Escaping the Evils of Centralized Control with self-certifying pathnames

David Mazières and M. Frans Kaashoek
dm@lcs.mit.edu, kaashoek@lcs.mit.edu
MIT Laboratory for Computer Science
545 Technology Square, Cambridge MA 02139

Abstract

People have long trusted central authorities to coordinate secure collaboration on local-area networks. Unfortunately, the Internet doesn't provide the kind of administrative structures individual organizations do. As such, users risk painful consequences if global, distributed systems rely on central authorities for security. Fortunately, security need not come at the price of centralized control. To prove it, we present SFS, a secure, global, decentralized file system permitting easy cross-administrative realm collaboration. With a simple idea, self-certifying pathnames, SFS lets users escape the evils of centralized control.

1 Introduction

As distributed applications become increasingly prevalent, we must confront a serious danger: Future distributed infrastructures may, as a condition for security, subject their users to rigid, centralized control with stifling consequences. People collaborating across local-area networks usually work within the same organization, limiting the incentive for malicious behavior and making it reasonable for someone trusted to coordinate the collaboration. The same does not hold true across the Internet. Global distributed applications must now handle previously unusual administrative relationships, from file sharing between organizations or homes to running untrusted foreign code on a local machine. If developers of global applications approach these situations with the techniques that worked on local-area networks, they will force everyone to submit to inappropriate, centralized authorities. Fortunately, the problems of centralized control can be avoided. This paper presents SFS, a secure *and* decentralized global file system.

One must ask two fundamental questions about any global file system: Who controls the namespace, and how do clients trust remote machines to serve parts of that namespace? A centrally controlled namespace hinders deployment; it prevents new file servers from coming on-line until

This research was supported by a National Science Foundation NYI award.

they gain approval from the naming authority. On the other hand, clients need cryptographic guarantees on the integrity and secrecy of remote communications; these guarantees require a binding between encryption keys and points in the global namespace. To provide this binding without relying on naming authorities, SFS embeds the equivalent of a server's public key in the pathname for a file. We call such pathnames *self-certifying pathnames*.

By exposing public keys in self-certifying pathnames, SFS manages to permit arbitrary key management policies within a single namespace. SFS recognizes that no single name to key binding process can satisfy all needs. Certification authorities, hierarchical delegation, informal key exchange, the "web of trust," and even hard-coded encryption keys all have their place in different situations. The rest of this paper will show that a secure file system can, in fact, support all of these situations.

The next section motivates the SFS design by outlining the evils of centralized control. The remaining sections show how users can escape these evils through SFS, and explain how SFS allows secure cross-administrative realm collaboration without resorting to any centralized control.

2 The Evils of Centralized Control

Authentication constitutes a major part of any secure network infrastructure. One cannot make use of user accounts or network resources without authenticating them. Whoever controls authentication therefore controls the creation of new resources. Concentrating that power in the hands of a central authority will hinder deployment, stifle innovation, complicate cross-administrative realm collaboration, exclude valid network resources, create single points failure, and generally put everyone at the mercy of the authority.

Several forms of centralized control can plague distributed systems. Most obvious is an overreliance on certification authorities. Any secure global infrastructure must let organizations bind names to public keys, but no system should inherently trust such certification authorities or grant them special privileges or status. A system that can't function without a particular certification authority places users at the mercy of that authority. Should the authority decide to extort high fees, disclaim all liability, exclude people from certain countries, or even mandate key escrow, users will have no choice but to comply.

As an example, SSL [5], the protocol over which secure HTTP runs, relies too heavily on certification authorities. Early versions of the Netscape web client could use only one certification authority, Verisign. One couldn't use Netscape with secure HTTP, even to debug a web server, without paying hundreds of dollars to buy a Verisign certificate for the server's host. To make matters worse, particularly for those developing free web servers, Verisign refused to issue certificates to anyone who didn't run one of several commercial web servers. Now the Netscape browser allows users to choose the certification authorities they trust, but it essentially awards all authorities the same level of trust. This means certification authorities all demand a similarly exacting certification process. The hassle of this process has limited the spread of secure HTTP servers.

Another form of centralized control results from systems that somehow penalize cross-administrative realm use. To facilitate collaboration, users of such systems tend to form inconveniently large administrative realms. The Kerberos [15] authentication system suffers from exactly this problem. Kerberos administrative realms often span many departments within an organization. People responsible for buying and otherwise maintaining machines often cannot create accounts or set up servers without involving a Kerberos administrator. The AFS [7] distributed file system has an even worse problem. Not only is authentication based on Kerberos, but administrators of client machines must also enumerate every single file server the client can talk to. An unprivileged user of an AFS client simply cannot access a file server his administrator does not know about. The rest of the paper will describe a distributed file system that avoids these problems.

3 Self-Certifying Pathnames

Through self-certifying pathnames, SFS provides a global name space over which no authority has control. Furthermore, this namespace permits cryptographic guarantees on communications with file servers.

Each SFS file system has a name of the form "`/sfs/`*Location*`:`*HostID*." *Location* tells an SFS client where to communicate with a server, *HostID* tells it how to communicate securely. *Location* is simply a DNS hostname or IP address. *HostID* is a cryptographic hash of the server's public key and hostname. SFS calculates *HostID* with SHA-1 [4], a one-way function for which there exist no known collisions. *HostID* therefore specifies a unique, verifiable public key with which clients can establish secure communication channels to servers. With this scheme, the name of an SFS file system entirely suffices to certify its server. Thus, *self-certifying pathnames* form an egalitarian namespace that assures every SFS file system a place.

SFS clients need not know about file systems before users access them. When a user references a non-existent directory of the proper format under `/sfs`, an SFS client attempts to contact the machine named by *Location*. If that machine exists, runs SFS, and can prove possession of a private key corresponding to *HostID*, then the client transparently creates the referenced directory in `/sfs/` and mounts the remote file system there. This automatic mounting guarantees everyone the right to create file systems. Given a host on the network, anyone can generate a public key, determine the corresponding *HostID*, run the SFS server software, and immediately reference that server by its *self-certifying pathname* on any client in the world.

Of course, no person will ever want to type a *HostID*—the hexadecimal representation of a 160-bit cryptographic hash. Instead, people can assign human-readable names to mount points through symbolic links. For instance, if Verisign acted as an SFS certification authority, client administrators would likely create symbolic links from `/verisign` to the the mount point of that certification authority—a pathname like `/sfs/sfs.verisign.com:75b4e39a1b58c265f72d-ac35e7f940c6f093cb80`. This file system would in turn contain symbolic links to other SFS mount points, so that, for instance, `/verisign/mit` might point to `/sfs/sfs.mit.edu:0-f69f4a059c62b35f2bdac05feef610af052c42c`.

There is nothing magic about `/verisign`, however; it is just a symbolic link. Thus, SFS supports the use of certification authorities, but neither relies on them for correct operation nor grants them any special privileges or status. In fact, organizations will probably install analogous symbolic links to their own servers in the root directories of their client machines. Individual users can create symbolic links for themselves in their home directories, perhaps after exchanging *HostID*s in digitally signed email.

Anyone with a universe-readable directory can create public links that others can make direct use of. As an example, `/verisign/mit/lcs/dm/sfs-dist` could be a path to the SFS distribution server. The certification chain of such a pathname is always explicit. This is the server called dm/sfs-dist by the server called lcs by the server called mit by the server locally called /verisign. Despite its not being officially certified by Verisign, people can still reach the sfs-dist server and understand the exact meaning of the name.

4 Using SFS

Every SFS server, directory, and file has the same canonical pathname on every machine in the world. Thus, SFS provides a truly global file system. We discuss how users view SFS and what is involved in running an SFS server.

4.1 User view

As described in the previous section, users mount remote SFS file systems simply by referencing self-certifying pathnames, either directly or through symbolic links. This mechanism makes every server accessible from any client machine, while always permitting cryptographic verification of file system contents. A second, user-authentication mechanism is needed to access files that are not universe-readable, however. To accomplish this, SFS file servers have a local file, /etc/sfs_users, analogous to the UNIX password file. sfs_users maps user accounts to public keys. Users must register public keys in that file and prove possession of the corresponding private keys to access protected files on a server.

Once users are set up to access protected files, authentication happens transparently with file access. Every user of an SFS client machine runs an unprivileged authentication agent process with access to private keys. When the user first references a file system, the client software contacts the agent, allowing it to authenticate the user to the remote server. The server sends the agent a challenge encrypted with the user's public key; if the user agent successfully responds to this challenge, then access is allowed.

SFS servers base access control entirely on the identity of the users sending requests, not the machines from which requests come. SFS thus has no notion of administrative realm for client machines. Every client is identically configured, though individual users may configure their agents differently, and will certainly give their agents access to different private keys. This architecture guarantees users access to their files from any client machines they trust enough to use.

4.2 SFS Servers

Setting up an ordinary read-write SFS file server is fairly simple. One need only edit a few configuration files such as sfs_users, generate a public/private key pair for the server, and start the software. The new server will instantly be accessible on any client by its self-certifying pathname. One can notify users of a new server either by distributing its self-certifying pathname or placing a symbolic link to the pathname in an already reachable directory.

Some SFS servers, such as certification authorities, may have very high integrity, availability, and performance needs. To meet these needs, SFS allows servers to prove the contents of read-only file systems with precomputed digital signatures. This makes the amount of cryptographic computation required from read-only servers proportional to the file system's size and rate of change rather than to the number of clients connecting. It also frees read-only servers from the need to keep any on-line copies of their private keys, which in turn allows read-only file systems to be replicated on untrusted machines.

4.3 Example

To illustrate the use of SFS across administrative realms, consider someone at a company collaborating with a research project at a university. Such collaboration usually requires manual file transfers or remote logins over high-latency networks. With SFS, however, the university can simply create an account for this user and put his public key in `sfs_users`. The user can then transparently access both the university's file server and his company's from the same local client machine.

There is no need to reconfigure the client or make any special arrangements to access the university's file system. The user only needs to reference that file system to mount it. He can reference it through a symbolic link in an already reachable directory, or, if need be, he can create his own link to it in a convenient location. Of course, the user will have to be authenticated separately to university and the company. However, his agent will handle this automatically.

5 SFS Implementation

The SFS authentication protocol is the heart of the SFS naming system. We discuss the protocols in detail, followed by a brief overview of the rest of the system.

5.1 Authentication protocol

The SFS authentication protocol was inspired by ssh [17], but with several fixes [1] and modifications. It takes place in two stages: first, authenticating the server to the client, and second, authenticating the user to the server, via the client. Three parties are involved: the SFS client software C, the SFS server software S, and the user's authentication agent A. The only current implementation of the agent software holds all private keys on the same machine as C. Each user can choose to run his own agent implementation, however. In the future, we envision dumb agents that forward authentication requests to smart cards or even to other agents through encrypted login connections, as is currently done with ssh.

The first stage, shown in Figure 1, authenticates the server to the client, with the client initiating the exchange. The goal is to agree on two client-chosen shared session keys, K_{sc} and K_{cs}, to encrypt and protect the integrity of future data communication between client and server. The server sends the client two public keys, PK_s and PK_t: the first, PK_s, is the long-lived key whose hash is in the pathname; the second, PK_t, is a temporary public key which changes every hour. Use of a temporary public key provides forward secrecy: even if SK_s, the long-lived secret key, is compromised, old communication cannot be decrypted without breaking PK_t once SK_t has been destroyed.

The *host ID* in this protocol is the hash obtained from the pathname by the client. The client sends the host ID and hostname to the server in its first message so the same machine can serve multiple file systems without needing multiple IP addresses. The *session ID*, which is a hash of the two session keys, the server's temporary public key, and a server-chosen nonce N_0, is used in future communication to identify the session. The plaintext echoing of N_0 has no effect on host ID $= \{PK_s, \text{hostname}\}_{\text{SHA-1}}$

session ID $= \{K_{cs}, K_{sc}, PK_t, N_0\}_{\text{SHA-1}}$

1. $C \to S$: hostname, host ID
2. $S \to C$: PK_s, PK_t, N_0
3. $C \to S$: $\{\{K_{cs}, K_{sc}\}_{PK_t}\}_{PK_s}$, session ID, N_0

Figure 1. Key exchange and server authentication.

1. $C \to A$: hostname, host ID, session ID
2. $A \to C \to_{K_{cs}} S$: PK_u
3. $S \to_{K_{sc}} C \to A$: $\{N_u\}_{PK_u}$
4. $A \to C \to_{K_{cs}} S$: $\{N_u, \text{hostname}, \text{host ID}, \text{session ID}\}_{\text{SHA-1}}$
5. $S \to_{K_{sc}} C$: authentication number and remote credentials (UIDs, GIDs, etc.)

Figure 2. Authenticating users.

authentication proper, but can be used by the server to filter out certain denial-of-service attacks that would otherwise require significant CPU time due to the expense of public-key decryption [8].

After this initial stage, all traffic from C to S is encrypted and authenticated with K_{cs}, and all traffic from S to C with K_{sc}. The client can be sure it is talking to the server as only the server could have decrypted these session keys. At this point, C can access S with anonymous permissions.

The second stage of the protocol authenticates the user to the server via the client, as shown in Figure 2. This stage is necessary to access protected files. Here, PK_u is the user's public key. The user's agent A has a copy of the user's private key, SK_u. N_u is a nonce, or challenge, chosen by the server. Note that all messages between client and server are encrypted with one of the private session keys K_{sc} and K_{cs}.

The authentication information in the last step of the user authentication is used to establish some reasonable correspondence between local and remote user and group IDs. In the first step, note that the agent has access to both host name and host ID, allowing it to certify the host ID if desired; however, the agent never sees either K_{cs} or K_{sc}, the session keys which are encrypting all client–server communication.

5.2 Implementation details

Both the SFS client and server are implemented at user-level, using NFS. Through a new asynchronous RPC library, fast encryption algorithms, aggressive on-disk client caching, and write-back caching, SFS performs comparably to Sun's unencrypted NFS [13] on standard application workloads, while additionally providing security and lease-based consistency. The specifics of our implementation are beyond the scope of this position paper; the system is described in detail in [10].

6 Related work

SFS's egalitarian namespace is similar in spirit to the SPKI/SDSI[3, 12] security infrastructure. In SPKI/SDSI, principals are public keys, and every principle acts as a certification authority for its own namespace. SFS effectively treats file systems as public keys. However, because file systems inherently represent a namespace, SFS has no need for special certification machinery—ordinary symbolic links already do the job. SDSI specifies a few special roots, such as `Verisign!!`, which designate the same public key in every namespace. SFS can achieve a similar result by convention if clients all install symbolic links to certification authorities in their local root directories. We think such standard links should be determined by common practice, however, and not hard-coded into the file system.

AFS [7, 14] is probably the most successful global file system to date. Like SFS, It provides a clean separation between the local and global namespace by mounting all remote file systems under a single directory, `/afs`. Unlike SFS, however, AFS client machines contain a fixed list of available servers that only a privileged administrator can update. AFS uses Kerberos [15] shared secrets to protect network traffic, and so cannot guarantee the integrity of file systems on which users do not have accounts. Though AFS can be compiled to encrypt network communications to servers on which users have accounts, the commercial binary distributions in widespread use do not offer any secrecy. DFS [9] is a second generation file system based on AFS, in which a centrally maintained database determines all available file systems.

The Truffles service [11] is an extension of the Ficus file system [6] to operate securely across the Internet. Truffles provides fine-grained access control with the interesting property that, policy permitting, a user can export files to any other user in the world, without the need to involve administrators. Unfortunately, the interface for such file sharing is somewhat clunky, and involves exchanging E-mail messages signed and encrypted with PEM. Truffles also relies on centralized, hierarchical certification authorities, naming users with X.500 distinguished names and requiring X.509 certificates for all users and servers.

WebFS [16] implements a network file system on top of the HTTP protocol. Specifically, WebFS uses the HTTP protocol to transfer data between user-level HTTP servers and an in-kernel client file system implementation. This allows the contents of existing URLs to be accessed through the file system. WebFS attempts to provide authentication and security through a protocol layered over HTTP [2]; authentication requires a hierarchy of certification authorities.

7 Conclusions

SFS demonstrates that a secure global file system with a uniform namespace does not need any centralized authority to manage that namespace. Through self-certifying pathnames, SFS permits arbitrary key management policies within a single namespace.

References

[1] Martín Abadi. Explicit communication revisited: Two new examples. *IEEE Transactions on Software Engineering*, SE-23(3):185–186, March 1997.

[2] Eshwar Belani, Alex Thornton, and Min Zhou. Authentication and security in WebFS. from `http://now.cs.berkeley.edu/WebOS/security.ps`, January 1997.

[3] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylönen. SPKI certificate documentation. Work in progress, from `http://www.clark.net/pub/cme/html/spki.html`.

[4] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T, National Tecnical Information Service, Springfield, VA, April 1995.

[5] 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.

[6] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[7] 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.

[8] Phil Karn and William Allen Simpson. The Photuris session key management protocol. Internet draft (draft-simpson-photuris-15), Network Working Group, July 1997. Work in progress.

[9] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer 1990 USENIX*, pages 151–163. USENIX, 1990.

[10] David Mazières. Security and decentralized control in the SFS distributed file system. Master's thesis, Massachusetts Institute of Technology, August 1997.

[11] Peter Reiher, Jr. Thomas Page, Gerald Popek, Jeff Cook, and Stephen Crocker. Truffles — a secure service for widespread file sharing. In *Proceedings of the PSRG Workshop on Network and Distributed System Security*, 1993.

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

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130. USENIX, 1985.

[14] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[15] 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*. USENIX, 1988.

[16] Amin M. Vahdat, Paul C. Eastha, and Thomas E. Anderson. WebFS: A global cache coherent file system. from `http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html`, December 1996.

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