Flexible Key Management with SFS Agents

by

Michael Kaminsky

B.S., Electrical Engineering and Computer Science
University of California, Berkeley, 1998

Submitted
to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Michael Kaminsky, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2000

Certified by . . . .
M. Frans Kaashoek
Associate Professor, MIT Laboratory for Computer Science
Thesis Supervisor

Certified by . . . .
David Mazières
Graduate Student, MIT Laboratory for Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Flexible Key Management with SFS Agents

by

Michael Kaminsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

One of the primary barriers to deploying secure systems is key management—the problem of mapping real-world notions of identity onto cryptographic keys. Systems typically mandate a particular way of performing key management, which is generally ill-suited to many situations. The wrong key management policy can reduce security and severely inconvenience people. Thus, to avoid unnecessarily burdening users, a system must allow people construct new key management schemes to meet unanticipated needs.

This thesis describes a toolkit approach to key management: implementing key management by plugging together small component utilities in the spirit of Unix command pipelines. We present a key management framework that facilitates this approach based on agents. Agents are separate processes that runs on a client, answering key management queries on behalf of the user. This framework lets users invoke simple, one-line commands to choose among key management techniques as varied as certification authorities, secure password protocols, Kerberos, SSL, SSH, and several novel techniques.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor, MIT Laboratory for Computer Science

Thesis Supervisor: David Mazières
Title: Graduate Student, MIT Laboratory for Computer Science

Acknowledgments

The SFS project and pluggable agent architecture are joint work with David Mazières and M. Frans Kaashoek. SFS-HTTP is joint work with Eric Banks. This research was partially funded by an NSF Graduate Research Fellowship.

for
my parents

Contents

1 Introduction **11**

2 Background **13**

3 Pluggable agent architecture **15**
3.1 Base functionality 15
3.1.1 User authentication 16
3.1.2 Server authentication 16
3.1.3 Key revocation 17
3.2 Architecture 17
3.3 Certification programs 18
3.4 Server key revocation 19
3.5 API/Protocols 20
3.5.1 The RPC programs 20
3.5.2 Agent identities 21

4 Evaluation **23**
4.1 Server authentication 23
4.1.1 Certification paths 23
4.1.2 Kerberos 24
4.1.3 SSL 25
4.1.4 SSH-style 25
4.2 Key revocation 26

5 Key management for the Web **27**
5.1 Self-certifying URLs 28
5.2 Key management examples 28
5.3 SFS-HTTP 29
5.4 Summary 30

6 Related Work **31**

7 Summary **33**

List of Figures

2-1 Overview of SFS 14

3-1 SFS agent architecture 17

5-1 A self-certifying URL 28
5-2 The SFS-HTTP system 29

Chapter 1

Introduction

One of the principal challenges for secure distributed systems is managing encryption keys. Given the right keys, cryptography permits secure communication over untrusted networks like the Internet. To make meaningful use of such communication, however, one must somehow map real-world notions of identity onto cryptographic keys—for instance, obtaining the public key of a well-known company, or of a private server on which one has an account. Unfortunately, no fixed mapping fits all situations. However, for any given real-world situation, there generally does exist a corresponding way of conveniently managing keys. Thus, to support the broadest possible range of uses, a system must let people construct new key management schemes to meet unanticipated needs.

This thesis describes a toolkit approach to key management. We describe a framework in which, rather than manage keys in a monolithic piece of software, one does so by plugging small utilities together in a style reminiscent of Unix command pipelines. As a result, powerful key management mechanisms have implementations short enough to be typed at the command prompt. Different mechanisms can be cascaded or bootstrapped over one another to manage keys in ways not previously possible. Moreover, a number of commands and utilities developed for other purposes can now be exploited for managing keys.

We implemented the toolkit framework in an *agent* program that manages keys for two applications—the SFS global file system [MKKW99] and SFS-HTTP [KB]. SFS separates key management from file system security, making it an ideal target for innovative key management techniques. SFS also names remote files by the public keys of their servers, making the keys easily manipulable through ordinary file system calls. SFS-HTTP takes SFS's approach to security and applies it to the web and SSL [FKK96]. Both systems invoke user-supplied agent programs to authenticate users and to translate human-readable server names to public keys.

The principal building blocks for key management in our system are *certification programs*—utilities that take a human-readable name as an argument and print either another name or the ASCII-encoded specifier of a public key to standard output. Users give their agents an ordered list of certification programs with which to map names to public keys.

The SFS file system calls into agents to resolve unknown file names. The agents run certification programs from the list to produce a public key or another name. The SFS file system encodes the results as symbolic links mapping one file name to another. SFS-HTTP similarly obtains public keys for URLs through agents, and maps one URL to another by

means of HTTP redirects.

Certification programs can leverage the SFS file system to great advantage. By placing key management data in the file system, one can ensure the data's integrity with the security of the file system. Because SFS names remote files by public keys, one can bypass all key management to access trusted servers with known keys. Moreover, because the results of certification programs manifest themselves in the file name space as symbolic links, any certification program can invoke any other by simply accessing a file of the appropriate name. Thus, most certification programs can accomplish their task with little more than a few file system calls. Very few certification programs need even contain any cryptography.

The remainder of this thesis is organized as follows. First, we present a brief background of SFS for readers who are unfamiliar with the system. Then, Chapter 3 describes SFS agent which are designed around the idea of certification programs. Chapter 4 evaluates our framework by showing a number certification programs in action. Finally, we present SFS-HTTP as an example of our framework's wider applicability.

Chapter 2

Background

This chapter provides background information and terminology for SFS [MKKW99] used throughout this thesis. SFS is a secure, global network file system that separates key management from file system security. SFS accomplishes this by specifying the public keys of servers in file names known as *self-certifying pathnames*. By specifying keys in this way, self-certifying pathnames free the file system from the need for any key management to map file names to keys.

Each self-certifying pathname has the form `/sfs/`*Location*`:`*HostID*`/`. *Location* is simply a DNS hostname or IP address; it is a hint telling the file system how to connect insecurely to the server for that file name. *HostID* is a cryptographic hash specifying a server's public key [MKKW99]. *HostID*s supply the file system with enough information to communicate securely with the server for a particular file. Underneath self-certifying file names (which are typically directories), file names correspond to files in the server's local file system.

Of course, while SFS eliminates the need for the file system to manage keys, it does not eliminate the need for key management. Instead, it reduces key management to the problem of securely obtaining a server's *HostID*—or finding the correct file name to access.

Symbolic links play an important role in SFS, allowing users to assign short, convenient names to long, hard-to-type self-certifying pathnames. When a user traverses a symbolic link that points to a self-certifying pathname, the SFS file system client sees the full pathname and establishes a secure connection to the server. Symbolic links can exist locally or on remote SFS file systems. In the second case, SFS file servers can play the role of a trusted certification authority, assigning human-readable names to the self-certifying pathnames of other servers.

A third type of symbolic link results from a mechanism called *dynamic server authentication*. Users run agent programs that have the power to create symbolic links on-the-fly in the `/sfs` directory. When a user references a non-self-certifying file name in `/sfs`, the file system calls into his agent, giving it a chance to create a symbolic link and redirect the user's file access. In this way, agents have the power to authenticate servers by mapping human-readable file names to self-certifying pathnames containing the appropriate *HostID*.

The file system also calls into a user's agent the first time he accesses a new self-certifying pathname. In this case, the agent has the opportunity to authenticate the user to the remote server, or to verify that the server's public key has not been revoked.

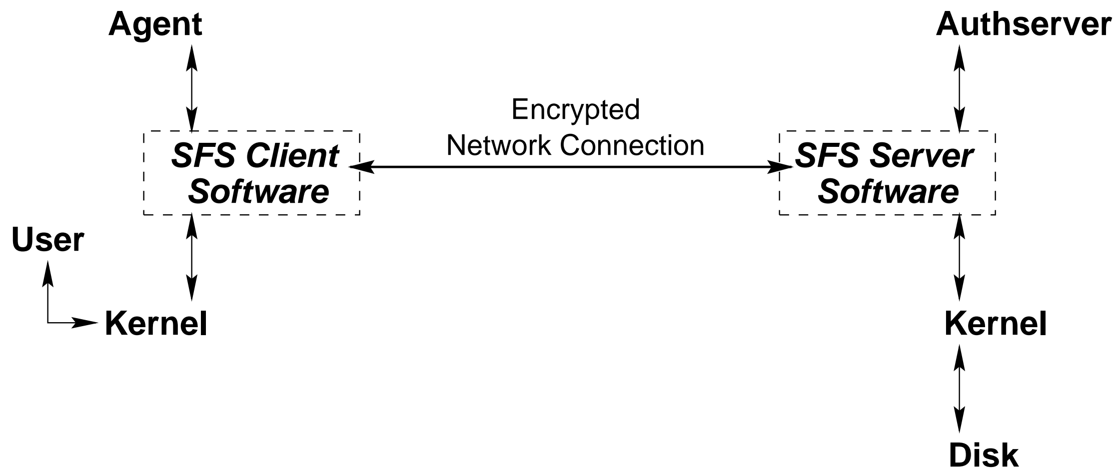

Figure 2-1: Overview of SFS

Figure 2-1 shows an overview of SFS file system software. At the highest level, SFS divides into components that run on the client and on the server; the two sides communicate through a cryptographically secured channel. The client components ("SFS Client Software") consist of the SFS *client master* which coordinates between the other pieces of the software and which manages the global `/sfs` directory, individual file system clients, and the SFS agent. Currently, there are two file system protocols: read-write, for standard file system access; and read-only, for highly available, replicated data.

On the server side of the connection, the SFS *server master* accepts all connections from clients and dispatches them to the appropriate component of the server software. After incoming connections are handed off, the specific client and server components communicate directly. The final component on the server side of the connection is the SFS authentication server (the *authserver*) which handles user authentication for the file system. All SFS components are written in C++ and use Sun RPC to communicate with each other; furthermore, all IO and RPC is asynchronous.

Chapter 3

Pluggable agent architecture

This chapter describes a framework for extensible key management implemented in the SFS agent program. The agent allows the user to plug in small key management utilities to handle the authentication process. The user registers these utilities, called *certification programs*, with the agent. When the agent subsequently gets a query from the file system, it invokes the certification programs to determine the appropriate response. The pluggable interface to key management lets different users easily employ different techniques and policies.

This chapter begins with a discussion of the base functionality required of an SFS agent by the file system. The next section describes the high-level architecture of the agent as it fits into a complete system. We then give a detailed description of how certification programs work and get plugged into the agent. We discuss a related mechanism, *revocation programs*. Finally, we discuss the RPC protocols between the SFS agent and the rest of the system. Throughout this chapter, we will talk about agents in the context of the SFS file system; Chapter 5 will demonstrate their wider applicability.

3.1 Base functionality

As described in [MKKW99], an agent is a separate process that runs in the background to perform authentication or key management on behalf of a user. When the SFS system software needs information from a user, it simply queries his agent asynchronously. Agents are unprivileged, so that users can modify their agents or replace them entirely. Furthermore, SFS agents support *agent forwarding* similar to the SSH [Ylö96] agent. If a user logs into a remote machine and then accesses a file over SFS, the remote machine's SFS client software can contact the user's local agent.

This section describes the basic functions an agent must perform for the file system, and the goals we had for implementing those functions. In each case, we sketch the design decisions that allow the agent to achieve its goal.

3.1.1 User authentication

The first function of the SFS agent is to authenticate users to remote file servers. User authentication needs to be transparent and easy for remote file access because in a global file system, users can easily touch files on several different machines without noticing. If the authentication process were interactive, access to a new file server would have to stall and wait for the user to respond (for instance, by typing a password).

Therefore, the agent operates by keeping a proof of the user's identity in memory. Typically, this means the agent holds the user's private key, and the file servers to which he has access store the corresponding public key. When a user accesses a remote file server for the first time, the client file system software contacts his agent and asks it to sign an authentication request. If the server can successfully verify the signature, it maps file system requests from that user on the client to the appropriate credentials on the server.

Because the agent keeps the user's private key in memory, it can authenticate the user without manual intervention. The user typically loads his key into the agent when he starts it by typing in a passphrase. The passphrase allows the user to safely store an encrypted version of the key on disk. The user authentication mechanism in the SFS agent is similar to that found in the SSH agent.

3.1.2 Server authentication

The second function of the SFS agent is to provide an easy and convenient way for the user to customize server authentication. Traditionally, key management in file systems is built into the file system software itself by its designers and cannot be modified or replaced. Users are bound to a particular scheme, which may not be appropriate for their environment and needs.

Agents offer flexible server authentication by offering the user the opportunity to map file names (for instance, DNS host names) to the public keys of file servers (or, more specifically, to *HostID*s). The SFS software automatically mounts remote file systems when a user accesses them for the first time. During the mounting process, the SFS client software verifies the identity of the file server using the *HostID*. Because *HostID*s are supplied by the user (either directly or via his agent), the means by which he names files determines his key management policy.

With dynamic server authentication, the SFS client software contacts the user's agent at mount-time to assist in authenticating the server. The agent is free to invoke arbitrary programs in order to map the given server's name to a *HostID*. These programs have the freedom to employ any existing key management techniques or novel ones to resolve the name. If successful, the agent returns the *HostID* to the file system client software which mounts the remote server. Otherwise, the client will not be able to mount the server. Finally, the file system client creates a symbolic link for the user under the `/sfs` directory from the server's name to its self-certifying pathname.

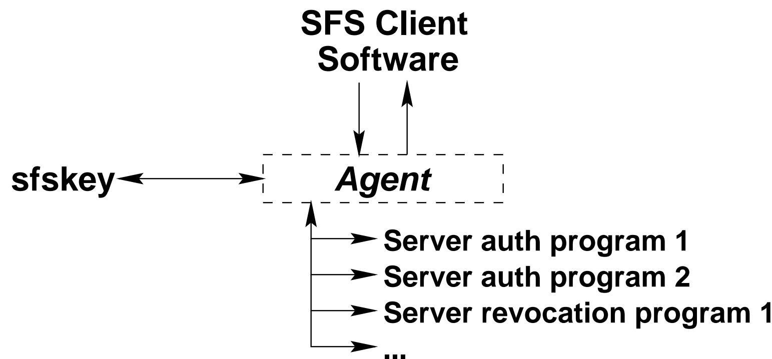

Figure 3-1: SFS agent architecture

3.1.3 Key revocation

The final function of the SFS agent is to allow the user to manage key revocation policies. Revocation is another aspect of key management that is often built into systems or into key distribution mechanisms. The agent architecture, as with server authentication, provides a way for the file system client to act based on the user's preference. When the client attempts to mount a remote file system, it again contacts the requesting user's agent, asking it for any revocation information it has for the given host. The SFS agent supports two types of revocation.

First, the agent can present a *key revocation certificate* for the given *HostID*. Key revocation certificates are self authenticating in that they are signed by their own private keys. Thus, if their private key is compromised, site administrators can issue key revocation certificates that anyone can trust. If the agent returns a properly signed revocation certificate, the file system client will refuse to mount the server.

Second, in addition to key revocation certificates, users can configure their agents to simply block access to specific file servers. The technique of *HostID blocking* allows users to arbitrarily decide that they do not want to access a particular set of servers (e.g., they do not trust the file server administrators). At mount time, the client asks the agent if the given server is acceptable. If the user is blocking that server, the SFS client will refuse to allow access.

3.2 Architecture

Figure 3-1 shows that the agent communicates both with the SFS client software ("above" it) and with arbitrary certification programs ("below" it). A third interface is available for the user to configure his agent with a program *sfskey*. The modular design of the SFS agent architecture allows it to accomplish the goals listed above.

The flow of information is as follows. When the SFS client sees a file system request

from a user, it first attempts to do dynamic server authentication,[1] querying the user's agent for the server's *HostID*. The agent, based on the user's configuration, launches one or more external programs to map the server's name to a *HostID*. The agent returns the full self-certifying pathname to the file system client.

Next, the SFS client asks the agent for revocation information. The client sends the agent the server's self-certifying pathname. The agent again launches any external programs defined by the user to potentially map the server to a revocation certificate. If such a certificate is available, or if the agent is *HostID* blocking that server, the agent notifies the SFS client, which denies the user access to the server.

If the *HostID* is not revoked or blocked, the agent will then attempt to authenticate the user to the server. The SFS client generates an authentication request and passes it to the agent, which signs it with the user's private key. The agent returns the signed authentication request to the client, which sends it over the encrypted channel to the SFS server. If the user has loaded more than one private key into the agent, the SFS software repeats the entire user authentication process until it succeeds or it exhausts all available keys.

Users configure the SFS agent through a utility program called *sfskey*. Users can load private keys into the agent from disk or over the network from an authentication server using SRP [Wu98]. With *sfskey*, users also register pluggable key management programs with the agent telling it how to respond to server authentication and revocation queries. All configuration is dynamic; changing user private keys or key management policies takes effect immediately.

The SFS agent architecture supports diverse configurations by allowing the user to run more than one SFS agent concurrently. Users associate processes with a particular agent, determining what key management policies apply to the process. Chapter 3.5.2 describes the mechanism by which processes are mapped to agents. Moreover, even a single process can have different key management policies based on the file server being contacted. The user can define multiple key management policies, each of which can be associated with different servers based on regular expressions. This idea is discussed in more detail in the next section.

3.3 Certification programs

Each user has a different view of the `/sfs` directory. *sfsagent* can create symbolic links in `/sfs` on-the-fly. When a user accesses a file other than a self-certifying pathname in `/sfs`, the SFS file system client notifies the user's agent of the event, giving the agent a chance to create a symbolic link and redirect the user's access. This mechanism can dynamically map human-readable names to self-certifying pathnames.

sfsagent has no built-in interpretation of non-self-certifying names in `/sfs`. Instead, it relies on certification programs to map those names to other pathnames. *sfskey* gives *sfsagent* a list of certification programs to use for the task. Each entry in the list is a 4-tuple

[1]The SFS file system client caches the results of user and server authentication requests, and thus only contacts the agent when required (usually when a new name is accessed).

of the form:

$$\langle \mathit{suffix},\ \mathit{filter},\ \mathit{exclude},\ \mathit{program}\ [\mathit{arg}\ldots]\rangle$$

suffix, *filter*, and *exclude* are all optional and can be left empty. *suffix* partitions the namespace of the `/sfs` directory into links managed by separate programs. If a certification program's *suffix* is non-empty, the program will only be run to look up names ending in *suffix*, and *sfsagent* will strip *suffix* from names before any further processing. *filter* and *exclude* are regular expressions to control the scope of a certification program. When *filter* is specified, the agent will only invoke the certification program on names containing the regular expression. Analogously, when *exclude* is specified, the agent will not invoke the certification program on any name containing *exclude*.

Finally, *program* is the actual certification program to run, along with the initial arguments to give that program. The final argument will be the actual name to look up. If a certification program exits successfully and prints something to its standard output, the agent will use the program's output as the contents of the dynamically created symbolic link. If not, the agent will continue down the list of certification programs.

As a simple example, suppose a directory `/mit` contains symbolic links to the self-certifying pathnames of various SFS servers around MIT. A user might want to map names of the form `/sfs/`*group*`.mit` to `/mit/`*group*, but exclude any *group* ending `.lcs`. The following certification program would accomplish this task:

```
⟨.mit,,.lcs$,sh -c 'test -L /mit/$0 && echo /mit/$0'⟩
```

The *suffix* `.mit` applies this certification program to names in `/sfs` ending `.mit`. An empty *filter* signifies the certification program applies to all names. The *exclude* regular expression excludes any names ending `.lcs`. Finally, *program* consists of three arguments, `sh`, `-c`, and a short shell script. The agent will supply the name being looked up as the final argument, which the shell calls `$0`. The script simply checks for the existence of a symbolic link in `/mit`. If one exists, it prints the path and exits successfully; otherwise, it exits with non-zero status, indicating failure.

3.4 Server key revocation

As with dynamic server authentication, *sfsagent* uses external programs to handle server key revocation. *sfskey* gives *sfsagent* a list of *revocation programs* to run on each self-certifying pathname the user accesses. Each entry in the list is a 4-tuple of the form:

$$\langle \mathit{block},\ \mathit{filter},\ \mathit{exclude},\ \mathit{program}\ [\mathit{arg}\ldots]\rangle$$

block is a boolean value that enables *HostID* blocking in addition to key revocation. *filter* and *exclude* are regular expression filters, as with certification programs, that here get applied to the *Location* part of self-certifying pathnames. *program* is the program to run to check the validity of a self-certifying pathname. The pathname's *HostID* is appended to the program's arguments.

A revocation program can write a self-authenticating revocation certificate to standard

output. If it does so, the agent uploads the certificate to the file system client, which subsequently blocks all access to files under that pathname by any user. If the program exits successfully but does not output a revocation certificate, then, in the case that the *block* flag is set, the agent triggers *HostID* blocking, preventing its own user from accessing the self-certifying pathname without affecting other users.

Because revocation programs may need to access files in SFS, there is a potential for deadlock. For instance, a revocation program cannot synchronously check the validity of Verisign's self-certifying pathname while also searching for revocation certificates under `/verisign` (a symbolic link to Verisign's hypothetical certification authority). Thus, *sfskey* also gives *sfsagent* a "*norevoke list*" of *HostID*s not to check synchronously. *HostID*s in the norevoke list can still be revoked by uploading a revocation certificate, but this is not guaranteed to happen before the user accesses the revoked pathname.

3.5 API/Protocols

The SFS agent has three interfaces. For communication to the SFS file system client software and for configuration with *sfskey*, the agent uses Sun RPC [Sri95]. Certification programs simply take their arguments and print their results in plain ASCII text. This makes it easy to build server authentication out of standard Unix file and text utilities.

3.5.1 The RPC programs

The SFS agent defines three RPC programs through which it communicates with the rest of the system. Two programs allow the agent to call into the file system client (`AGENT`) and vice-versa (`AGENT-CALLBACK`). The third program is for the user to configure the SFS agent (`AGENT-CONTROL`).

The `AGENT` RPC program allows the agent to request an action from the file system client, typically in response to a direct request from user running *sfskey*. For example, if the user wants to create a special symbolic link under `/sfs` for dynamic server authentication, the agent uses a `SYMLINK` RPC to request that the SFS client software create the link. Similarly, if the user decides to kill his agent, the agent notifies the SFS client using the `KILL` RPC from the `AGENT` program.

The `AGENT-CALLBACK` program defines a set of procedures that the SFS client uses to call into the agent. These RPCs form the basis for the key management described above. The interface consists of three categories of RPCs corresponding to the three goals listed in Chapter 3.1. First, two procedures, `AUTHINIT` and `AUTHMORE`, allow the agent to authenticate its user to a remote file server. The user authentication mechanism is opaque to the file system itself and only needs to be agreed upon between the agent and the entity handling authentication for the file server. The `AUTHMORE` procedure allows the user authentication process to take multiple rounds.

Second, the SFS client uses an RPC `LOOKUP` to query the agent for server authentication. The RPC allows the agent to do its own key management for server authentication. The procedure takes a plain filename as an argument (often the server's name) and returns either

a fully specified self-certifying pathname, or the name of another symbolic link. The agent uses pluggable key management programs to answer the query.

The third RPC category consists of the `REVOKED` procedure. This RPC functions similarly to `LOOKUP`. Here, the file system client sends the agent a filename and asks for any relevant revocation information. The agent has three responses: none, indicating that the client is free to connect to the server; blocked, telling the client not to allow that user access; and certificate, informing the client that it has a revocation certificate for the file server in question (which it includes in the response). As with the `LOOKUP` RPC, the agent uses pluggable key management programs to generate its answer.

The `AGENT-CONTROL` RPC program allows the user to configure his agent and environment. The user invokes the procedures here directly or indirectly through *sfskey*. The `AGENT-CONTROL` program provides for the registration, removal, and listing of user-authentication keys, certification programs, and revocation programs. The remaining RPCs in this program help implement auxilliary functionality such as agent forwarding, creating symbolic links, avoiding deadlock, and seeding random number generators.

3.5.2 Agent identities

While the current agent implementation stores multiple private keys for authentication to more than one SFS server, users needing greater flexibility can also run several agent programs simultaneously. The SFS client software maps each file system request to a particular agent. It does so not simply through a process's user ID, but through a more generalized notion called the agent ID or *aid*.

Ordinarily, a process's *aid* is the same as its 32-bit user ID. Depending on a process's group list, however, the process can have a different 64-bit *aid* with the same low-order 32-bits as its user ID. System administrators set a range of reserved group IDs in the main SFS configuration file, and these groups mark processes as belonging to different *aid*s. A 150-line setuid root program *newaid* lets users switch between reserved group IDs to spawn processes under new *aid*s.

*aid*s let different processes run by a single local user access remote files with different credentials—for instance as a user and superuser. Conversely, a user may want to become root on the local machine yet still access remote files as himself. Root processes therefore can be assigned any *aid* of any user (i.e. with any low-order 32-bits). A utility *ssu* lets users become local superuser without changing *aid*s. *ssu* is a wrapper around the setuid Unix utility *su*, and thus does not itself need any special privileges.

Chapter 4

Evaluation

This section offers a means by which to evaluate agents and certification programs as tools for doing key management. Traditional performance metrics are less illustrative than a demonstration of the effectiveness of the toolkit approach to key management. Furthermore, in practice, the cost of mounting a remote file server is a public key operation, which typically overshadows the cost of an interprocess RPC and spawning a small number of certification programs.

We present several examples of how to use SFS agents to do various kinds of key management in the context of SFS. In particular, we show trivial implementations of *certification paths* (see below) as well as SSL, Kerberos [SNS88], and SSH-style authentication. Furthermore, we demonstrate how to do key revocation. In Chapter 5, we then show that the SFS agent is general purpose enough to do server authentication for the Web.

The examples below use *sfskey* to register pluggable key management programs with an SFS agent. The relevant *sfskey* usage is as follows:

```
sfskey certprog [-s suffix] [-f filter] [-e exclude] prog [arg ...]
sfskey revokeprog [-b [-f filter] [-e exclude]] prog [arg ...]
```

These invocations correspond to the 4-tuples presented in the previous chapter. In the second form, the `-b` option tells the agent to do *HostID* blocking.

4.1 Server authentication

Because the SFS agent can be configured with certification programs, the user can easily and efficiently devise entirely new authentication schemes or even implement or exploit existing key management techniques. Here, we give one example from the former category and three from the latter.

4.1.1 Certification paths

A *certification path* is a set of directories that contain symbolic links to SFS file servers. Conceptually, certification paths are similar to the `$PATH` variable in shells; directories in the path are searched for links in a predefined order, permitting the user to set up a hierarchy

of trust. A particular link name may appear in more than one directory, so directories earlier in the certification path are more trusted than those which appear later. The directories can be either on a local disk or a network file server allowing the user to rely on local certification authorities then remote, global ones, for example.

A program *dirsearch* simplifies the task of searching through a certification path for a given link name. In its basic invocation, *dirsearch* takes a list of directories followed by a name as command-line arguments. It searches each directory in turn for a file with the given name, and if successful, *dirsearch* prints out the full path to the file.

Thus, a user can easily implement certification paths. He configures his agent to run *dirsearch* as one of his certification programs. The user provides the list of directories, and the agent, at execution time, will supply the final argument, which is the name of the server to lookup. For example, if a user wants to search a personal directory `$HOME/.sfs/known_hosts` followed by a university-wide symbolic link repository `/campus`, he can run the following command:

```
% sfskey certprog dirsearch $HOME/.sfs/known_hosts /campus
```

The name `/campus` here itself is a symbolic link pointing to the user's university's public SFS file server. Furthermore, users can refine how lookups are done by registering multiple instances of the *dirsearch* program using the regular expression filters described in Chapter 3:

```
% sfskey certprog -f '\.lcs\.mit\.edu$' dirsearch /lab/links
```

4.1.2 Kerberos

Kerberos is the primary means of authentication in the Andrew File System (AFS) [HKM+88]. The two systems are tightly integrated, requiring users to fetch Kerberos tickets in order to authenticate file system requests. With the SFS agent, users can also use Kerberos to authenticate remote SFS file systems.

First, the user gets tickets as usual (typically through `kinit` or `login`). Then, he issues the following command:

```
% sfskey certprog -s krb sh -c 'rsh -x $0 sfskey hostid -'
```

When he accesses a remote server name under the `/sfs` directory with the `.krb` suffix, the agent will invoke the shell command enclosed in single quotes with the server's name (without the `.krb` suffix) as the `$0` argument. The script invokes Kerberized *rsh*, which attempts to start a shell for the user on the remote machine. The *rsh* program authenticates the user to the remote login server using Kerberos, and if successful it runs *sfskey*. The *sfskey hostid -* command connects securely to the local SFS server and asks it for its public *HostID*. The script prints the server's self-certifying pathname, which the agent sends back to the file system client.

4.1.3 SSL

The Secure Socket Layer (SSL) is the key distribution mechanism used on the Web for secure HTTP. If a user has a certificate for a server or a certificate authority public key to vouch for a certificate, he can easy use SSL to authenticate the corresponding SFS file server. Again, the user loads a one-line shell script into his agent:

```
% sfskey certprog -s ssl sh -c 'lynx -source https://$0/sfspath.txt'
```

This example assumes that server administrators are willing to distribute SFS self-certifying pathnames from a file on their Web servers called `sfspath.txt`. Here, the suffix `.ssl` tells the agent to do SSL server authenticate for names ending in `.ssl`. The script simply invokes an existing Web browser to make a secure SSL connection and download the SFS pathname.

4.1.4 SSH-style

The Secure Shell (SSH) is a popular remote login program that deals with the problem of key distribution by allowing for the possibility of a man-in-the-middle attack. The first time a user connects to a given remote machine using SSH, the client software trusts the host key with which it is presented (usually after asking the user for confirmation). The SSH client stores the key in a "known hosts" file, and, on subsequent connections, it verifies the server with the stored key.

The agent can use this idea for authenticating SFS servers. When a user accesses a remote server, a certification program looks up the server in a certification directory. If it finds a symlink by that name, it returns the corresponding self-certifying pathname using *dirsearch*. If not, the script retrieves the server's *HostID* over the network insecurely and stores it in the directory. Then on subsequent lookups, the certification program can return the self-certifying pathname to the file system client. The following code implements this idea:

```
#!/bin/sh

# Make known_hosts directory if it does not exist
linkdir=${HOME}/.sfs/known_hosts
test -d $linkdir || mkdir -p $linkdir || exit 1

# Use link in known_hosts directory if one exists
dirsearch -l $linkdir $1 && exit 0

# Retrieve and store pathname (insecurely, for the first time)
srvpath="`sfskey hostid $1`"
test "$srvpath" || exit 1
ln -s "/sfs/$srvpath" "$linkdir/$1"

# Print self-certifying pathname just retrieved
exec dirsearch -l $linkdir $1
```

4.2 Key revocation

The toolkit idea is also easily applied to revocation certificates and *HostID* blocking. As an example, a user may not want to connect to any servers in the `some-ca.com` domain, a certification authority that he does not trust. Thus, he does not want to accidentally traverse a symbolic link that points to an SFS file server in this domain. The following command instructs his agent to block such mounts:

```
% sfskey revokeprog -b -f '\.some-ca\.com$' /bin/true
```

As another example, a central authority like Verisign might keep an online directory of revocation certificates. Each file in the directory would contain a single certificate and be named by revoked *HostID*. The user could configure his agent to check the directory for a particular *HostID* as follows:

```
% sfskey revokeprog dirsearch -c /verisign/revoked
```

The directory `/verisign/revoked` contains the revocation certificates; the `-c` option to *dirsearch* says to output ("cat") the contents of the matching file instead of printing its pathname.

Chapter 5

Key management for the Web

This chapter addresses key management on the Web and attempts to show that the approach presented earlier in this thesis applies to secure systems beyond SFS. A key issue for users of a secure system like the Web (when using `https`) is that they want to be sure that when they ask to be connected to a particular server, they are actually connected to that server. The Internet, however, cannot be trusted to ensure that users will really reach the right place. As users increasingly send private information such as passwords and credit card numbers over the Web, this issue becomes more important.

The current approach to key management on the Web is based on the existence of several well-known, trusted certification authorities (CAs). Every secure Web server has an *SSL certificate* [FKK96] which contains the server's public key. For a fee, the CAs will digitally sign a Web server's certificate; later, the server can present the signed certificate to any client that asks for it. If the client successfully verifies the signature, it will proceed to set up a secure connection using the public key contained in the certificate. Clients verify signatures using a built-in set of CA public keys that ships with the browser.

The current Web security model has several problems with respect to key management. First, the certification authority model of key management may not be appropriate for all users. For example, a student in dorm room may lack the authority or money to get a certificate for his personal computer signed by a commercial CA. Alternately, an organization like MIT which relies heavily on Kerberos may prefer to have its Web users rely on pre-arranged Kerberos passwords to authenticate Web sites instead of bothering with an external CA.

Another problem with the Web's notion of key managament is that users trust any certification authority key that they have installed in their browsers. Specifically, to authenticate the server, the client needs to verify the certificate's signature with one of the CA public keys that it knows. If the client can verify the signature with *any* key, it trusts the server and establishes the secure connection. Thus, in the current state of the Web, a user's security is only as good as the *least* trusted certification authority known to his browser.

Finally, Web users suffer from the inability to directly specify a public key. If a user obtains the public key for a server he wishes to contact in a completely novel way, he has no way to tell his browser how to use it. The ability for a user to provide his own mapping from some server (i.e., a hostname) to its public key is the essential problem.

The solution to this Web security problem is to offer users the ability to use key man-

Location — *HostID* (specifies public key) — *Web page*

```
http://www.bn.com:we36hsq8xwgam3tcgzkuay8gy7rb3y8h.web/index.html
```

Figure 5-1: A self-certifying URL

agement schemes other than SSL and CAs to authenticate Web connections. Because no subset of key management techniques is sufficient, users need a generic way to do key management. Just as SFS agents provide this flexibility for the SFS file system, they can do the same for the Web. Since SFS agents invoke certification programs to handle all key management, they can be configured to authenticate Web servers in arbitrary, user-defined ways.

The remainder of this chapter discusses a solution to the Web key management problem. The following sections outline the basic concepts that comprise the solution. Section 5.3 describes the particular implementation called SFS-HTTP, including how the user interacts with the system.

5.1 Self-certifying URLs

The basic idea in SFS-HTTP is to use *self-certifying URL*s to name secure Web sites. Like self-certifying pathnames, self-certifying URLs contain a *HostID* that uniquely specifies the public key of the server. Thus, once a user obtains a self-certifying URL, the URL is by itself completely sufficient to establish a secure, authenticated connection to a remote machine. The anatomy of a self-certifying URL is shown in Figure 5-1. Self-certifying URL *HostID*s have a special suffix `.web` that distinguishes them from self-certifying pathnames.

With self-certifying URLs, users can now directly specify the public key of a server. Thus, similarly to SFS, users can authenticate Web servers by transforming a *Location* (hostname) into a *HostID* (public key). Again like SFS, the mapping process can be coordinated by an agent which runs certification programs to produce complete self-certifying URLs from hostnames.

5.2 Key management examples

With SFS agents and arbitrary certification programs, the user is able to define his own key management schemes. The examples given in Chapter 4 apply equally to the Web and can in many cases be used with little modification. The following example, specific to the Web, shows how to implement certification authorities the same way that Web browsers do.

```
% sfskey certprog -s ssl sslgetkey
```

This certification program only runs for hostnames ending in `.ssl`. The program *sslgetkey* takes the hostname as its argument and establishes an SSL connection to the

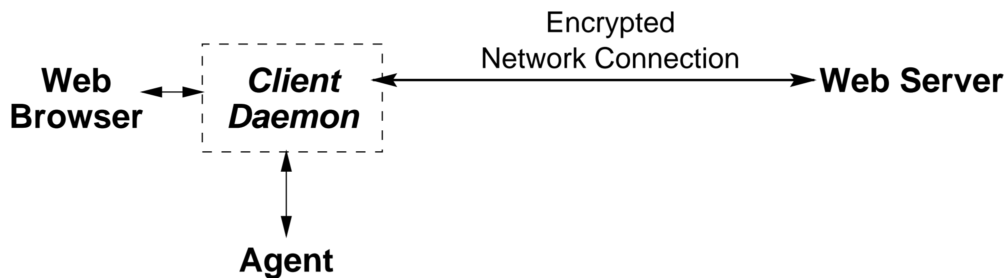

Figure 5-2: The SFS-HTTP system

Web server running on that machine. It downloads the server's certificate and, like a Web browser, has a database of CA keys with which it attempts to verify the certificate's signature. If successful, *sslgetkey* prints out the self-certifying URL for the server (i.e., with the *HostID* hashed from the public key). Thus, if the user types `http://www.bn.com.ssl`, the agent would launch *sslgetkey* to produce a URL similar to that shown in Figure 5-1. In effect, this URL is equivalent to `https://www.bn.com/`.

5.3 SFS-HTTP

The SFS-HTTP system [KB] consists primarily of one new component called the *client daemon*. Like SFS, the SFS-HTTP system is designed to run transparently with existing software; system administrators and users do not need to modify their Web server or browser software. The basic system design is shown in Figure 5-2. Boxed components are part of SFS-HTTP, and unboxed components represent pre-existing software.

Each half of the diagram represents a single machine. The right-hand side consists solely of the Web server which is assumed to speak SSL. The left-hand side of the diagram shows the client daemon, Web browser and SFS agent; all local communication between processes on the same machine is secure. The two machines communicate through an authenticated, encrypted connection.

When a user types a URL into the Web browser, the browser generates an HTTP request and sends it to the client daemon which acts like a Web proxy. The client daemon will analyze the given URL and take one of two paths. If the URL is self-certifying, then the client daemon will proceed to connect to the Web server using SSL. The browser, however, only uses SSL as its encrypted transport, *not* for key management. The client daemon verifies the public key is gets from the server using the *HostID* in the URL.

If the user enters a URL that is not self-certifying, the client daemon will contact the user's agent to do key management. As described above, the agent will decide which certification programs to launch based on one or more patterns that are tested against the server's hostname. If a certification program returns a new URL for the given hostname, the client daemon will generate an HTTP redirect. Typically, the certification program will return a self-certifying URL which the browser will use to connect to the server securely.

By default, the client daemon will revert to insecure HTTP if no certification program

can successfully perform the mapping (or there are no matching certification programs to run). For a given suffix pattern, the user can force the Web browser to only connect to servers securely by registering a certification program that runs after all others and that returns an invalid URL. The browser will be redirected to this improper URL and fail. For example, in addition to the SSL certification program given above, the uesr could register the following program:

```
% sfskey certprog -s ssl echo ":INVALID:"
```

If the *sslgetkey* program cannot map the hostname to a *HostID*, the agent will try this certification program which causes the browser to access an invalid URL.

5.4 Summary

Because current key management on the Web is limited and not very configurable, users are required to use certification authorities to do server authentication. The SFS-HTTP system uses self-certifying URLs, agents, and certification programs to allow arbitrary key management techniques on the Web.

Chapter 6

Related Work

This section first addresses work related to agents and concludes with work related to SFS-HTTP. The idea of an agent that acts automatically for a user in some capacity was first popularized by the remote login program SSH [Ylö96]. The SSH agent, however, is not general purpose and is only limited to public-key-based user-authentication for remote login. An important idea found in the SSH agent is forwarding.

SFS is the first file system to use the notion of an agent. Other file systems have a predetermined mode of host authentication. For example, NFS [Sun89] has no notion of host authentication beyond that provided by the DNS and IP routing. AFS [HKM$^+$88] uses Kerberos [SNS88] and requires that administrative realms wanting to communicate exchange keys ahead of time.

The Taos operating system [LABW92, WABL94] and the Echo file system have a notion of an authentication agent, but unlike SFS, their agent refers to a component of the operating system rather than a user-controlled process. In Taos, key management is based on a hierarchy of trust in which a machine names the path to its destination, and the source trusts each node along the way to authenticate the next one.

SFS agents can leverage other key-management approaches than the examples discussed previously in this thesis. For example, an agent could have a certication program that uses SDSI [RL], SPKI [EFL$^+$], or Secure DNS [EK97]. Our whole approach is to allow users to easily take advantage of existing schemes or combinations of existing schemes.

SFS-HTTP is related to a number of other proxy-based approaches. The Gecko NFS Web Proxy [BH] has an overall design similiar to SFS-HTTP in its tunnelling of Web requests over a different protocol, NFS. Gecko presents data from a web server as a file system by translating file system requests into Web requests and vice-versa. Users can access pages on the Web server just as they access a file system. Additionally, Gecko offers users a Web proxy by which they can use a browser (speaking HTTP) to access the system. Because Gecko uses NFS, it does not offer the security and name-mapping of SFS.

WebNFS [Cal96], from Sun Microsystems, offers similar functionality to Gecko by providing a modified version of the NFS protocol which is more adept to handling HTTP requests. Unlike SFS and Gecko, WebNFS requires users to install a new protocol (beyond stock NFS). More importantly, like Gecko, WebNFS also lacks security and name mapping.

The WebFS [Vah98] system from UC Berkeley has similar goals to the above projects by providing file system access to the URL name space, but it uses the HTTP as its network

protocol instead of NFS. A kernel module on the client interprets file system requests and passes them to a remote, user-space HTTP server. WebFS aims to provide security through certification authorities layered on top of HTTP. WebFS does not provide any proxy for browsers; users have no way to send HTTP headers to servers (making CGI impossible). Furthermore, the security model in WebFS lacks the primary advantage of SFS-HTTP: easy and arbitrary host authentication.

Chapter 7

Summary

Key management is a primary barrier to deploying secure systems. Existing key management systems are often fixed by the software designer and are immutable or awkward to configure. SFS agents provide a flexible and convenient mechanism for doing key management in a wide variety of scenarios by providing a "toolkit" approach to the problem. Any system that requires key management queries the user's agent to perform the necessary authentication. Based on the user's configuration, the agent effects the appropriate policy by invoking one or more pluggable certification programs.

Certification programs combined with agents provide a framework in which users can quickly and easily design, build, and configure new key management policies. Although the pluggable nature of certification programs provides a convenient way to use and combine existing software, the agent can ultimately execute arbitrary code to do server key management. Thus, the user has the power to authenticate a server using virtually any key management technique. In many scenarios, however, implementing new key management policies is as simple as a one-line shell script.

Though originally designed for file systems, SFS agents are general-purpose and can be used to solve the key management problem in other areas such as the Web. The SFS-HTTP system allows users to browse the Web securely without depending on SSL for server authentication.

Bibliography

[BH] Scott Baker and John H. Hartman. The gecko nfs web proxy. http://www8.org/w8-papers/5c-protocols/gecko/gecko.html.

[Cal96] B. Callaghan. WebNFS server specification. RFC 2055, Sun Microsystems, Inc., October 1996.

[EFL+] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylönen. SPKI certificate documentation. Work in progress, from `http://www.pobox.com/~cme/html/spki.html`.

[EK97] D. Eastlake and C. Kaufman. Domain name system security extensions. RFC 2065, Network Working Group, January 1997.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[KB] Michael Kaminsky and Eric Banks. SFS-HTTP: Securing the web with self-certifying URLs. from `http://www.pdos.lcs.mit.edu/~kaminsky/sfs-http.ps`.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward P. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[MKKW99] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999. ACM.

[RL] Ronald L. Rivest and Butler Lampson. SDSI—a simple distributed security infrastructure. Working document from `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[SNS88] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988. USENIX.

[Sri95] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.

[Sun89] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Working Group, March 1989.

[Vah98] Amin Vahdat. *Operating System Services for Wide-Area Applications*. PhD thesis, Department of Computer Science, University of California, Berkeley, December 1998.

[WABL94] Edward P. Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[Wu98] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[Ylö96] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.