mygss-server"

  GSS-API Library   :  libmygss.so.0.0

NOTICE: This GSS-API mechanism REQUIRES an external SNC-Adapter!

======================================================================
  Current Date&Time :  Tue, 14-Mar-2006   23:28:50   GMT -01:00
  Operating System  :  Linux
          -Release  :  2.4.20-8
  Hardware/Machine  :  i686
  scalar C-types    :  void* ptrdiff_t size_t time_t long int wchar_t char
    (sizes in bits) :   32      32s      32u    32s   32s 32s   32s    8u
  Perf-Index (p-90) :  dbg= 7.80   (opt= 4.10)
  Timer Resolution  :  0.001 millisec using "gettimeofday()"
  Hostname          :  example.com
  Current user      :  xi
======================================================================

(( 1 a ))
  Mechanism = {1 2 840 113554 1 2 15}
            = { 9, "\052\206\110\206\367\022\001\002\017" }

(( 1 b ))
  Nametype  = {1 2 840 113554 1 2 2 15}
            = { 10, "\052\206\110\206\367\022\001\002\002\017" }
```

```
(( 1 c ))
  default    Initiator name: "client01"
  given      Acceptor  name: "mygss-server"
  canonical  Acceptor  name: "mygss-server"

  (default)  Acceptor  name:  not available -- don't worry

----------

(( 2 a ))
Performance of names management calls            min      avg      max :
  gss_import_name()           ( 1126 calls)     0.00     0.01     0.04  ms
  gss_display_name()          ( 1268 calls)     0.00     0.00     0.01  ms
  gss_export_name()           (  970 calls)     0.00     0.01     0.03  ms
  gss_canonicalize_name()     ( 1279 calls)     0.00     0.00     0.01  ms
  gss_compare_name()          (  851 calls)     0.00     0.00     0.01  ms
  gss_release_name()          ( 3950 calls)     0.00     0.00     0.01  ms

(( 2 b ))
Observed sizes of names:
  printable names                 [  8 ..  15 ]  bytes
  exported binary canonical names  [ 27 ..  31 ]  bytes

Support of Hostbased Service Names:
  gss_inquire_names_for_mech() includes GSS_C_NT_HOSTBASED_SERVICE,n  and our
sample hostbased service name is accepted.

----------

(( 3 a ))
Performance of credential management calls       min      avg      max :
  gss_acquire_cred() Ini      (  128 calls)     0.00     0.01     0.34  ms
  gss_acquire_cred() Acc      (  109 calls)     0.03     0.05     0.12  ms
  gss_inquire_cred() Ini      (   17 calls)     0.00     0.01     0.03  ms
  gss_inquire_cred() Acc      (    1 calls)     0.00     0.00     0.00  ms
  gss_release_cred()          (  237 calls)     0.00     0.00     0.04  ms

(( 3 b ))
Observed Credentials lifetime(s):
                                Elapsed real time : 00h 00m 09s
  Initiator credentials lifetime constant at      : Indefinite
  Acceptor  credentials lifetime constant at      : Indefinite
```

```
----------
Security context establishment  (52 contexts)


(( 4 a ))
Mechanism uses 2-way authentication

(( 4 b ))
Security Context Attribute results:
     requested  = (MUTUAL,REPLAY,CONF,INTEG,TRANS)
     provided   = (MUTUAL,REPLAY,CONF,INTEG,TRANS)

Performance of context establishment calls        min      avg      max :
  gss_init_sec_context()   #1  (   52 calls)     0.32     0.36     0.99  ms
  gss_init_sec_context()   #2  (   52 calls)     0.04     0.04     0.04  ms
  gss_accept_sec_context() #1  (   52 calls)     5.76    43.18   117.34  ms

(( 4 c ))
Total context establishment overhead              min      avg      max :
  gss_init_sec_context()       (   52 calls)     0.36     0.40     1.03  ms
  gss_accept_sec_context()     (   52 calls)     5.76    43.18   117.34  ms
  gss_delete_sec_context()     (  104 calls)     0.00     0.01     0.01  ms
  gss_inquire_context()        (  738 calls)     0.00     0.01     3.52  ms


(( 4 d ))
Observed initial lifetimes for established security contexts:

                             Elapsed real time : 00h 00m 09s
  Initiator context lifetimes constant at      : 01h 00m 00s
  Acceptor  context lifetimes constant at      : 01h 00m 00s

(( 4 e ))
Observed token sizes for    gss_init_sec_context():  [  78 ..  78 ]  bytes
Observed token sizes for  gss_accept_sec_context():  [  22 ..  22 ]  bytes


----------
Security context transfer: 234 context transfers, 34 cross-process

(( 5 a ))
Performance of security context transfer          min      avg      max :
  gss_export_sec_context() Ini (  135 calls)     0.01     0.01     0.29  ms
  gss_export_sec_context() Acc (  168 calls)     0.00     0.01     0.03  ms
  gss_import_sec_context() Ini (  135 calls)     0.00      0.01      0.02  ms
  gss_import_sec_context() Acc (  168 calls)     0.00      0.01      0.03  ms


(( 5 b ))
Interprocess token sizes for Initiator:  [ 300 .. 300 ]  bytes  ( 135 calls)
Interprocess token sizes for  Acceptor:  [ 300 .. 300 ]  bytes  ( 168 calls)



----------
Message Protection Services:
(( 6 a ))
Confidentiality and Integrity

(( 6 b ))
Performance of per-message calls                  min       avg       max :
  gss_context_time()          ( 2049 calls)     0.00      0.01      0.22  ms
  gss_wrap_size_limit()       ( 7654 calls)     0.00      0.00      3.46  ms

(( 6 c ))
GSS-API message protection throughput   avg     max :
  gss_getmic()          ( 1048 calls)  16.1     27.8   KByte/msec  min= 0.02 ms
  gss_verifymic()       ( 1048 calls)  15.5     27.6   KByte/msec  min= 0.02 ms
  gss_wrap(mic)         (  883 calls)  13.0     23.7   KByte/msec  min= 0.03 ms
  gss_unwrap(mic)       (  883 calls)  13.6     22.2   KByte/msec  min= 0.02 ms
  gss_wrap(conf)        ( 1148 calls)   7.4     12.3   KByte/msec  min= 0.04 ms
  gss_unwrap(conf)      ( 1148 calls)   6.6     11.0   KByte/msec  min= 0.04 ms


(( 6 d ))
Token sizes for           gss_getmic()  :  [ 26 .. 26 ]  bytes
Message size increase for gss_wrap(mic) :  [ 31 .. 31 ]  bytes
Message size increase for gss_wrap(conf):  [ 31 .. 46 ]  bytes

----------

(( 7 a ))
Performance of remaining GSS-API calls            min       avg       max :
  gss_display_status()        (  220 calls)     0.00      0.01      0.02  ms
  gss_release_buffer()        (10617 calls)     0.00      0.00      0.17  ms
  gss_release_oid_set()       (    9 calls)     0.00      0.00      0.00  ms
  gss_indicate_mechs()        (    2 calls)     0.01      0.03      0.05  ms

----------
```

```
Displaying of status information via gss_display_status():

(( 7 b ))
  gss_display_status(major) output message size [  8 ..109 ] avg= 40 chars

-------------------------------------------------------------
  Current limits for SNC-interoperability
    with R/3 Releases 3.1x, 4.0x, 4.5x:

  max. length of printable names           =     90   octets
  max. length of exported (binary) names   =    126   octets
  max. size of MIC token by gss_getmic()   =    128   octets
  max. message size increase by gss_wrap() =    128   octets
  max. size of context establishment token =  25000   octets
  max. size of exported context token      =   8000   octets
  max. context establishment time          =   1000.0 millisec
  max. CPU time import/export_sec_context  =      3.0 millisec  ( 0.7)
  max. CPU time for gss_context_time()     =      1.0 millisec  ( 0.2)
  max. getmic()/verifymic() latency        =      9.0 millisec  ( 2.1)
  max. wrap(mic)/unwrap(mic) latency       =      9.5 millisec  ( 2.2)
  max. wrap(conf)/unwrap(conf) latency     =     10.0 millisec  ( 2.4)
-------------------------------------------------------------


==================

Passing all API result tests.
Passing all SAP constraints.

** No problems detected, BUT
** this gssapi mechanism REQUIRES the use of an external SNC-Adapter

  Mechanism  = {1 2 840 113554 1 2 15}
  Nametype   = {1 2 840 113554 1 2 2 15}

  Max. data protection level =   3 (Privacy Protection)

  Hardware Platform          =   Linux 2.4.20-8


==================
Done.
```

Appendix 2

1024-bit RSA key pair used in tests

```
-----BEGIN RSA PUBLIC KEY-----
MIGHAoGBAJpDqOZQqUr+Cf/Wd318MEB0G5XLszvZDj+OW6E2hV3txSa3NF7E7hMj
9Ohw583TyHFZft58lenb93jzWBqPJlufdmfvM6JqStAcopbHlPF3SbkYbNkkKn1n
E/T3Cz8J3MQjgzMw2aavvvyWYtR98GQf074399+56QvDgSqJp2uxAgED
-----END RSA PUBLIC KEY-----

-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQCaQ6jmUKlK/gn/1nd9fDBAdBuVy7M72Q4/jluhNoVd7cUmtzRe
xO4TI/TocOfN08hxWX7efJXp2/d481gajyZbn3Zn7zOiakrQHKKWx5Txd0m5GGzZ
JCp9ZxP09ws/CdzEI4MzMNmmr778lmLUffBkH9O+N/ffuekLw4EqiadrsQIBAwKB
gGbXxe7gcNypW//kT6j9dYBNZ7kyd307XtUJkmt5rj6eg28kzZSDSWIX+Jr1796N
MEuQ/z79uUaSpPtM5WcKGZFhVu/Kxq3pp6e5QHIB7yfnEBvCVucbszi6i3Mc34QV
tnCtVvPrHrjUWiHilfdthKTykS/t7mheUY1M5/V/E55DAkEAx3494J2aiwQDHxCC
ibvs2zgqBeflOcKmin/Df5GY4uWwf829S6OFFe2BrdF89A8SHdnc8RY4PG5n5D+q
AmaS7wJBAMX1wl5sAwDLUWexaTrySMF5ZW8CmUDbAcTDBMwqYAZlap+zCASIDFtK
SBSwZGWaWpYgGWH8CvAAT+vlkEija18CQQCE/tPrE7xcrVdqCwGxJ/M80BwD7+4m
gcRcVSz/tmXsmSBVM9OHwli5SQEei6iitLa+kT32DtAoSZqYKnFW7wyfAkEAg/ks
PvKsqzI2RSDw0fbbK6ZDn1cQ1edWgyyt3XGVWZjxv8ywAwVdkjGFYyBC7mbnDsAQ
6/1coAA1R+5gMGzyPwJABptNTC2ek/gATdHkicZFx5bj9MBDx2LzXKlwzqx3w2I2
uEVAmWLZ11twaTgAZlrgTdz5RDJ6kUp51GcrpHAwJA==
-----END RSA PRIVATE KEY-----
```

Appendix 3

Output of sample programs (public-key mode, 1024-bit key)

mygss-client:

```
ONE-TIME-PASSWORD: 990526
Sending initial context token (size=191)...
60 81 bc 06 09 2a 86 48 86 f7 12 01 02 0f 00 08
63 6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d 73
65 72 76 65 72 00 80 46 0e 2c 93 4e a1 88 7a 5f
e5 a9 f1 a8 f8 3a a0 12 41 2c a6 86 52 c3 92 4b
9a e6 62 40 30 db 65 c5 f9 cf a3 56 a3 44 0f 5d
35 70 50 fe 75 60 cb 4e 7a 40 2b 39 7e e7 c4 d2
59 c8 99 52 92 8d 42 d1 71 82 37 eb a1 82 c7 8b
b0 4d d1 9b 77 c9 71 2c b6 22 c4 e8 dd f9 bf b7
70 74 ec 9b 20 e0 e3 f9 34 3e 8b 87 61 6b 6d 6c
0d 1b 86 e9 4c 6e b9 b6 e6 53 ce b3 e0 64 6c 51
9a 97 8e 5a f5 38 0d 00 00 01 32 78 dc 43 a0 c8
8c f8 4b c2 87 86 fe d3 4b 34 bf f4 d9 49 83
continue needed...

Received response token (size=22)
02 00 f6 87 02 b1 6b e2 ce e2 a6 08 50 43 e5 9b
90 36 d7 dd c8 0b

"client01" to "mygss-server", context established

Sending wrap token (size=59)...
04 00 00 00 00 00 01 df 71 b6 9b fa 42 b0 c1 3c
b1 4d cc 88 e3 46 cd 7e e3 15 75 e4 39 db 54 4f
ab 18 d5 9e 0f b6 c8 00 b1 3e 0e a1 dc fe cd c2
45 af ce ac 30 57 af e5 d7 24 3c

Received MIC token (size=26)...
03 00 00 00 00 01 5d ef d2 de 2b 5b 52 c9 30 9f
bc 05 b9 84 ec a0 b9 de d7 97

MIC token verified.
```

mygss-server:

```
Created a socket.
Got connection
run_server...(PID = 2572)

Received initial context token (size=191):
60 81 bc 06 09 2a 86 48 86 f7 12 01 02 0f 00 08
63 6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d 73
65 72 76 65 72 00 80 46 0e 2c 93 4e a1 88 7a 5f
e5 a9 f1 a8 f8 3a a0 12 41 2c a6 86 52 c3 92 4b
9a e6 62 40 30 db 65 c5 f9 cf a3 56 a3 44 0f 5d
35 70 50 fe 75 60 cb 4e 7a 40 2b 39 7e e7 c4 d2
59 c8 99 52 92 8d 42 d1 71 82 37 eb a1 82 c7 8b
b0 4d d1 9b 77 c9 71 2c b6 22 c4 e8 dd f9 bf b7
70 74 ec 9b 20 e0 e3 f9 34 3e 8b 87 61 6b 6d 6c
0d 1b 86 e9 4c 6e b9 b6 e6 53 ce b3 e0 64 6c 51
9a 97 8e 5a f5 38 0d 00 00 01 32 78 dc 43 a0 c8
8c f8 4b c2 87 86 fe d3 4b 34 bf f4 d9 49 83

Sending response token (size=22):
02 00 f6 87 02 b1 6b e2 ce e2 a6 08 50 43 e5 9b
90 36 d7 dd c8 0b

Accepted connection: "client01"

Received wrap token (size=59):
04 00 00 00 00 00 01 df 71 b6 9b fa 42 b0 c1 3c
b1 4d cc 88 e3 46 cd 7e e3 15 75 e4 39 db 54 4f
ab 18 d5 9e 0f b6 c8 00 b1 3e 0e a1 dc fe cd c2
45 af ce ac 30 57 af e5 d7 24 3c

Received message: 54 68 69 73 20 69 73 20 61 20 74 65 73 74

Sending MIC token (size=26):
03 00 00 00 00 01 5d ef d2 de 2b 5b 52 c9 30 9f
bc 05 b9 84 ec a0 b9 de d7 97

run_server...DONE! (PID = 2572)
```

Appendix 4

Output of sample programs (public-key mode, 2048-bit key)

mygss-client:

```
ONE-TIME-PASSWORD: 112312
Sending initial context token (size=320)...
60 82 01 3c 06 09 2a 86 48 86 f7 12 01 02 0f 00
08 63 6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d
73 65 72 76 65 72 01 00 34 27 53 e5 d7 c8 0b 47
a0 7a 62 e6 de b7 db 4c dd 0a 02 97 ec 24 6d bb
5b d7 9c ce f7 3b 85 d1 3d af c8 cc 9a ed b3 9f
dc dd 2c 9f b6 92 42 3c 9a b8 54 3d a5 9c 01 5a
0d da c2 50 b7 f2 02 f4 9d 7a 08 1c 6c a0 05 c0
ef d2 3a 95 26 a5 b1 e3 51 2e ed e0 78 00 e9 db
d7 1d 58 66 d6 33 5c ea c3 78 87 b6 d5 4c 90 22
f6 5b e8 84 92 3e 76 f7 e1 73 62 0a 95 ff 02 f7
e5 25 8e 96 e0 0a ca de 87 a8 84 22 63 09 f8 cc
ff ac d6 11 85 37 8e 23 af 24 02 41 58 3d f9 9f
e0 3a 25 e4 c1 a9 ab 73 d2 8d 48 72 44 21 24 ee
b9 8d 0c 51 c8 27 8b 41 43 e6 d4 a5 f1 e3 9e 39
55 42 f7 c3 a0 3c 5f 4b f4 7a 67 72 91 89 b8 40
4e 46 9a fd a1 a3 48 0b e1 f4 3e be a8 10 e5 a8
8a 3c 9c c0 11 e4 9e 74 07 fa 31 b4 f8 9e 76 e8
de 87 f0 87 9e f6 f2 93 2b 1d 22 1e d0 ba 94 4c
a3 8b ad 33 92 1e 52 9e 00 00 01 32 8d 6b 88 5b
fa e7 b8 da 78 c8 ab 81 db 9a e6 e8 d9 d5 08 86

continue needed...

Received response token (size=22)
02 00 c7 fc e4 ef d3 2e a9 ca 8f 9d 13 b6 11 0b
31 04 89 5c df 0d

"client01" to "mygss-server", context established

Sending wrap token (size=59)...
04 00 00 00 00 00 01 f1 59 d2 23 14 01 c6 ec 63
55 aa 3a c0 74 7d db eb b8 e4 3b 48 a0 0d 03 84
86 e5 6e 4f 70 f5 39 52 55 b8 df b7 05 50 49 c1
50 df 51 4a 2e e7 89 47 f1 1f 93
```

```
Received MIC token (size=26)...
03 00 00 00 00 01 59 05 30 97 2a 66 71 1a ab 47
43 de 46 1e 44 9a 9d db 01 6b

MIC token verified.
```

mygss-server:

```
Created a socket.
Got connection
run_server...(PID = 2613)

Received initial context token (size=320):
60 82 01 3c 06 09 2a 86 48 86 f7 12 01 02 0f 00
08 63 6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d
73 65 72 76 65 72 01 00 34 27 53 e5 d7 c8 0b 47
a0 7a 62 e6 de b7 db 4c dd 0a 02 97 ec 24 6d bb
5b d7 9c ce f7 3b 85 d1 3d af c8 cc 9a ed b3 9f
dc dd 2c 9f b6 92 42 3c 9a b8 54 3d a5 9c 01 5a
0d da c2 50 b7 f2 02 f4 9d 7a 08 1c 6c a0 05 c0
ef d2 3a 95 26 a5 b1 e3 51 2e ed e0 78 00 e9 db
d7 1d 58 66 d6 33 5c ea c3 78 87 b6 d5 4c 90 22
f6 5b e8 84 92 3e 76 f7 e1 73 62 0a 95 ff 02 f7
e5 25 8e 96 e0 0a ca de 87 a8 84 22 63 09 f8 cc
ff ac d6 11 85 37 8e 23 af 24 02 41 58 3d f9 9f
e0 3a 25 e4 c1 a9 ab 73 d2 8d 48 72 44 21 24 ee
b9 8d 0c 51 c8 27 8b 41 43 e6 d4 a5 f1 e3 9e 39
55 42 f7 c3 a0 3c 5f 4b f4 7a 67 72 91 89 b8 40
4e 46 9a fd a1 a3 48 0b e1 f4 3e be a8 10 e5 a8
8a 3c 9c c0 11 e4 9e 74 07 fa 31 b4 f8 9e 76 e8
de 87 f0 87 9e f6 f2 93 2b 1d 22 1e d0 ba 94 4c
a3 8b ad 33 92 1e 52 9e 00 00 01 32 8d 6b 88 5b
fa e7 b8 da 78 c8 ab 81 db 9a e6 e8 d9 d5 08 86

Sending response token (size=22):
02 00 c7 fc e4 ef d3 2e a9 ca 8f 9d 13 b6 11 0b
31 04 89 5c df 0d

Accepted connection: "client01"

Received wrap token (size=59):
04 00 00 00 00 00 01 f1 59 d2 23 14 01 c6 ec 63
```

```
55 aa 3a c0 74 7d db eb b8 e4 3b 48 a0 0d 03 84
86 e5 6e 4f 70 f5 39 52 55 b8 df b7 05 50 49 c1
50 df 51 4a 2e e7 89 47 f1 1f 93

Received message: 54 68 69 73 20 69 73 20 61 20 74 65 73 74

Sending MIC token (size=26):
03 00 00 00 00 01 59 05 30 97 2a 66 71 1a ab 47
43 de 46 1e 44 9a 9d db 01 6b

run_server...DONE! (PID = 2613)
```

Appendix 5

Output of sample programs (secret-key mode)

mygss-client:

```
ONE-TIME-PASSWORD: 188452
Sending initial context token (size=78)...
60 4c 06 09 2a 86 48 86 f7 12 01 02 0f 01 08 63
6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d 73 65
72 76 65 72 00 10 76 0b c9 5c c9 ac ad d6 66 9b
cf cf ab 3b 16 2f 00 00 01 32 99 56 5c 0c 9a 67
6e 68 15 a6 46 a5 d0 cc 2b 61 ac 2c 60 74
continue needed...

Received response token (size=22)
02 00 47 1d f5 3d f0 83 02 c6 85 b6 78 a8 d8 bf
1b 7f a9 97 02 f7

"client01" to "mygss-server", context established

Sending wrap token (size=59)...
04 00 00 00 00 00 01 5a 3a fe 26 7e 55 05 56 37
48 d6 d4 47 fd 4a 0c 73 63 e8 e4 56 20 16 f5 79
0e d9 b6 63 f9 e6 b1 e2 84 7c 81 14 0d 72 ae 5f
ae 91 42 75 03 45 3c 42 41 18 9e

Received MIC token (size=26)...
03 00 00 00 00 01 4c cf 13 d9 5d 5b 93 6b 8e 1f
7c b5 a4 19 05 d8 ba 6c dc b2

MIC token verified.
```

mygss-server:

```
Created a socket.
Got connection
run_server...(PID = 2638)

Received initial context token (size=78):
60 4c 06 09 2a 86 48 86 f7 12 01 02 0f 01 08 63
6c 69 65 6e 74 30 31 0c 6d 79 67 73 73 2d 73 65
72 76 65 72 00 10 76 0b c9 5c c9 ac ad d6 66 9b
```

```
cf cf ab 3b 16 2f 00 00 01 32 99 56 5c 0c 9a 67
6e 68 15 a6 46 a5 d0 cc 2b 61 ac 2c 60 74

Sending response token (size=22):
02 00 47 1d f5 3d f0 83 02 c6 85 b6 78 a8 d8 bf
1b 7f a9 97 02 f7

Accepted connection: "client01"

Received wrap token (size=59):
04 00 00 00 00 00 01 5a 3a fe 26 7e 55 05 56 37
48 d6 d4 47 fd 4a 0c 73 63 e8 e4 56 20 16 f5 79
0e d9 b6 63 f9 e6 b1 e2 84 7c 81 14 0d 72 ae 5f
ae 91 42 75 03 45 3c 42 41 18 9e

Received message: 54 68 69 73 20 69 73 20 61 20 74 65 73 74

Sending MIC token (size=26):
03 00 00 00 00 01 4c cf 13 d9 5d 5b 93 6b 8e 1f
7c b5 a4 19 05 d8 ba 6c dc b2

run_server...DONE! (PID = 2638)
```

Appendix 6

List of Acronyms

AAA	Authentication, Authorization and Accounting
AES	Advanced Encryption Standard
ASN.1	Abstract Syntax Notation One
DER	Distinguished Encoding Rule
DES	Data Encryption Standard
CBC	Cipher Block Chaining
ECB	Electronic Codebook
GSS-API	Generic Security Service Application Programming Interface
HMAC	keyed-Hash Message Authentication Code
IV	Initialization Vector
KB	Kilobytes
MIC	Message Integrity Code
OID	Object Identifier
OTP	One-Time Password
PEM	Privacy Enhanced Mail
PFS	Perfect Forward Secrecy
PIN	Personal Identification Number
PKI	Public Key Infrastructure
QoP	Quality of Protection
RADIUS	Remote Authentication Dial In User Service
SCTP	Stream Control Transmission Protocol
SHA	Secure Hash Algorithm
SPKM	Simple Public-Key GSS-API Mechanism
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

CoS/CCS 2006-09

www.kth.se