SFS-HTTP: Securing the Web with Self-Certifying URLs

Michael Kaminsky and Eric Banks
MIT Laboratory for Computer Science
`kaminsky@lcs.mit.edu, bank@mit.edu`

Abstract

The current solution to secure Web communication is SSL which relies on certificate authorities for key management, limiting the ability for individuals to independently set up secure Web sites and forcing them to trust a small number of third parties. We propose a new model for Web security—SFS-HTTP—based on SFS. While SFS uses self-certifying pathnames to separate key management from file system security, SFS-HTTP uses self-certifying URLs to separate key management from Web security. Users and Web site maintainers can define their own security models without having to rely on centralized certification authorities. We implement SFS-HTTP using the SFS libraries; users configure host authentication through the SFS agent.

1 Introduction

The current solution to security and authentication on the Web is SSL [7] which, while providing these desired properties, lacks one critical feature: extensible host authentication. Currently, Web security models all depend on (centralized) certificate authorities to validate the authenticity of remote Web servers. SFS [10] solves this problem in the file system arena with self-certifying pathnames, but can we leverage this idea for HTTP?

We propose *self-certifying URLs* as the correct model for security on the Web. Self-certifying URLs, like self-certifying pathnames, completely specify a remote server, including its public key. The central idea in SFS is to remove the burden of authentication from the file system; in SFS-HTTP, we separate the means of authentication from Web servers and browsers themselves. A user can verify hosts in a variety of ways. For example, he may have stored locally the URL or he might retrieve it from a certification authority.

Unfortunately, HTTP [5] does not translate directly into the file system paradigm of reads and writes because of the large and increasing amount of Web data today which is dynamic and must be accessed through `http://` URLs (versus `file://`). SFS-HTTP is a set of components that build on the SFS infrastructure in order to allow users (via their browsers) to access self-certifying URLs. The Web requests are passed through an authenticated, encrypted channel to a remote Web server which returns the results through the same tunnel. Users can leverage off of the SFS agent architecture to dynamically map simple names into full self-certifying URLs. SFS-HTTP strives to combine the extensibility of self-certifying pathnames with the flexibility and ubiquity of the Web.

The remainder of the paper is structured as follows. The next section discusses the overall design of the system. Section 3 describes the details of the SFS-HTTP implementation. The concluding sections discuss the system performance, other related work, and future work.

2 Design

This section begins by describing the basic idea behind the SFS-HTTP system—self-certifying URLs—and outlines the differences between SFS-HTTP and SFS. Next, we give an overview of the system and how the components fit together. Finally, the section concludes by tracing the flow of information when a user initiates a Web request.

2.1 Self-certifying URLs

The anatomy of a self-certifying URL is shown in Figure 1. The key idea that self-certifying URLs borrow from self-certifying pathnames is that the public key of the remote Web server (or file server) is essentially encoded into the pathname. Thus, self-certifying URLs are by themselves completely sufficient to establish a secure connection to a remote machine.

The *Location* portion of the URL indicates the name of the machine hosting the Web server to which the user wants to connect. *Location* can either be a fully-qualified DNS name

---

SFS-HTTP is work in-progress. This paper is available online at
`http://www.pdos.lcs.mit.edu/~kaminsky/sfs-http.ps`

$$\texttt{http}:/\!/\overbrace{\texttt{www.amazon.com}}^{\textit{Location}}:\overbrace{\texttt{we36hsq8xwgam3tcgzkuay8gy7rb3y8h}}^{\textit{HostID}}/\overbrace{\texttt{index.html}}^{\textit{Web page}}$$

Figure 1: A self-certifying URL

or an IP address. The *HostID* portion of the URL contains, in effect, a SHA-1 [6] hash of the the Web site's public key and *Location*.[1] See [10] for the precise computation. The remainder of the self-certifying URL specifies the page on the Web server being requested.

Self-certifying URLs are intended to look and operate like traditional URLs. The only addition is the *HostID* which allows the client to authenticate the remote server. Because self-certifying URLs use HTTP, browsers already know how to encode requests. How the client interprets and uses the *HostID* is discussed below.

Though SFS borrows the notion of a self-certifying identifier from SFS, SFS alone is insufficient to allow a user to surf the Web. If all Web content was static, site administrators could simply export their Web sites as an SFS server and clients could access the site by typing `file://sfs/...` followed by the name of the exported Web server into their browsers. Most sites today, however, rely extensively on dynamic content, cookies, and forms whose implementation is incompatible with the file system model of information access. Dynamic content often requires that the remote server be able to execute a program on-demand which file servers do not support. Futhermore, cookies require the browser to send the server additional information beyond the name of the file. This and other headers are an explicit part of the HTTP protocol but do not have a counterpart in the file system world. Web forms are a common example that combines both of these ideas.

Conversely, the Web lacks symbolic links which are a key component of using SFS, allowing users to conveniently refer to one server from another. The concept of hyperlinks on Web pages is similar except that a user cannot refer directly to a hyperlink, rather only to the Web page which contains it. In SFS, a user could potentially access several SFS servers with a single self-certifying pathname; in SFS-HTTP, a self-certifying URL can only *explicitly* name a single server.

2.2 System Overview

The SFS-HTTP system consists primarily of two components: a *server daemon* and a *client daemon*. Like SFS, our 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 show in Figure 2.

Boxed components are part of SFS-HTTP, and unboxed components represent pre-existing software. The SFS-HTTP system was designed such that existing software could be used as-is without patches, possibly requiring only minor reconfiguration. For example, during testing we used recent versions of Netscape Communicator and the Apache Web server.

Each half of the diagram represents a single machine. The *server daemon* and Web server run on the same machine as do the *client daemon*, Web browser and SFS agent; we assume that all connections between processes on the same machine are secure. The two machines (specifically the two daemons) communicate through an authenticated, encrypted connection. Furthermore, the *server daemon* speaks to the Web server by acting as a Web client and the *client daemon* speaks to the Web browser by acting as a Web proxy.

The SFS agent is a process that runs on behalf of the user which can asynchronously answer queries for him; users configure their agents ahead of time. In SFS, the agent is primarily responsible for authenticating users to remote file systems, though it also supports features such as key revocation. SFS-HTTP in particular uses the agent's ability to do extensible host authentication (discussed below).

2.3 Flow of Information

There are several scenarios that can occur when a user wants to surf the Web through the SFS-HTTP system. In the first case, the user simply types into his browser a fully specified self-certifying URL. The URL is passed to the *client daemon* which extracts the *HostID* from the URL and initiates a secure connection to the remote *server daemon*. After removing the *HostID* from the HTTP request ("GET", "POST", etc.), the *client daemon* passes it to the *server daemon*, which simply forwards it to the local Web server. The server generates a reponse HTML [2] page (or image, Java applet, etc.) and returns it to the browser over the reverse path. Thus, the request and response never travel over the network unencrypted.

*HostID*s are 32 characters long, making them difficult for users both to remember and to type into their browsers. Because even traditional URLs are often long and hard to remember, users often bookmark pages in their browsers. Sim-

[1] This public key is distinct from any SSL certificate that the site may have.

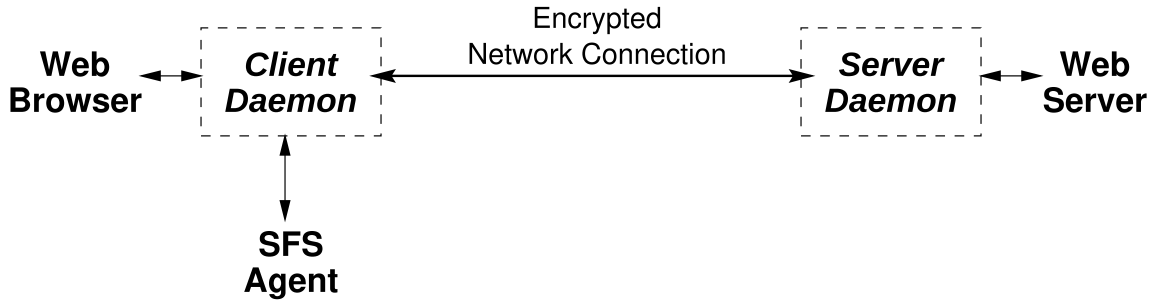

Figure 2: The SFS-HTTP system

ilarly, SFS-HTTP users can also bookmark pages accessed through self-certifying URLs. The common case when surfing the Web, though, is for users to click on links rather than type in URLs or use bookmarks. Link targets are passed to the proxy in the same way as typed URLs, so that Web page authors can encode self-certifying URLs directly into their HTML to provide secure hyperlinks to remote Web servers.

Often users will want to access a Web site for which they do not explicitly have a *HostID* (as in the above case). In this scenario, the *client daemon* can query the requesting user's agent to resolve a traditional URL into a self-certifying one. For example, if the *client daemon* sees `http://www.cnn.com`, it will create a connection to the agent and ask whether it knows the *HostID* corresponding to "`www.cnn.com`." If the agent returns a mapping, the *client daemon* uses it to access the remote Web server as described above. If the agent fails to return a *HostID*, the *client daemon* falls back to an insecure connection, acting as a traditional Web proxy.

3 Implementation

This section describes the details of the SFS-HTTP implementation. We consider each of the major components in turn and how they communicate with each other.

3.1 Client-Server Communication

All of the SFS-HTTP components are written using asynchronous RPC, network, and data structures libraries originally developed for the SFS project. The *server daemon* and the *client daemon* communicate with each other using SUN RPC over an encrypted and MACed transport (see below). The *client daemon* caches the encrypted connections to avoid repeatedly incurring the cost of the public key operations required to authenticate the connection. The cache is indexed by both *Location* and *HostID*, so that two users only share the authenicated connection if they trust the same *HostID* for a server.

The cryptography used in SFS-HTTP is the same as that used in SFS; the details including cryptographic hardness assumptions are found in [10]. Essentially, key negotiation is as follows. The client verifies the server's public key with the *HostID* and then encrypts and sends to the server its (short-lived) public key and two random session key halves. The server responds by encrypting to the client the other halves of the two session keys (also random). Each direction's session key is computed by combining the key halves. Once the session keys are established, the two sides can turn on encryption and integrity. The SFS cryptographic libraries use an ARC4 [8] stream cipher for encryption and a SHA-1 based Message Authentication Code (MAC) for integrity. SFS uses the Rabin public key cryptosystem [12] for encryption and signing.

3.2 Server Daemon

The *server daemon* runs on the same machine as the Web server and typically makes requests to the server through TCP connections to the *localhost* interface. The *server daemon*, though, can be configured to contact any Web server if the network connection to it is considered trusted. Because the RPC libraries are asynchronous, the *server daemon* can receive multiple concurrent requests from the *client daemon* and pass them in parallel to the Web server for efficiency.

A side effect of forwarding all requests to the Web server from the the *server daemon* is that domain name and IP based host access is rendered ineffective. Web servers often support access restrictions based on the source location of the TCP connection. Such means of authentication are susceptible to DNS spoofing attacks, but are nonetheless a popular way to support site licenses. In SFS-HTTP, all secure connections to the Web server come from the local machine (the *server daemon*), effectively giving all machines unrestricted access to host-restricted pages.

3.3 Client Daemon

The *client daemon*, Web browser, and user's SFS agent typically run on the same machine. The *client daemon* and agent communicate via RPC as described above. The *client daemon* acts as a Web proxy and the browser is configured to send HTTP requests to it by opening TCP connections to a port on the local machine.

The current implementation of the *client daemon* requires that every user has his own instance of the proxy. This implementation artifact results from the fact the Web requests are essentially anonymous. When the proxy receives a request, it has no way of knowing which agent to contact. Our solution was to have each user run his own copy of the daemon on a port of his choosing. The user would configure his Web browser to contact his *client daemon* which would then communicate with his agent (by contacting the agent corresponding to the user and group under which it is running).

3.4 User Agent

Every user in the SFS-HTTP system optionally runs an SFS agent to dynamically map traditional URLs to self-certifying URLs by providing *HostID*s for hostnames (*Location*s). The agent and the *client daemon* communicate using RPC through a UNIX domain socket.

The SFS agent allows the user to define arbitrary programs to perform this name-to-*HostID* mapping. If the user provides more than one program, he can specify both an order for them to be run and a regular expression pattern for each that must match in order for the program to execute. Thus, the user has complete control over the method by which remote Web sites are authenticated.

3.5 User Authentication

Typically, Web servers do not authenticate incoming connections; some servers, however, support host-based authentication while others offer limited user-based authentication. Web sites that support user-based authentication do so with primarily two mechanisms: personal certificates and passwords.

Personal certificates are a feature of SSL and are not unlike site certificates for Web servers. Because personal certificates are a part of SSL and not standard HTTP, they are not directly applicable to SFS-HTTP. Clients that use SSL already have the privacy, integrity, and authentication that it provides, so using personal certificates in this situation is safe.

Password authentication, however, is typical done over unencrypted network connections and the password is sent in the clear. In SFS-HTTP, standard HTTP connections are encrypted, so password authentication transparently becomes secure.

4 Performance

We evaluate the performance of the SFS-HTTP software by fetching 1,261,862 bytes in 108 files from a Web server in four ways: directly (without SFS-HTTP), with SFS-HTTP through a self-certifying URL, with SFS-HTTP where the agent provides the *HostID*, and with SFS-HTTP unencrypted. In each case, all objects are downloaded the same way; that is, we do not mix self-certifying URLs with traditional ones.

All of the sofware is running on a Linux 2.2.10. The Web server is Apache 1.3.9 compiled out-of-the-box (the set of pages fetched is actually Apache's default `htdocs` directory). The client is wget 1.5.3, a command line utility that can recursively download Web pages via HTTP.[2]

The client slide of the connection, which runs the SFS agent, the *client daemon*, and wget is a Pentium II, 200 MHz with 128 MB of memory. The server is a Pentium II, 300 MHz with 128 MB of memory, and it runs the Apache Web server and the *server daemon*. All of the experiments were conducted when the machines were under a typical, average load.

Mode	Throughput (KB/sec)
Direct	566
SFS-HTTP (encrypted, no agent)	297
SFS-HTTP (encrypted, with agent)	172
SFS-HTTP (unencrypted)	270

Figure 3: Throughput for fetching approximately 1.2 MB in 108 files.

Figure 3 shows the throughput for each of the four experiments described above. Clearly, retrieving the data without involving SFS-HTTP provides the highest throughput because the connections did not need to go through a proxy.

The second case demonstrates a basic encrypted connection through SFS-HTTP and a self-certifying URL that does not involve user's agent. This case was almost twice as slow as connecting directly to the Web server. Here, all HTTP requests go through the *client daemon* proxy which we believe accounts for almost all of our slowdown.

In the third case, the connection is also encrypted, but we need to query the user's agent for the *HostID*. There are two main sources of slowdown in this case: the *client daemon* neither caches the connection to the user's agent nor the mapping between *Location* and *HostID*. Futhermore, on each query, the agent needs to `exec` an external program to

[2]We use a slightly modified version of wget to fetch pages with self-certifying URLs because wget's argument processing tries to interpret the `HostID` as a port number. Netscape, conversely, simply passes the entire URL to the proxy.

provide the *HostID* (in our case, a one line shell script that calls `echo`).

In the fourth case, the *client daemon* still needs to contact the user's agent, but the agent fails to return a *HostID*. This benchmark is slightly slower than the encrypted connection using a self-certifying URL because the *client daemon* still needs to establish a connection to the agent for each HTTP request. This case is faster than the third case because the *client daemon* does not need to open a secure connection to the *server daemon*; rather, it sends requests directly to the Web server.

5 Related Work

In this section, we first address SSL which is the current standard for security on the Web. Then, we briefly address another potential candidates for securing HTTP connections, IPsec. Finally, we consider three systems which seek to combine HTTP with file systems for enhanced functionality and/or performance.

Currently security on the Web is based on SSL [7]. SSL relies on digital certificates that are signed by trusted certification authorities who guarantee the veracity of the server. CA public keys are built into Web browsers, essentially predefining a limited set of CAs. Users can add new CAs and site certificates to their browsers, but have no general-purpose, configurable mechanism to authenticate servers. Moreover, Web sites that want security need to purchase certificates from a CA, a process which is often expensive and time-consuming.

IPsec [9] offers another possible option for securing Web communications. Unfortunately, IPsec has no deployed global key management proposal. Though the privacy and integrity features of IPsec could be used to provide some degree of host-to-host security, the SFS architecture already provides these features along with its improved host authentication.

The Gecko NFS Web Proxy [1] 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. Gecko's overall design is similiar to SFS-HTTP in its tunnelling of Web requests over a different protocol, NFS. NFS, however, does not offer the security and name-mapping of SFS.

WebNFS [4], 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 [11] 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: arbitrary host authentication.

6 Future Work

This section suggests possible improvements and extensions to the SFS-HTTP system. Currently, each user that wants to use his agent to map *Location*s to *HostID*s must run his own copy of the *client daemon*. Future implementations of SFS-HTTP may provide a way for multiple users to make use of one proxy. In that scenario serveral users accessing the same Web server will be able to share the encrypted channel, saving connection startup time. Futhermore, since the *client daemon* can already handle multiple connections, running duplicate copies needlessly consumes system resources.

The current version of the *client daemon* does not cache any data. With each user running his own version of the daemon, a typical cache would not necessarily add functionality to the system over the Web browser's builtin cache. However, if multiple users shared a single *client daemon*, users could benefit from a shared cache.

As discussed in Section 3.2, all requests sent to the Web server come from the local machine, disabling any host-based access control in the Web server. One solution is to build host-based access control into the *server daemon* itself. When the deamon receives a connection from the *client deamon*, it can decide to accept or reject the connection based on the originating host and the page requested. This solution does not provide any added functionality over doing the checking in the Web server, yet it offers sites the ability to use this pre-existing access mode.

Finally, SFS-HTTP does not currently distinguish for the user between Web pages that are accessed securely and those that are accessed insecurely (when the user's agent cannot provide a *HostID*). Future versions of the *client daemon* could modify the retrieved Web page to indicate that it was fetched over a secure connection. A simple graphic in the corner of a page could indicate its security status. Moreover, the *client daemon* could insert HTML or XML [3] tags to flag hyperlinks with self-certifying URLs, so that users know that clicking the link will take them to a secure Web site (provided of course they trust the page containing the link).

7 Summary

SFS-HTTP attempts to provide security on the Web using self-certifying URLs. Self-certifying URLs alone are sufficient to authenticate a remote machine and establish a secure connection to it because they encode a cryptographic hash of the server's public key. Thus, if a user can name a Web page, he can securely retrieve it.

Users have arbitrary control over the host authentication in SFS-HTTP through their SFS agent. In the absence of a self-certifying URL, the client software will ask the querying user's agent to provide a *HostID* for the specified host. The user's agent is free to execute arbitrary programs to complete the mapping.

The SFS-HTTP system is built on top of SFS libraries and strives to integrate transparently with existing software. One can use standard Web browsers and servers without modification and with little to no extra configuration. SFS-HTTP offers incremental deployment, reverting to an insecure Web proxy if no *HostID* for the remote server is available.

Thus, SFS-HTTP offers a new model for Web security in which server authentication is no longer fixed and centralized. Users can freely and easily set up and access secure Web servers based on their individual needs and environments.

References

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

[2] T. Berners-Lee and D. Connolly. Hypertext Markup Language—2.0. RFC 1866, Network Working Group, November 1995.

[3] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 Specification. W3C Recommendation REC-xml-19980210, W3C, February 1998.

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

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol—HTTP/1.1. RFC 2616, Network Working Group, June 1999.

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

[7] 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.

[8] Kalle Kaukonen and Rodney Thayer. A stream cipher encryption algorithm "arcfour". Internet draft (draft-kaukonen-cipher-arcfour-03), Network Working Group, July 1999. Work in progress.

[9] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Network Working Group, November 1998.

[10] 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 (SOSP '99)*, Kiawah Island, South Carolina, December 1999.

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

[12] Hugh C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.