Using SESAME's GSS-API to add Security to Unix Applications

Paul Ashley
Mark Rutherford

Information Security Research Center
School of Data Communications
Queensland University of Technology
GPO Box 2434, Brisbane - AUSTRALIA
Tel. +61 7 3864 1940
ashley@fit.qut.edu.au

Mark Vandenwauver
Sebastien Boving

Katholieke Universiteit Leuven
Dept. Elektrotechniek, ESAT-COSIC
Kardinaal Mercierlaan 94
B-3001 Heverlee - BELGIUM
Tel. +32 16 321134
mark.vandenwauver@esat.kuleuven.ac.be

Abstract

SESAME is a security architecture that starts from the Kerberos protocol and adds to it public-key based authentication, role based access control, delegation of rights and an extensive auditing facility. SESAME provides the GSS-API for securing applications and this paper describes our efforts in securing some of the most important Unix applications using SESAME: telnet, the BSD rtools and the remote procedure call. We have found the benefit of using SESAME is that the applications are secured in a uniform manner, additional security services are provided to the applications that are unavailable with other architectures, and the impact of SESAME on the application's performance is not excessive.

Keywords: network security, access control, Kerberos, SESAME, GSS-API.

1. Introduction

The need for security in networked applications has resulted in a variety of solutions. Telnet [4] a common application that allows a remote login across a network, traditionally had only username and password for authentication (that are passed in the clear over the network) and no data protection. Telnet has previously been secured with the Secure Sockets Layer (SSL) [6] and Kerberos [9], to improve this situation.

The common BSD rtools [8] traditionally had host-based authentication only, and the security was considered so weak that the rtools were banned from most installations. To improve the situation the rtools have been secured with Kerberos, SSL, and a replacement version called Secure Shell Protocol (SSH) [16] is available.

The remote procedure call, and in particular the version called Open Network Computing (ONC) [12] that is available on a number of platforms, traditionally only provided authentication services. Attempts have been made to add data protection services to ONC RPC [7, 5] and the version of RPC from OSF's DCE also provides an access control service.

The same situation occurs for a number of other common networked applications: they lack security and a variety of techniques have been used to secure the applications. We have chosen to secure the three applications described above with SESAME. There are a number of reasons to use SESAME:

- SESAME provides an excellent range of security services, more than any other architecture used to date for securing the applications, and we were interested in how we might use these additional security services.
- To investigate using the SESAME GSS-API [3] to secure different applications: whether or not it is useful using a uniform method, and what is the impact on performance.
- To encourage the use of SESAME by freely providing the source code for the *sesamized* applications to the general public.

The paper is organized as follows. Section 2 and 3 introduce the SESAME architecture and the GSS-API. Section 4 describes our efforts in securing the applications and difficulties encountered. Section 5 describes the performance of each of the applications. We finish with our conclusions.

2. SESAME

The SESAME architecture is the result of an EC research project that was instigated by Bull, ICL and Siemens. This means that the main European computer manufacturers were a driving force behind the project. When the project finished in 1995, the work was continued at ESAT/COSIC and QUT.

The concrete objectives of the SESAME project were to define and implement protocols both for authentication and for the management of access control in a client-server scenario. SESAME provides the following advantages:

- Single or mutual authentication using either Kerberos or public-key based authentication [15]
- Confidentiality and Integrity of the data during transit
- Access control based on an Role Based Access Control (RBAC) scheme [14]
- Delegation of rights
- Auditing service
- Multi-domain support [2]

In 1995, a first public release of the SESAME code (version 4.0) was issued at [13]. Unfortunately the regulations in Europe did not allow for a distribution of SESAME containing symmetric key cryptography. Therefore, according to the guidelines of the SOG-IS (Senior Officials Group on Information Security), the DES was removed and replaced by a straightforward XOR. It is thus essential for the users of SESAME to plug in their own version of the DES. To make it easy for non-programmers, the version for Linux [1] that resides in Australia, now contains an excellent implementation of the DES [17].

3. GSS-API

The goal of the Generic Security Services Application Program Interface (GSS-API) [11] is to provide the application programmer with a uniform library of functions that implement security in a client-server scenario. The big advantage to the programmer is that he does not need to know or understand the details of how every security function (e.g. user authentication or data integrity) is actually implemented. The programmer only needs a basic understanding of security in order to make the appropriate calls in his programs. The GSS-API is a standard, so it should not make a difference who has made the GSS-API implementation.

Several network security architectures provide an implementation of the GSS-API. However they do not all offer the same functionality. The Kerberos GSS-API [10] for example gives the application programmer a set of security services based on the underlying Kerberos architecture. In comparison to the SESAME GSS-API, it is limited because it only offers mutual authentication, data confidentiality and data integrity. The SESAME GSS-API [3] provides the extra services outlined in Section 2. It also offers these services across a large multi-domain platform.

Basically, a SESAME GSS-API secured client-server application works in the following way. Both the client and server will acquire their credentials, performing an authentication (`gss_acquire_credential`) to the security server. The client will then present its context (containing the keying information and privilege attribute certificate) (`gss_init_sec_context`) to the server. The server can then take an access control decision based on the information that is provided to it, and in the case of success will establish a security context (`gss_accept_sec_context`). In the case that the client has asked for a mutual authentication, the server will send back a token, that will be verified by the client before he establishes his side of the security context (the client loops `gss_init_sec_context` until he receives the server's token). From that moment on, the client and server application can use the per message security services provided by the GSS-API such as encryption (`gss_seal`, `gss_unseal`) and digital signature (`gss_sign`, `gss_verify`). When the connection is broken off, the security context will be deleted (`gss_delete_sec_context`).

4. Applications

SESAME is available for a range of Unix platforms including Linux [1], and the applications described in this section were developed for Linux but should port to the other Unix platforms.

4.1. telnet

4.1.1. General

Telnet is a client-server based program, that allows users to work on remote computers. The main goal of telnet is to make an environment-independent representation of the transmitted data. It was meant to be used both for terminal to mainframe connections and for terminal to terminal connections and process to process communications. Nowadays, it is mainly used to get a remote shell across the network.

4.1.2. Authentication

A stronger authentication mechanism needed to be implemented. The current telnet implementations are based on passwords, or on the caller's (pretended) IP address. The SESAME architecture provides a way to achieve strong authentication using its GSS-API implementation.

The latest BSD UNIX telnet source code, released in October 1995, has also a built-in facility to negotiate this authentication [4]. This facility is a framework for any security system to perform the authentication, but it does not actually implement any authentication mechanism. It does follow the basic GSS-API client-server setup.

During the implementation process we encountered several problems:

1. *Buffer round-trip*:

One of the first problems we encountered was the fact that the token sent by the client to the server would not be received in full. Looking at the transmitted tokens, we noticed that telnet sent about 1500 bytes, but telnetd only processed about 500 of them before stating the suboption was not properly terminated and processing the next option. The problem was that when telnetd receives a suboption, it places the suboption's data in a static circular buffer (reading or writing past the end wraps around to the beginning) of 512 bytes long. As the security token sent with this suboption is much longer, it wrapped a couple of times around this buffer.

2. *Environment variables*:

At several points in SESAME's code, SESAME needs to get information from the environment of the user that starts up the application. Since telnetd clears its environment variables during the start-up procedure, they are of course not available. Our solution consists of setting these with the `setenv()` function before actually calling the SESAME routines.

3. *Autologin*:

When SESAME authentication finally succeeds, this result has to be passed to the rest of telnetd's code so it knows that no password-based authentication is needed. In order to do this, the `autologin` variable should be set to `AUTH_VALID`, as is explained in the telnet documentation. Only when it is set to this value, telnetd will try to launch `/bin/login` in a 'special' way and will not prompt the user for a password.

The first thing to do was to set `autologin` to `AUTH_VALID` once successful SESAME authentication had been accomplished (it was set to `AUTH_USER`). But telnetd would not try to launch `/bin/login` in this special way. The problem was that somewhere, in a quite uncommon place, this value was always reset to `AUTH_USER` if it had the `AUTH_VALID` value. Deleting this line fixed this problem.

Even then, autologin was still not successful. To login without supplying a password, telnetd had to provide `/bin/login` the user's name. This seemed to be done by the `name` variable. Debugging information showed that this variable was always empty. The pointer to this `name` buffer is passed-by-value several times to subfunctions (so the buffer itself is passed-by-reference), but we could not locate the place where the string got filled with the user name available in `auth_name()` (and apparently it did not get filled at runtime either). So we decided to put the name in a global variable (`savename`) and copy it in telnetd's `name` buffer when leaving `auth_wait()`.

4. *Stop default behavior*:

The need for strong authentication is now hard-coded in *sesamized* telnet, unlike normal telnet which will fall back to password-based authentication after an unsuccessful strong authentication.

4.1.3. Encryption

To ensure data confidentiality, users should be given the possibility to encrypt their telnet session, both from the client to server and server to client. This has been realized using SESAME's GSS-API.

Telnet has also included facilities to add encryption. Unfortunately, the `ENCRYPTION` option Internet Draft never made it to an RFC status, and is now waiting for telnet's author David Borman to continue his work. Due to U.S. export restrictions of software using encryption, the encryption code has also been removed from the source code that is available outside the U.S.

As explained in the `AUTH_ENCRYPT` Internet-Draft, encryption should start as soon as successful authentication has taken place.

Some additional care needs to be taken. Telnet puts data from the network in a buffer (circular buffers are used in the client). These buffers can be flushed anytime: if the data makes part of an option negotiation, it can either be immediately processed (putting the client or server in another 'state'), or, if it is (part of) a suboption, it is pushed in the suboption buffer until the end of suboption is received. Data meant for the (pseudo)terminal can also be treated immediately. So except for suboption negotiation, telnet is never waiting for a complete token to be received.

If we want to encrypt n bytes from a buffer before sending it, what we will finally have to do, is send a token of length $N > n$. At the other side of the communication, it will only be possible to unwrap this token when all N bytes are received. This creates two problems:

1. The receiving side will have to wait until N bytes are received. As it does not know N beforehand, this length has to be sent before sending the token released by `gss_wrap`. Another possibility (e.g. used in the telnet suboptions) would be the use of an escape character.

2. The encryption will imply a network traffic overhead. This might not seem important but it is. In most cases, for client to server traffic n will be much smaller than N. Even worse in the usual case (character-by-character mode) characters are sent over the network one by one (when the client host is slow and the user is typing extremely fast it can happen that more than one character is sent at once). While this is a bad characteristic of telnet (one character typed is one full IP packet on the network), using the SESAME GSS-

API would make it even worse: a bunch of miscellaneous information will be added to the one encrypted character.

To solve all previously mentioned problems, it was decided to add two new GSS-API calls:

- `gss_wrap_char()`
- `gss_unwrap_char()`

These are implemented using DES in Cipher FeedBack mode (CFB). The encryption of one character will rely on the preceding characters. The implementation will also result in an output token of exactly the same length as the input token, so that simple calls to these GSS-API routines can be made without needing to add another buffer to telnet, and the network overhead due to encryption will be nonexistent.

4.2. rtools

4.2.1. General

The rtools are a suite of remote host-to-host communication programs. One of the major features of the tools, is the ability to access resources (files, programs) on the remote host without explicitly logging into the remote host. We have focussed on securing three of the important rtools: rlogin (remote login), rsh (remote shell) and rcp (remote file copy). The version of code used was the Linux NetKit-0.09 package.

4.2.2. Authentication

A stronger authentication mechanism needed to be implemented. The current rtools implementation is based on trusted hosts and is very weak. SESAME provides the services necessary for strong authentication of both client and server.

1. *Environment Variables*:

All of the rtools clear the environment variables during the start-up procedure. Similarly to telnet we saved the environment variables and used `setenv()` function before actually calling the SESAME routines.

2. *Common Library*:

Since there was such a large amount of code that would be common to both servers, and also common to all clients, it made sense to develop a library of functions that could be used by all rtool client and server applications.

3. *Server Authenticating a Client*:

A single line change to the server programs was required for client authentication, with all work done in the library. After the SESAME authentication had been performed, the normal server processing was resumed, with the exception that the host and paranoia checking were deleted. The original code checked that the client connection originated from a reserved port on the client machine. This assumption is at best weak, and at worst dangerous, since getting root access on one machine could compromise the security of many machines, and many accounts. The paranoia code is no longer needed as SESAME guarantees that a PAC that is accepted is the verfied user. For each server, the calls to accept the SESAME connection were identical, and consisted of calls `rauth_init()`, then `rauth_accept_client` which returned the `RauthHeader`.

4. *Client Authenticating a Server*:

After making the initial TCP connection to the server, and obtaining `RauthHandle` from `rauth_init` the client calls `rauth_client_login` which takes care of obtaining a PAC from the SESAME server, transferring it to the application server and receiving the server's PAC for mutual authentication. After this authentication takes place, normal client processing is resumed.

4.2.3. Encryption

The code has been written to ensure data protection for all data sent between client to server, and server to client.

Securing the streams was less straightforward than authentication. Normally the rtools put the network stream in non-blocking mode, and expects the network as a stream, reading a variable number of bytes, and writing a variable number of bytes. The problem was to convert the `gss_wrap` and `gss_unwrap` procedures which worked on blocks of data into a stream.

Some form of buffering was needed for the block to stream conversion, as the blocks sent by the application could be of arbitrary size. The alternative was to encrypt a byte at a time but this was considered too inefficient especially for file copies.

In the applications, putting the network stream into non-blocking mode was turned off. If it was left on, the reading of a block of encrypted data, would have to poll or wait for the end of the encrypted block, or to keep retrying to send all of the encryption blocks before returning. Making this change does not seem to have altered the operation of the programs, and is the approach taken by the Kerberos rtools.

4.3. RPC

4.3.1. General

The RPC is a system that allows an application client program to execute a procedure on a remote networked server. It has two main functions: it is the underlying transport technology for applications such as NFS and it can be used as a programming tool to allow quicker development of network applications.

4.3.2. Authentication

We chose to secure the ONC RPC source code from Sun Microsystems that has built-in facilities for different authentication flavors: none, Unix (based on simple UID and GID), DES (DES encryption with client and server keys) and Kerberos, and used the SESAME GSS-API to provide SESAME flavor authentication.

ONC RPC has been designed to be very modular and we found this design simplified our task immensely. It allows developers to add new authentication flavors with minimal change to existing code, and with the addition of new functions. We added a new flavor called RPCSEC_GSS, by adding routines and modifying only small amounts of existing code.

The main difficulty was to add an authentication flavor that required multiple phases for authentication. The existing flavors relied on a single phase of authentication, whereas GSS-API implementations may require a number of phases. This was achieved by the client keeping track of the phase during authentication.

The ONC RPC authentication has been written so that authentication occurs once where a context (a secure session) is established. The client performs a call to `clnt_rpcsec_gss_create` to authenticate and establish context (similar to the existing system where the client would call a specific flavor routine such as `clnt_authnone_create` or `clnt_authunix_create`).

After the session is established no further authentication is required. The SESAME ONC RPC gives the user the choice of both single and mutual authentication. In the normal ONC RPC way, when the client wants to finish a session it calls `auth_rpcsec_gss_destroy`.

4.3.3. Encryption

We have used the GSS-API `gss_wrap` and `gss_unwrap` routines to secure the data for transit. Currently we encrypt the client arguments and server results before they are passed. We give the user the option of no wrapping, integrity only, or both integrity and encryption protection for the data.

ONC RPC uses the External Data Representation (XDR) to allow heterogeneous systems to communicate. All data is passed through an XDR encode routine before transit and XDR decode routine after transit. We chose to place the `gss_wrap` and `gss_unwrap` routines in the main XDR routines, and used them if the flavor was RPCSEC_GSS.

4.3.4. Access Control

Similarly to OSF DCE, we used SESAME to pass a PAC from client to server. The server has two choices when using the PAC: the server can rely on SESAME to examine the PAC and determine if access should be granted, or the server itself can examine the PAC and make a finer access control decision. Our version of SESAME ONC RPC supports both methods, with the advantage that comes with RBAC. SESAME also supports delegation of privileges and this is potentially very useful for RPC.

5. Performance

We are endeavouring to encourage the use of SESAME and this has resulted in two main criteria for the design of our *sesamized applications*: ease of use, and minimal impact on performance. This section outlines the performance impact of SESAME, the results being for a single computer (no network latency) using Pentium 120 MHz, 16 MB RAM (T1), and Pentium 200MHz MMX, 64 MB RAM (T2) both running Linux Redhat 4.2.

5.1. telnet

Telnet traditionally used username and password for authentication and provided no data protection. Table 1 shows the results for *sesamized* telnet, note that there was no observable difference in performance between telnet and *sesamized* telnet.

Security Service	T1	T2
Single Authentication	430 ms	215 ms
Mutual Authentication	445 ms	240 ms
`gss_wrap` character	24.0 μs	17.8 μs
`gss_unwrap` character	7.20 μs	5.60 μs

Table 1. Performance Results for SESAME Telnet

5.2. rtools

The rtools traditionally used host based authentication and provided no data protection. Table 2 shows the results for *sesamized* rlogin, rsh and rcp (for rsh and rcp the file size was 100K). There was no observable difference between rtools and *sesamized* rtools.

5.3. RPC

The design of SESAME ONC RPC has the following philosophy: on the first RPC call single or mutual authentication occurs and a session is established. After the first

Program	Security Service	T1	T2
rlogin	Single Authentication	390 ms	190 ms
	Mutual Authentication	410 ms	210 ms
	`gss_wrap` character	42.0 μs	32.1 μs
	`gss_unwrap` character	13.0 μs	10.1 μs
rsh	Regular 'rsh cat file'	7.98 s	5.11 s
	SESAME 'rsh cat file'	14.9 s	6.41 s
rcp	Regular 'rcp file'	3.43 s	3.29 s
	SESAME 'rcp file'	10.4 s	5.41 s

Table 2. Performance Results for SESAME rtools

call only data protection services are provided (authentication is implicit through the use of the session keys). Note that the timing results are for a full procedure call, so this would involve two calls to `gss_wrap` and `gss_unwrap` for each procedure call using SESAME. The timing results are shown in Table 3.

Flavor	Security Service	T1	T2
Unix	Single Authentication	0.44 ms	0.32 ms
	Procedure Call	0.60 ms	0.35 ms
SESAME	Single Authentication	395 ms	210 ms
	Mutual Authentication	420 ms	230 ms
	Call	0.66 ms	0.42 ms
	Call (Int.)	3.4 ms	2.4 ms
	Call (Int.+Conf.)	3.6 ms	2.5 ms

Table 3. Performance Results for SESAME ONC RPC

6. Conclusion

SESAME provides to applications additional security services that are unavailable with other current technologies: Kerberos and SSL provide authentication and data protection, SSH provides authentication, data protection, data compression and limited access control, and OSF's DCE uses Kerberos and adds access control. SESAME on the other hand provides single or mutual authentication using either Kerberos or public-key based authentication, data protection, access control based on RBAC, delegation of rights, and an auditing service. Using a single architecture to secure all of the applications also provides uniformity and interoperability between the applications.

References

[1] P. Ashley. ISRC SESAME Application Development Pages, http:// www.fit.qut.edu.au/˜ashley/sesame.html.

[2] P. Ashley. Authorization For a Large Heterogeneous Multi-Domain System. In *Proceedings of the AUUG National Conference*, Brisbane, Qld., September 1997.

[3] E. Baize, S. Farrell, and T. Parker. The SESAME GSS-API Mechanism, 1996. Internet Draft IETF Common Authentication Technology WG, November 1996.

[4] D. Borman. Telnet Authentication Protocol, 1993. RFC1416.

[5] M. Eisler, R. Schemers, and R. Srinivasan. Security Mechanism Independence in ONC RPC. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA., July 1996.

[6] A.O. Freier, P. Karlton, and P.C. Kocher. The SSL Protocol Version 3.0, March 1996. Internet Draft IETF Transport Layer Security WG.

[7] B. Jaspan. GSS-API Security For ONC RPC. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 144–151, San Diego, CA., February 1995.

[8] B. Kantor. BSD rlogin, September 1991. RFC1258.

[9] J. Kohl and C. Neuman. The Kerberos Network Authentication Service V5, 1993. RFC1510.

[10] J. Linn. The Kerberos Version 5 GSS-API Mechanism, 1996. RFC1964.

[11] J. Linn. Generic Security Service Application Program Interface Version 2, 1997. RFC2078.

[12] R. Srinivasan. Remote Procedure Call Protocol Specification Version 2, 1995. RFC1831.

[13] M. Vandenwauver. The SESAME home page, http://www.esat.kuleuven.ac.be/cosic/sesame.

[14] M. Vandenwauver, R. Govaerts, and J. Vandewalle. How Role Based Access Cis implemented in SESAME. In *Proceedings of the 6-th Workshops on Enabling Technologies Infrastructure for Collaborative Enterprises*, pages 293–298. IEEE Computer Society, 1997.

[15] M. Vandenwauver, R. Govaerts, and J. Vandewalle. Security of Client-Server Systems. In J. Eloff and R. von Solms, editors, *Information Security*, pages 39–54, 1997.

[16] T. Ylonen, T. Kivinen, and M. Saarinen. SSH Protocol Architecture, November 1997. Internet Draft, IETF Secure Shell Working Group.

[17] E. Young. Libdes, Version 4.01, 1997. available at ftp://ftp.psy.uq.oz.au /pub/Crypto/DES/libdes-x-xx.tar.gz.