REX: A Flexible Remote Execution Service

by

Eric Peterson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2001

© Massachusetts Institute of Technology 2001. All rights reserved.
Abstract

REX is a secure remote execution service that does not perform internal key management. Instead, REX servers are named by self-certifying hostnames, a combination of a DNS name and a one-way hash of a public key. By exposing the mapping between public keys and file servers, REX can function with any key management scheme. REX also supports fast session creation through persistent authentication; a modular, extensible architecture based on inter-process file descriptor passing; and a secure mechanism for obtaining file servers public keys over an untrusted network using the SRP protocol.
1 Introduction

This thesis presents REX, a secure global remote execution service. A remote execution service allows users to run interactive terminal sessions on a remote machine over a network.

The growth of the Internet has provided an economical means for connecting remote machines. Remote execution services such as SSH[8] have become popular since they allow the large group of Internet connected users to work from home, administer machines remotely, connect with remote offices, and transfer files. Remote execution services for use on the Internet must also provide security guarantees to protect against monitoring and modification of transmitted data by an adversary.

Secure remote execution services such as SSH[8] provide security guarantees through public key cryptography for user and server authentication and symmetric key cryptography to protect transmitted data. However, computer systems are only as secure as the method used to obtain public keys. In SSH, key distribution is completely decentralized, permitting any client to securely connect to any server reachable through a network as long as it has the servers public key. While this decentralized architecture is one of the primary reasons for the success of SSH, the protocol does not provide a secure means for distributing public keys.

SSH key management functions well for individuals who want to connect among a small group of machines, but functions poorly in settings in which a high degree of centralization is required, such as a large number of identical, centrally administered workstations. To deal with this situation, SSH can be configured to instead use the more centralized Kerberos authentication library. However, Kerberos does not scale well beyond administrative realms and does not permit normal users to modify the centralized authentication database (for example if a user wants to create a new host or user principal).

Another option would be to enable both Kerberos and public key authentication with
SSH, thereby getting the benefits of both systems. However, the login session are then no longer independent of the authentication method used. For example, if Kerberos is used to authenticate the SSH connection, then the user will also be able to access other services that use the Kerberos library such as AFS filesystems. However, if a user connects to the same account using SSH public key authentication, they will have to invoke a Kerberos utility and type a password to access the same services. This type of SSH configuration does not provide a transparent, distributed remote execution service.

1.1 REX features

REX takes an entirely different approach to flexible key management. Unlike previous remote execution services, REX does not perform key management. New REX servers, like new web servers, need only obtain an Internet address or domain name to be securely accessible from every client in the world. Any client can obtain a secure connection to any REX server over an untrusted network by merely specifying its corresponding self-certifying hostname. A self-certifying hostname specifies both the network name and the public key of a server. REX’s host namespace was adapted from SFS’s self-certifying file namespace[4]. Therefore, the self-certifying hostname inherently specifies enough information to obtain a secure connection with a server. As a result, REX requires no additional information beyond a self-certifying hostname to establish a secure connection with a server. Pushing key management out of REX has powerful consequences; it lets arbitrary key management policies coexist, which in turn makes REX useful in the broadest range of applications.

Public key cryptography enables REX to provide mutual authentication. First, by knowing the server’s public key through the self-certifying hostname, a REX client can determine that it is communicating with the intended server. Second, users have public
keys known to REX servers. A REX user runs an authentication agent process that holds his or her private key; through these agents, REX clients can prove the identities of users to servers without disclosing any passwords. Authentication requests intended for a user's agent can be forwarded securely over REX connections, freeing users from having to start additional agents on remote machines. Typically, a user will start an agent and install their private key only once, when they first login to a local machine.

Since REX uses the same authentication agent (called sfsagent[3]) as SFS, REX public key identities can also be used with the SFS global, secure filesystem. sfsagents are accessed through a secure IPC interface called suidconnect, which allows other programs to easily access a user's agent. Other services can be easily adapted to authenticate using sfsagent identities thanks to the agent's general interface. The sfsagent facilitates transparent, distributed computing by only requiring a user to manage a single public key identity.

REX provides a convenient, secure method for obtaining the public keys of servers using only a password. REX provides this by means of the SRP protocol[7], which lets a client and server negotiate a secret session key starting only from a poorly-chosen password. REX uses this key to establish a secure channel over which the server can transmit its self-certifying hostname and optionally a user's private key. Therefore, a user need only remember their SRP password to securely obtain both their server's public key and their own private key over an untrusted network.

Existing secure remote execution services such as SSH[8] perform user and server authentication every time a session is created. Consequently, the computationally expensive public key operations that must be performed lead to high session creation latency. Inefficient session creation not only slows down interactive terminal logins but also seriously affects the performance of programs like CVS[1] that tunnel through SSH by creating a new
session for each invocation of the command.

REX supports fast session creation by only requiring public key cryptography on a client's first session with a server. During this initial session, a master session key is established with the server. All subsequent sessions are protected by symmetric cryptographic keys generated efficiently from the master session key. Persistent authentication allows REX to perform comparably to unencrypted remote execution services without sacrificing security.

Unlike previous remote execution services, functionality in REX is partitioned into separate binary modules that communicate through Unix domain sockets. Unix domain sockets, beside providing efficient inter-process communication, also allow open file descriptors to be passed between processes. The ability to pass file descriptors provides a powerful interface for designing client-server applications. In REX, functionality related to creating connections and opening files is placed in binary modules and accessed through a file-descriptor-passing interface. For example, TCP port forwarding in REX is implemented using `connect` and `listen` modules. The `listen` module listens for TCP connections and will write accepted sockets to its standard output. When a `connect` module reads a socket on its standard input, it will connect the other end of the socket to the local TCP port.

The modular architecture of REX provides ease of extensibility and generality. The REX client is run-time configured to execute pairs of modules, one of which runs on the remote side of the REX session. The REX protocol merely shuttles I/O traffic between the file descriptors of every local module and those of its corresponding remote module. The purpose of the REX protocol is to transparently connect local modules with remote modules. Therefore, from a REX module's perspective, it is connected directly with the remote module through a set of Unix domain sockets. Depending on the consistent and
well-defined Unix domain socket interface instead of the underlying REX protocol simplifies REX module development. Moreover, functionality can be added to REX through binary modules without having to recompile other parts of REX or modify protocols that would render other clients or servers incompatible.

The modular architecture of REX also provides a convenient means for isolating code that must execute as superuser. By minimizing functionality that runs in privileged mode, REX limits the consequences of security holes caused by programming errors such as improper buffer management. Placing privileged code in concise and well-defined modules also simplifies security audits of REX source code.

The modular design of REX necessitates a secure means for establishing connections to persistent modules that run in the background (for example connecting to one’s agent). REX accomplishes this by calling the *suidconnect* program. *suidconnect* not only allows clients to connect to modules by name but also presents the module with the caller’s UNIX credentials (user and group identifiers) so that access control can be performed.

## 2 Related Work

REX uses SFS protocols for obtaining secure connections to servers and authenticating users. As a result, REX inherits a number of important features from SFS including external key management, the certification program interface, and self-certifying hostnames[4]. In addition, REX user authentication requests are fulfilled by *sfsagents*, which store private keys for users. The *sfsagent* was also extended to store information related to persistent authentication of servers. The SFS protocol also allows REX clients to obtain user private keys and server public keys securely using only a poorly-chosen password through the use of the SRP protocol.
REX's decentralized architecture and use of public key authentication is similar to SSH[8]. Unlike SSH, REX only performs computationally expensive public key authentication on the first connect with a particular server. SSH also does not provide a secure mechanism for obtaining the public keys of servers remotely. Consequently, SSH is sometimes vulnerable to “man in the middle” attacks between the client and the server. SSH also does not provide a secure mechanism for obtaining the private keys of users remotely. Users who are unable to obtain their own private key can only authenticate themselves using a password, making “man in the middle” attacks even more disastrous. SSLtelnet[2] also establishes secure connections using server public keys. However, it only supports password-based user authentication and is vulnerable to “man in the middle” attacks, a dangerous combination.

Traditional remote execution programs such as RSH and telnet have been modified to use the Kerberos Network Authentication Protocol[6]. However, Kerberos does not scale well beyond administrative realms due to it’s centralized nature. In addition, Kerberos has a number of security flaws including vulnerability to offline dictionary attacks and dependence on clock synchronization.

3 Design

This section describes the design of REX by breaking it into four parts: the global server namespace; session key cryptography; the REX network protocol; and the REX binary module architecture.
HostID (specifies public key)

new - york.1cs.mit.edu: 85xq6pznt4mgfvj4mb23x6b8adak55ue

Location

Figure 1: A self-certifying hostname

3.1 Server namespace and server authentication

REX uses public-key cryptography to guarantee authentic and secure connections with servers. However, the client must get the public key of the server from somewhere. By exposing the binding between server network addresses and server public keys, users are free to choose their own key management schemes. Specifically, REX hosts are identified by names of the form Location: HostID called self-certifying hostnames. The Location tells the REX client how to connect to the remote host while the HostID tells the client how to certify a secure channel to that server. HostID is a cryptographic hash of that key and the server’s Location. HostIDs let clients ask servers for their public keys and verify the authenticity of the reply. Figure 1 is an example of an actual self-certifying hostname.

Knowing a server’s public key suffices to communicate securely with it. However, requiring users to explicitly specify a self-certifying pathname for every REX session would be inconvenient. To cope with this usability issue, SFS allows users to register certification programs with their agent, which map strings to self-certifying hostnames[4]. When the REX client encounters a hostname which is not self-certified, it will attempt to resolve it by calling the certification programs that the user has registered with their SFS user agent.

An example of a useful certification program is dirsearch, which will search a list of directories for a file with the name to be resolved and return its contents as the self-certifying hostname. For example, the following command would configure the agent to search for
certified hostnames in the directory /u/ericp/.sfs/hosts:

sfskey certprog dirsearch -c /u/ericp/.sfs/hosts

REX also lets users retrieve self-certifying hostnames securely from remote servers using only a password. REX accomplishes this with the SRP protocol, which lets a client and server negotiate a secret session key starting only from a poorly-chosen password[7]. REX uses this key to establish a secure channel over which the server can transmit its self-certifying hostname. Users can even retrieve their own private keys over this secure link.

3.2 Session key cryptography

The first time a user connects to a REX server, public key cryptography is performed to authenticate both the user and the server and to establish a secure connection. A master session key is established over this secure connection. However, subsequent connections can be established using only symmetric key cryptography. If REX were to use a single key to protect all subsequent connects, it would be possible for an attacker to replay REX sessions. REX instead uses the hash of the master session key and a sequence number to create a separate session key for every subsequent connection.

Actual session keys, which protect traffic through symmetric key cryptography, are generated from a hash of the master session key and a sequence number. Since it is feasible for a user to have multiple master session keys established with the same server\(^1\), a means for identifying master sessions is required. The SessionID generated from a sequence value of zero is used to identify the master session. A SessionID can be safely sent over insecure links since session keys cannot be derived from it.

\(^1\)This can happen when a user connects to a server twice using different agent instances.
\[ \text{SessionKey} = \text{SHA-1}(\text{MasterSessionKey}, \text{SeqNo}) \]

\[ \text{SessionID} = \text{SHA-1}(\text{SessionKey}) \]

\[ \text{MasterSessionID} = \text{SessionID}(\text{SeqNo} = 0) \]

The REX server only creates new sessions with sequence numbers that have not been previously used. This prevents an active network attacker from replaying old sessions. The use of the SHA-1 cryptographic one-way hash to generate session keys guarantees that they cannot be derived from each other. Therefore, if a session key is disclosed, the secrecy of other sessions will not be affected.

The SFS user agent was extended to store master session keys and create and pass off session keys to REX clients. Every time a REX session is established, the REX client first requests a MasterSessionID, session key, and sequence number from the user’s agent. If the agent does not have a MasterSession key established with that REX server, it authenticates the server and user through public key cryptography\(^2\) and then establishes a master session key with a newly spawned process called proxy on the server. Subsequent connects are made directly to the proxy server without public key cryptography. proxy implements the REX network protocol, which will be discussed in the next section.

### 3.3 REX network protocol

Clients and servers in REX communicate through an RPC network protocol that provides a transparent interface to remote UNIX domain sockets. UNIX domain sockets differ from

\(^2\)REX uses the SFS protocol for user and server authentication
other stream interfaces by allowing open file descriptors to be passed over them in addition to data.

REX provides the abstraction of bidirectional data streams between client and server modules. These streams are not sockets that directly connect client to server modules. Rather, REX uses stub sockets on both ends. In between the client and server, there is a single, secure RPC channel over which I/O traffic between the stub descriptors is relayed. For example, if data is read on the client side descriptor it is then encoded, sent over the RPC channel, decoded on the server, and finally written to the server side descriptor. However, these intermediate operations are completely transparent to the connected modules. Figure 2 shows both the transparent interface to the file descriptors of remote modules and the stub descriptors in a typical REX session.

Since the REX network protocol can provide access to multiple remote binary modules over a single RPC connection, an additional level of multiplexing is needed. REX handles this by organizing file descriptors into channels. Each remote module is associated with a channel, which is the set of file descriptors connected to or passed from that module.

### 3.3.1 RPC interface

REX clients spawn remote processes by creating channels through the `mkchannel` procedure. The `mkchannel` procedure takes as arguments the path and the argument vector of the remote program to spawn and returns the channel number of the newly created process. Numbering channels allows clients to spawn multiple processes concurrently within a single REX session. Each channel is associated with a set of descriptors that is initially the stdin, stdout, and stderr descriptors of the remote process.

The `data` and `newfd` procedures handle writing respectively data and file descriptors to
the descriptors of the remote process. Since sockets are bidirectional, both the REX client
and server export the data and newfd procedures. The data procedure takes as arguments
the channel number, descriptor number, and data to write to the remote file descriptor.

The newfd procedure takes as arguments the channel number, the remote descriptor
number over which a file descriptor should be passed, and the newly allocated descriptor
number which will be passed over the remote file descriptor. The REX protocol layer
implements sending file descriptors over the network by creating a socketpair on the remote
side, and passing one end of the socket pair to the remote process. The REX network
protocol then relays writes between the file descriptor received on the client side and the
socketpair on the server side. From the perspective of binary modules that are connected
by the REX network protocol, file descriptors can be passed over the network.

### 3.4 REX modular architecture

The ability to pass open file descriptors between processes provides a powerful interface for
designing client-server applications. For example, a client process can call upon a server
process to create a network connection or allocate a pseudo terminal; the server process
can then pass the open file descriptor back to the client process so that it can used with
all of the generic I/O routines. All of the details related to creating the file descriptor is

<table>
<thead>
<tr>
<th>RPC:</th>
<th>MKCHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARGUMENT:</td>
<td>path, argument vector</td>
</tr>
<tr>
<td>RESULT:</td>
<td>channel</td>
</tr>
<tr>
<td>RPC:</td>
<td>DATA</td>
</tr>
<tr>
<td>ARGUMENT:</td>
<td>channel, file descriptor, data</td>
</tr>
<tr>
<td>RESULT:</td>
<td>void</td>
</tr>
<tr>
<td>RPC:</td>
<td>NEWFD</td>
</tr>
<tr>
<td>ARGUMENT:</td>
<td>channel, file descriptor, new file descriptor</td>
</tr>
<tr>
<td>RESULT:</td>
<td>void</td>
</tr>
</tbody>
</table>

Table 1: REX network protocol RPC interface
conveniently hidden within the server process.

Since the REX protocol allows file descriptors to be passed over the network, client processes can call server processes on remote machines. REX aggressively exploits this feature by moving as much functionality as possible into binary modules on the server. The REX server daemons, *rexd* and *proxy*, only implement user authentication and the REX network protocol. All other functionality including TCP and UNIX port forwarding, pseudo-terminal allocation, and X-Windows display forwarding is implemented in binary modules. REX also allows binary modules on the client side of the connection. For example, a client could connect a remote TCP *listen* module to a local TCP *connect* module to implement port forwarding. When the remote module receives connections it will send the connected socket to the client module which will then connect to a local port.

Figure 2 shows the binary modules connected in an interactive login session with X forwarding. When the *tcpconnect* module receives a file descriptor on the UNIX domain socket labeled 1, it will connect to port 6000 and relay I/O between the newly created connection and the file descriptor it was passed. In this example, there is currently one X connection being forwarded, labeled 2. On the server side, the *tcplisten* module listens for X connections and writes accepted file descriptors to the UNIX domain socket labeled 1. The terminal file descriptors labeled 4 are the client and server side of the interactive login session. The *tty* and *ttyd* modules handle forwarding of terminal device behavior that can't be expressed on a UNIX domain socket. For example, when a client resizes the xterm, the application on the slave side of the pseudo-terminal (/dev/tty) must receive a SIGWINCH signal and be able to read the new size through an *ioctl*. Upon window resizing, the *tty* module catches the SIGWINCH signal, reads the new terminal dimensions through an *ioctl*, and sends the new size to the remote side over the descriptor labeled 3. Upon receiving the window resize
Figure 2: REX module example implementing login session and X forwarding. The gray lines are not actual network connections. They represent the transparent interface to the UNIX domain sockets of remote modules provided by the REX network layer.

message, the \textit{ttyd} module will update the server side pty through an \textit{ioctl}.

REX’s modular design has a number of important advantages. Moving most functionality into binary modules makes the REX network protocol generic and flexible. Clients can select any binary module at run-time and access it remotely through the REX network protocol. Therefore, extending REX to support new functionality does not require any changes to the REX network protocol. REX’s modular design also simplifies the implementation of binary modules since they only depend on the UNIX domain socket abstraction implemented by the REX network protocol. Binary modules also provide a convenient means for isolating code that must run in privileged mode. In REX, privileged functionality is minimized and placed into binary modules. Unprivileged REX code communicates with these binary modules through a secure IPC interface called \textit{suidconnect} that is based on UNIX domain sockets bound to protected pathnames in the local file-system. Using sep-
arate binary modules and file-system protection mechanisms is more powerful and robust than the traditional “dropping privileges within a monolithic binary” approach.

4 REX example

This section shows an example of how we use REX within the MIT Laboratory for Computer Science environment. Self-certifying pathnames to certified servers are maintained centrally on our file server, new-york.lcs.mit.edu. The file server also exports these self-certifying pathnames using the SFS filesystem protocol. Every workstation contains a symbolic link from /shome to /sfs/new-york.lcs.mit.edu:85xq6...dak55ue/pub/shome, the exported certification directory.

Since the self-certifying pathname within the /shome link is secure, any pathname within the /shome directory is certified by our file server. By default, our agents are configured to register a certification program that searches /shome. The command (sfskey invocation) is shown below which enables this behavior along with the contents of the /shome directory.

[sure:]$sfskey certprog dirsearch -l /shome

[sure:]$ls -1 /shome/

lrwxr-xr-x 1 root wheel 59 Jun 28 22:30 am@ ->
/sfs/amsterdam.lcs.mit.edu:tqqzic2qsik3k57s2mfnupcdq84xerz
lrwxr-xr-x 1 root wheel 59 Jul 21 20:37 butt@ ->
/sfs/naugabutt.lcs.mit.edu:w75e2acqs5zqfkhvst4cym79537pi3r3
lrwxr-xr-x 1 root wheel 66 Jul 29 08:04 cif@ ->
/sfs/civil-forfeiture.lcs.mit.edu:7p4ryy2ubf464d7xduw5ehz2xxmwe96
While REX does accept self-certifying hostnames on the command line, this is often inconvenient. After the above certification program is enabled, REX can be invoked with the short names listed in /shome. For example, the following shows a REX session to fourplay.lcs.mit.edu.

```
[sure:~]$rex fourplay
rex: RESULT: /sfs/fourplay.lcs.mit.edu:q2ginkx6ms4papy2b6fq35fqh6q8if9y
rex: Connecting to fourplay.lcs.mit.edu:q2ginkx6ms4papy2b6fq35fqh6q8if9y
rex: made channel: socklisten -x
rex: made channel: /usr/X11R6/bin/xauth add :2 MIT-MAGIC-COOKIE-1 92a5684790a303302557270392dfe970
rex: made channel: ttyd ericp-dev .
[ericp@fourplay]$`

However, a user within our environment might want to connect to a REX server outside of LCS, for example at another university. The LCS system administrators might be unwilling to add this outside machine to the certification server because they don't trust it. However, since REX does not perform internal key management, the user is free to either specify the self-certifying hostname of the outside server as an argument to REX or
to configure his agent to map a short name to it. The example below shows an invocation of REX that does not use the /shome certification directory and an example of registering a custom certification program.

[sure:~]$rex leland.stanford.edu:gdfsgef6msre1lkdf45ldknjfdjksfj

[sure:~]$sfskey certprog dirsearch -l /sfs/leland.stanford.edu:gdfs...jfdjkfj

5 Implementation

This section discusses in more detail the modules that implement the REX remote execution service. The first part of the section will discuss the modules that manage REX sessions. These modules are responsible for authenticating sessions and implementing the REX network protocol. The latter part of the section will discuss modules that provide access to remote resources such as TCP port forwarding and pseudo-tty allocation.

5.1 Sessions

On the server side of REX, the rexd and proxy modules are responsible for maintaining sessions. The rexd module provides a means for clients to create session processes and efficiently (without public key cryptography) connect to these processes. In order to create a session, a client first authenticates itself to the REX server using public key cryptography. REX uses the SFS user authentication protocols, which are described in Mazières' dissertation[4].

After user authentication has been performed, rexd can spawn off remote processes on the client's behalf since it has the client's UNIX credentials. Clients can accomplish this through the REXD SPAWN call of the REXD PROG RPC interface that rexd exports. See appendix A for prototypes of all REX specific RPC interfaces. The REXD SPAWN call
also establishes a master session key between the client and server, which was described in section 3.2. Since the REXD.SPAWN call involves establishing the \textit{MasterSessionKey}, it should only be made over an encrypted channel. Therefore, spawning a remote process using \textit{rexd} involves public key cryptography in order to authenticate the user and to create a secure channel over which the \textit{MasterSessionKey} can be established.

The REXD.ATTACH procedure allows clients to connect to remotely spawned processes. The attach procedure takes as arguments the \textit{MasterSessionID}, the sequence number for that session, and a hash of the session key. The \textit{MasterSessionID} specifies which remote process to attach to. The \textit{MasterSessionID} and sequence number together uniquely specify a session key. However, the session key can only be calculated from these arguments if the \textit{MasterSessionKey} is known. Since only the rex client and \textit{rexd} know the \textit{MasterSessionKey}, a REXD.ATTACH call can be made over an unencrypted channel.

The REXD.ATTACH call will fail if the sequence number specified has already been used. This prevents an active network attacker from replying old REX sessions. An active attacker might still cause problems by exhausting the available sequence numbers, since REXD.ATTACH calls can be made over unauthenticated connections. REX prevents this attack by verifying the hash of the session key presented as an argument to REXD.ATTACH before updating sequence state.

When a REXD.ATTACH call succeeds, the client connection along with an encoding of the session key is passed off to the process through the REXCTL.PROG RPC interface. \textit{rexd} performs as little as possible since it must run as root. The REX network protocol described in section 3.3.1 is instead implemented by the \textit{proxy} server. The first time a client connects to a REX server, \textit{proxy} is spawned, which requires public key cryptography. All subsequent connects attach to the running proxy without public key cryptography since the
REXD_ATTACH call can be made over an unauthenticated channel.

The proxy server implements the REX network protocol. REX_PROG and REXCB_PROG are respectively the server and client side of the REX network protocol. All REX network protocol traffic is protected by the session keys that were generated from the MasterSessionKey. These interfaces are asymmetric primarily to avoid naming collisions. For example, file descriptor and channel numbers are always selected by the server side. In addition, there’s no need for the client to export a REX_MKCHANNEL procedure since only the client selects which modules to run. proxy exits when it has no client connections to service.

5.2 Client session modules

The rex program allows users to connect to remote servers using the REX network protocol. However, communicating with a set of remote modules requires that a proxy server be running. While rex could spawn the proxy server, this would require public key authentication every time it is called. Therefore, proxy needs to be spawned by something that persists, even across rex executions.

The sfsagent was modified to spawn proxy sessions and pass session keys to rex on demand. sfsagent was a prime candidate for this task, since SFS user authentication requests are fulfilled by a single sfsagent that persists in the background.

The rex program first passes the self-certifying hostname of the destination machine to sfsagent. If this is the first request sfsagent has received for that hostname, it will remotely spawn a proxy a server and store the MasterSessionKey. In addition, the sfsagent keeps a connection open to prevent the proxy server from exiting and will also keep track of the next available sequence number. The sfsagent will then pass the MasterSessionID, a session key, and its corresponding sequence number back to rex, providing enough information for rex.
to attach to *proxy*.

*sfsagent* connections can also be forwarded across a REX connection. Consequently, users don’t have to start a user agent on every machine they connect to. Another benefit of forwarding agents is that a user can efficiently (without public key cryptography) establish REX sessions with any server that they have previously accessed using the same agent. Therefore, when a user’s *sfsagent* has been forwarded to a set of servers, REX can create connections efficiently between any two of them. Figure 3 shows the connections among a set of machines that have been connected to using the same agent. In this example, the user initially logged into host *A* and started an *sfsagent*. The user then used *rex* to connect to servers *B* and *C* and, from within the *C* REX session, remotely connected to *D*. The connections between *sfsagent* and the *proxy* servers not only prevent *proxy* from exiting, but also forward agent requests back to the single *sfsagent* running on *A*. Agent forwarding allows the authentication state of all machines a user accesses via *rex* to be stored centrally in the user’s *sfsagent*. As a result, REX could efficiently (without public key cryptography) form a connection between hosts *B* and *D* even though the user hasn’t previously connected between these machines.

5.3 Remote resource modules

Remote resources such as pseudo-terminals and TCP ports are accessed through REX modules. These modules are spawned by clients using the REX network protocol, which *proxy* exports. The REX network protocol also provides a UNIX domain socket interface to these remote modules. Both data and file descriptors can be passed along UNIX domain sockets.

Most REX modules are used to access remote file descriptors. These modules provide
Figure 3: Example of connections between agents and proxy on different machines with agent forwarding

REX clients with access by writing file descriptors to their stdout. For example, *ttyd* writes a file descriptor connected to the client side of a newly allocated pseudo-terminal to it’s stdout. *Listen* modules, such as *tcplisten* or *unixlisten*, will initially bind to respectively a TCP port or pathname and then write accepted file descriptors to their stdout. These type of modules can also write data to stdout or read from stdin in situations where additional information about the connection needs to be communicated. For example, the pseudo-terminal allocation module, *ttyd*, will read window change messages on stdin so that it can perform an ioctl on the pseudo-terminal when clients resize their terminals.

### 5.4 Secure IPC through *suidconnect*

While the previous section discussed modules that are spawned per session, it is often necessary for modules to have a single instance which is shared across all REX sessions on a machine. For example, certain modules perform tasks that require root privileges,
such as ptyd allocating pseudo-terminals. If these type of modules were spawned by clients through proxy, they would have to be setuid root. However, then any user could call the program, giving attackers an opportunity to exploit poor buffer management or exhaust system resources.

Instead, a single instance of these root modules is spawned during system initialization. Users connect to the single instances of these modules through the suidconnect interface of SFS[4]. suidconnect not only allows clients to connect to these modules by name through a UNIX domain socket, but also passes the UNIX credentials of the client to the module.

In the suidconnect interface, single instance modules bind to pathnames in a private directory, /var/sfs/sockets. Only processes with special group permissions can access this directory. Therefore, clients cannot directly connect to pathnames in this protected directory. They instead have to call the suidconnect binary, which has setgid permissions that allow it to access this directory. The suidconnect binary then connects to the requested pathname, sends the UNIX credentials of the client to the module, and passes the newly created connection back to the client. suidconnect not only provides a convenient means for inter-process communication but also provides a secure means for server modules to perform access control.

The suidconnect interface is used by ttyd to communicate with ptyd, which must run as root so that it can allocate a pseudo-terminal. Since ptyd is informed of the client’s UNIX credentials, it can limit the number of pseudo-terminals allocated by a single user, thereby preventing resource exhaustion attacks.

suidconnect is also useful in situations in which there is an instance of a module for each user. Traditionally, the server process will bind to a pathname in a temporary directory. The user must then set an environment variable that provides clients with the pathname to
the server module. However, a client in an environment that is not inherited from the environment in which the module was spawned will not be able to connect. *sfsagents* do not have the usability problems associated with environment variables since clients connect through the *suidconnect* interface. On system initialization, *sfscd* binds to the agent.sock pathname in the *suidconnect* directory. When an *sfsagent* starts up, it connects to agent.sock through *suidconnect*. Any subsequent connects to agent.sock as that user will get passed back a connection to the corresponding user’s agent. *sfscd* accomplishes this by holding connections to user’s agents. When a client connects to agent.sock, *sfscd* creates a socket pair, sends one side back to the client and the other side to the agent whose UNIX credentials matches that of the client. Therefore, a requesting process is matched up with an agent based on it’s UNIX credentials, not an environment variable.

6 Performance

This section presents the results of performance tests comparing REX with SSH. The tests focused on session latency since these are the areas in which REX’s persistent authentication and modular design should have the most dramatic effect. Tests were performed both on high latency connections, to emphasize latency due to round trips required by the protocols, and low latency connections, to emphasize latency due to CPU usage. The high latency tests were performed between two machines with a .1 second round-trip time.

Figure 4 shows the result of measuring the delay for executing a simple command (/bin/echo) on a remote machine. The test consisted of measuring the time it took to execute a null command (in this case /bin/echo with no arguments) on a remote machine. For this test, all interactive session features, such as X-forwarding, agent forwarding, and pseudo-terminal allocation, were disabled. The results show that REX performs worst on
the first connect and best on subsequent connects, displaying the benefits of persistent au-
thentication. REX performs significantly better than SSH on subsequent connections (SSH
performs a factor of 2.6 worse on high latency and 6 on low latency). The results also
indicate that utilities like CVS that pipe through non-interactive sessions when invoked will
perform better through REX.

Figure 5 shows the result of measuring the delay for executing an interactive command.
The test consisted of measuring the time it took to execute a null command (/bin/echo
again) with X-forwarding, agent-forwarding, and pseudo-terminal allocation enabled. These
features are typically enabled for interactive sessions. REX performs worse (by 86%) on
the low latency test due to the extra computation involved in spawning modules for each
feature (socklisten, suidconnect, and ttyd respectively). However, REX still outperformed
SSH on the high latency test.

The fact that REX performed comparably to SSH on the interactive test shows that REX
achieves modularity without sacrificing efficiency. Since commands that invoke and tunnel
through REX or SSH connections, the primary motivation for low latency design, do not
need interactive session features, the command test, in which REX performs significantly
better, is more significant.

7 Status

REX is currently in a usable state but has been minimally tested. REX currently supports
TCP port forwarding, pseudo-tty allocation, and agent forwarding. Important tasks that
still need to be completed are improved error handling and reporting, X forwarding, an
interface for listing and invalidating sessions, and an anonymous execution service.
Figure 4: Measures the latency for simply spawning a remote command. A pseudo-terminal is not allocated and no ancillary channels such as X-forwarding are created.

Figure 5: Measures the latency for setting up an interactive login session. The session includes an X-forwarding channel, an agent forwarding channel, and pseudo-terminal allocation.
A  REX RPC prototypes

REX remote procedure calls are encoded using the Sun RPC XDR specification[5]. Below are the prototypes for REX specific RPC protocols. The \texttt{REX.PROG} and \texttt{REXCB.PROG} protocols implement the REX network protocol. The \texttt{PTYD.PROG} protocol describes the pseudo terminal allocation interface for \texttt{ptyd}. The \texttt{REXD.PROG} protocol implements the interface to \texttt{rexd} for spawning and connecting to programs. The \texttt{REXCTL.PROG} describes the interface that allows \texttt{rexd} to pass session key information (along with a socket connected to a client) to \texttt{proxy}.


/*
 * This file was written by David Mazieres and Eric Peterson. Its contents
 * is uncopyrighted and in the public domain. Of course, standards of
 * academic honesty nonetheless prevent anyone in research from
 * falsely claiming credit for this work.
 */
#endif include "sfs_prot.h"

typedef string ttypath<63>;
typedef string utmphost<16>;

/* Note, a successful PTYD_PTY_ALLOC result is accompanied by a file
 * descriptor for the master side of the pty. */
union pty_alloc_res switch (int err) {
 case 0:
 ttypath path;
 default:
 void;
 };

program PTYD_PROG {
 version PTYD_VERS {
 void
 PTYD_NULL (void) = 0;

 pty_alloc_res
 PTYD_PTY_ALLOC (utmphost) = 1;

 int
 PTYD_PTY_FREE (ttypath) = 2;
 } = 1;
 } = 344431;
typedef string rex_progarg<>;
typedef rex_progarg rex_cmd<>;
typedef string rex_envvar<>;
typedef rex_envvar rex_env<>;

struct rexd_spawn_arg {
    sfs_kmsg kmsg;
    rex_cmd command;
};

struct rexd_spawn_resok {
    sfs_kmsg kmsg;
};

union rexd_spawn_res switch (sfsstat err) {
    case SFS_OK:
    rexd_spawn_resok resok;
    default:
        void;
};

struct rexd_attach_arg {
    sfs_hash sessid;
    sfs_seqno seqno;
    sfs_hash newsessid;
};

#elif 0
struct rexdAttachResok {
};

union rexdAttachRes switch (sfsstat err) {
    case SFS_OK:
    rexdAttachResok resok;
    default:
        void;
};
#else
typedef sfsstat rexdAttachRes;
#endif

struct rex_sesskeydat {
    sfs_msgtype type; /* = SFS_KSC or SFS_KCS */
    sfs_seqno seqno;
    sfs_secret sshare; /* Server's share of session key */
    sfs_secret cshare; /* Client's share of session key */
};

program REXD_PROG {
version REXD_VERS {
    void
    REXD_NULL (void) = 0;

    /* Must be accompanied by a previously negotiated authuint */
    rexd_spawn_res
    REXD_SPAWN (rexd_spawn_arg) = 1;
    rexd_attach_res
REXD_ATTACH (rex_attach_arg) = 2;
} = 1;
} = 344424;

program REXCTL_PROG {
    version REXCTL_VERS {
        void
        REXCTL_NULL (void) = 0;
        /* REXCTL_CONNECT is preceded by a file descriptor */
        void
        REXCTL_CONNECT (sfs_sessinfo) = 1;
} = 1;
} = 344426;

struct rex_payload {
    unsigned channel;
    int fd;
    opaque data<>;
};

struct rex_mkchannel_arg {
    int nfds;
    rex_cmd av;
    rex_env env;
};

struct rex_mkchannel_resok {
    unsigned channel;
};

union rex_mkchannel_res switch (sfsstat err) {
    case SFS_OK:
        rex_mkchannel_resok resok;
    default:
        void;
};

struct rex_int_arg {
    unsigned channel;
    int val;
};

struct rexcb_newfd_arg {
    unsigned channel;
    int fd;
    int newfd;
};

struct rex_setenv_arg {
    string name<>;
    string value<>;
};

typedef string rex_unsetenv_arg<>;

program REX_PROG {
    version REX_VERS {

    29
void
REX_NULL (void) = 0;

bool
REX_DATA (rex_payload) = 1;

/* val is fd to close, or -1 to close channel */
bool
REX_CLOSE (rex_int_arg) = 2;

/* val is signal to deliver */
bool
REX_KILL (rex_int_arg) = 3;

rex_mkchannel_res
REX_MKCHANNEL (rex_mkchannel_arg) = 4;

bool
REX_SETENV (rex_setenv_arg) = 5;

void
REX_UNSETENV (rex_unsetenv_arg) = 6;
}) = 1;
}) = 344428;

program REXCB_PROG {
version REXCB_VERS {
void
REXCB_NULL (void) = 0;

bool
REXCB_DATA (rex_payload) = 1;

bool
REXCB_NEWFD (rexcb_newfd_arg) = 2;

/* val is exit status or -1 for signal */
void
REXCB_EXIT (rex_int_arg) = 3;
}) = 1;
}) = 344429;

References


Work in progress.


