

Presented at the Atlanta Linux Showcase
October 12–16, 1999, Atlanta, Georgia

A Universal Access, Smart-Card-Based, Secure File System

James Hughes
Storage Technology Corp.
jim@network.com

Chris Feist, Steve Hawkinson, Jeff Perrault, Matthew O’Keefe
University of Minnesota
{feist, hawkinson, perrault, okeefe}@lcse.umn.edu

David Corcoran
Purdue University
corcordt@cs.purdue.edu

October 12, 1999

Abstract

The SFS provides transparent, end-to-end encryption [1] support to users accessing files across the Internet on HTTP or FTP servers. In this paper we describe a SFS architecture, the current implementation, and the future directions for our work.

1 Introduction

In this paper we describe the Secure File System (SFS) which provides end-to-end encryption and key management support to users accessing file across the Internet on HTTP or FTP servers. The SFS implementation is only one instantiation of the Secure File System architecture, which consists of a file access layer, a smart card module, and the SFS Client and Group Server. In the rest of the paper we describe previous work in encrypting file systems, outline the Secure File System architecture, the current SFS implementation, and the SFS Client and Group Server. Finally, we discuss future directions for our efforts.

2 Previous Work

In this section we review other research or product efforts in encryption for shared, persistent storage.

2.1 Volume Encryptors

Several disk encryption systems are available which use a device driver layer (called a filter driver in Windows) to encrypt and decrypt data as it is sent to and received from a disk. These systems include PGP Disk from Network Associates, the Secure File System by Peter Gutmann at the University of Auckland [2], and TorDisk by Alexander Tormasov [3]. Though convenient for protecting whole disk volumes, volume encryptors do not allow access controls on fine-grain objects like directories and files.

2.2 File Encryptors

To achieve end-to-end encryption the encryption can be done in the presentation layer or application layer. Brute force encryption at the application layer requires that all applications that need to work with encrypted files be rewritten to include support for encryption. This is clearly unacceptable for storage systems.

Files may be encrypted on a per-file basis using tool like PGP, developed by Phil Zimmerman [4]. Though useful for the short-term encryption requirements of a single user, it is not generally useful managing shared information stored for the long term because it is based on identity of the consumer. If the consumer is not directly known or changes (as is the case in many organizations) then any PGP-encrypted file must be re-encrypted.

2.3 File System Encryptors

The Cryptographic File System (CFS) [5] was developed by Matt Blaze at AT&T. CFS allows users to encrypt files on a per-directory basis using a single key. An NFS layer implemented the encryption, decryption, and key management locally on a trusted client: files were encrypted while in transit between the trusted client the untrusted network and server. In the original CFS implementation, sharing files require sharing the keys. The key distribution problem makes this difficult. Blaze proposed a key management scheme that helps address these issues in [6].

The Transparent Cryptographic File System (TCFS) [7] was developed at the University of Salerno, Italy. It improves on the CFS design by removing the NFS client encryption layer but still has a limited key management scheme.

The Satan File System was developed at Carnegie-Mellon University. The implementation employs C library modifications that read a file into main memory, decrypts the data and then delivers it to the application. The basic idea is to link the applications against a set of libraries that provide encrypted versions of standard library calls. This solves the problem of rewriting the applications, but the applications still need to be recompiled or at least relinked. Also every program will have to have an unencrypted *and* encrypted version for working with encrypted files or unencrypted files.

IBM's Distributed File System (originally known as AFS and then commercialized by Transarc, a company acquired by IBM) assumes that security is a network problem. Many systems expect users and administrators to assume that their implementation is trusted and that network security measures can be effectively implemented independently of other security measures. This also assumes that the security measures for the backups, HSM, file caches and administrators themselves are flawless.

The Networked Attached Secure Disks (NASD) project [8] at CMU created security for accessing files on storage devices (NASD disk drives) attached directly to a network. Network keys are generated and distributed to the users. The entire system is based on the file system controller being trusted. NASD also has a single symmetric master key between each file system and disk drive.

Microsoft's Encrypted File System (EFS) is available in Microsoft NT 5.0. It can encrypt and decrypt on a per-file and per-directory basis. This system specifies that there are backup "persona's" that have access to all data in the clear and thus administrative data protection.

Other related work includes [5], [9], [6], [10], [11], [12].

3 The Secure File System (SFS) Architecture

3.1 Overview

We are developing the Secure File System as OS- and application-independent security middleware. Our approach is to develop techniques which cryptographically protect information at the source and unprotect data at the intended consumer. The identity of who "needs-to-know" is kept as an attribute of the file.

Cryptographically sealing a file or directory allows the information infrastructure security to be relaxed: routine system administration, backup and restore, archiving need not be trusted nor even physically secure since the data is protected at all times.

Our approach also allows decentralized access rights, where small groups of people can define among themselves (without the help of an untrusted system administrator) who is allowed to look at certain information and what set of rules must be followed to do so. This is critical because secure information sharing is required for effective cross-organizational decision making.

We define the following terms:

- Information Producer — has (by definition) the data in the clear and has the authority to define who can see their data
- Information Consumer — needs the data in the clear (by definition) and has the (undeniable) ability to pass the information on
- Group Agent — determines group membership (who needs to know), and provides a non-repudiable audit trail.

Information needs to be communicated between between the producer and the consumer: except for the group agent, no other entity is allowed to "see" the information in the clear. This means the information is cryptographically-sealed while stored on disk or tape and while in transit on the network.

Information ownership is based on organizations and their mandate: producers and consumers can dynamically define and create their own groups for information sharing.

Trusted systems are difficult. Trusted networks of file systems are virtually impossible. This system is focusing instead on creating information security middleware to protect information from the producer to the consumer so that the number of entities that must trusted is minimized.

- True "End-to-End" — other systems encrypt a link or to a web server, but SFS encrypts from the information producer to information consumer.
- Distributed Membership — defined by the user departments, not by system administrators.
- Stored Encrypted — backups, archives, networks, etc. are all protected.
- Audit Trails — non-repudiable audit trails of requests to access files are maintained by the Group Agent.
- Flexibly Tag Information — by its purpose(s), not just the producer's identity: Board of Directors, Executive Policy makers, Project X, etc.
- Cryptographically Enforced — strong encryption used for the information and smart cards and PKI used to secure and manage keys.

With our approach, the producer defines "Access Control Meta-data". The access control list is a specification of purpose of the information which defines the need to know. This specification of purpose is defined through an access formula which defines the actions of the consumer to agents of the producer (or others) to unlock the information produced by the consumer.

By solving the formula, the key to the file is revealed (potentially only) to the consumer. The formula can be any combination of users, groups, and projects. Any combination of logical AND and OR operations as well as N-person control can be used. N-person control for backups and sensitive information requires that multiple identities or organizational membership. An example of the meta-data format is shown in figure 4.1.

3.2 Real-world examples

3.2.1 Private projects within an organization

Every organization has projects that are more sensitive than others. These projects include people that are not typical IS support personnel. These projects include corporate reduction projects, employee medical information, mergers and acquisitions, Board of directors, executive compensation, etc. This sensitive information is typically handled by locking up paper copies. Long term centralized storage of this kind of information is both dangerous to a corporation's viability and potentially can potentially result in SEC, civil and criminal penalties.

This kind of sensitive information should not be stored on centralized file servers because of the large number of

people outside the project membership required to manage the networks, servers, backups, etc. It is typical that 5 percent of every organization is Information Systems support personnel. While IS support is a necessity, the fact that a sensitive 10 person project within a 10,000 person company has 500 people to support them is a significant vulnerability.

While each support person does not have universal access or knowledge, each of the professional systems administrators (SA) have access to a portion of the process. The email SA can surf the email, the desktop SA has remote access to the desktop machine, the file server SA can surf the files, the network SA can sniff the networks, the backup SA can surf the backups and even the person that works in the warehouse that contains the backup tapes can either access the tapes or give others access to this sensitive information.

This system is designed to protect the data from one project member's desk to the other. Management of the project membership is not an IS job, a simple to operate, tamper resistant "group server" allows group membership to be directly managed by a project member in a decentralized manner.

Once data is protected in this manner, malicious, disgruntled or just curious employees are no longer even tempted to access this information because the information is protected.

A side issue to note is that if a hacker gains access to an internal network or even gains access to a SA account, the protected information will not be vulnerable.

3.2.2 Outsourced Information storage

While IS employees are a potential risk, the tendency of companies to outsource their IS professionals as well as the IS equipment to outside companies either at their own location or the other companies location. This increases the risks to sensitive information. The same outsourcing company can be have a competitor as a customer and the potential for information cross pollination is there.

By protecting the information while it is still under control of the project members, it does not matter who manages the storage.

3.2.3 Outsourced Intranet Servers

Most companies have many more internal Intranet web servers than Internet servers. While today it is possible to outsource the management of the Internet web server, it is not possible to securely outsource the intranet web sites

without protecting the data in a way that the administrators of the hosting web site are not a vulnerability.

This technology can allow all of a companies sensitive intranet information be stored and managed by companies like AOL.

3.2.4 Sharing sensitive information

Many organizations have partnerships with other organizations and have a necessity to share sensitive information with that organization. Today, this is managed by allowing one organization to have access to the other, maybe with leased lines behind the firewalls. This results in a significant amount of administrative overhead for networks and foreign access to file servers and significantly higher vulnerability. SFS allows sensitive information to be protected and then published outside the firewalls so that the other organization can access the information that they need without having unfettered access to the entire organization.

This sharing can be anonymous to the producing information or can be with audit trails back to the producing organization.

Another aspect of sharing sensitive information is the ability of project members to be able to mandate multi person control of information access and data recovery. Other systems lack this fundamental feature.

3.3 Group Server

The Group Server is the only trusted entity of the SFS architecture. All file keys are encrypted to the Group Server's public key and stored in a header with every encrypted file. This header contains an access control list which is forwarded to the Group Server when a file is accessed. The Group Server is then able to determine if the user has access to the file. If so, the file key is returned to the user and the user can access the data.

The Group Server is also the single administrative point of the SFS Architecture because it controls access to all encrypted files. This allows the addition and removal of persons from groups fairly simple and allows for an extensive audit trail detailing what file was accessed, who accessed it, what time it was accessed, and where it was accessed from.

3.4 XML Access Control List

The XML access control list allows the user to explicitly define exactly who has access to their data. An example is shown in Figure 4.1. This example starts out with two fields defining the owner and their owning group. This is then followed by several blocks of XML which define who has access to the data. The statement `<any m="1">` means that only one of the enclosed blocks is necessary to access the file. This allows users to specify access to one person (me@asdf.com) or a group and project (design). The groups can be specified in series, this requires that the user be a member of both groups. It is also possible to keep half of the the key with one group and half with another, this prevents the owner of either group from being able to access your data. With the access control list it is also possible to specify multi-person access, `<any m="3">` specifies that only three of the six enclosed blocks are required for access to the file.

```
<?XML VERSION="1.0"?>
<FileData>
  <!-- What authority am I doing this for -->
  <OwningGroup id="myproject"/>
  <!-- I write this. -->
  <Writer id="me@asdf.com"/>
  <ACL>
    <!-- Any one of the following options accesses the data. -->
    <any m="1">
      <!-- Or I can read my own data, or else -->
      <individual id="me@asdf.com" >
        <key data="12341234 12341234 12341234 12431234" />
      </individual>
      <!-- Or access through the following group -->
      <group id="group3" project="Celeron">
        <key data="12341234 12341234 12341234 12431234" />
      </group>
      <!-- Access through the following groups in series -->
      <group id="group1" project="Design">
        <group id="group2" project="Pentium" >
          <key data="12341234 12341234 12341234 12431234" />
        </group>
      </group>
      <!-- Or get half the keys from the following locations -->
      <any m="2">
        <group id="group4" project="Xeon">
          <key data="12341234 12341234 12341234 12431234" />
        </group>
        <group id="group5" project="Merced">
          <key data="12341234 12341234 12341234 12431234" />
        </group>
      </any>
      <!-- Escrow -->
      <any m="3">
        <individual id="Escrow1" > <key data="12341234 12341234 12341234
12431234" /> </>
        <individual id="Escrow2" > <key data="12341234 12341234 12341234
12431234" /> </>
        <individual id="Escrow3" > <key data="12341234 12341234 12341234
12431234" /> </>
        <individual id="Escrow4" > <key data="12341234 12341234 12341234
12431234" /> </>
        <individual id="Escrow5" > <key data="12341234 12341234 12341234
12431234" /> </>
      </any>
    </>
  </>
</>
```

Figure 4.1

4 An Implementation of the SFS Architecture

Figure 4.2 shows the current Secure File System implementation. It employs a Linux client and allows universal access to any Web-accessible file available on an FTP or HTTP server. The SFS implementation acts as a transparent middleware layer so that applications need not be changed: SFS accesses appear just the same as local file system accesses.

The SFS implementation we describe in this paper is only one instantiation of the SFS architecture. We believe SFS Client and Group Server described here can be plugged into other distributed and local file systems with little difficulty [10], [13].

In the next sections, we describe our initial SFS implementation, including the universal file access layer known as UFO, the smart card authentication and key management module, and the SFS Client and Group Server.

4.1 UFO

UFO is a file system extension that provides transparent access to files on remote FTP and HTTP file servers [9]. Using UFO you can access and manage files on remote sites using familiar UNIX file utilities like `ls`, `cat`, `grep`, and `vi`.

UFO controls subject processes like a debugger. Using the native debugging interface it is possible to intercept system calls before and after they enter the kernel. Once the calls are intercepted by a controlling process, the controlling process has the opportunity to modify the arguments and return values of the call.

Unlike a debugger, UFO modifies and services some of the system calls made by the controlled application. During the servicing of the system calls extra OS functionality can be added. This provides a convenient way of adding extra features to the operating system without the normal drawbacks of doing kernel development, recompiling large parts of the operating system, recompiling applications, or any other system administration that accompanies an operating system upgrade.

UFO's original purpose was to make remote FTP and HTTP file services look like they were local filesystems. When an open system call was made on a file, if the file was located on a remote filesystem, UFO would satisfy the request by downloading the file into a local file cache and then change the open request to an open request for the downloaded copy in the cache. All further operations

on the file were directed to the cached copy of the file. The file stays in the cache until a close was performed on it.

An encryption layer was easily added into this scheme. First, UFO downloads an encrypted file into its cache; then the SFS decrypt file routine is called on the downloaded file (assuming the user is authenticated and is allowed to access the file). Once the file is decrypted, the open call proceeds using the decrypted version of the file. All further operations are directed to the decrypted version of the file in the local cache. When the file is closed and is about to be written out to the FTP server, a call is made to the SFS encrypt file routine. Once the file is encrypted, the close routine continues, writing out the new encrypted version of the file to the FTP server.

4.1.1 UFO Advantages

Encryption can be added to any layer in the data stream. The question of where to put the encryption depends on many factors. We will look at some of the factors that influenced our decision to use UFO.

Key questions include: What is the granularity of the encryption? Can everything on a disk be encrypted with the same key? Can everything in a certain directory be encrypted with the same key? Or should every file be encrypted with a separate key?

We believe that in many situations there will be sets of files that should be shared with one group of information consumers, and other sets of files that should be shared with other groups of information consumers. Where these files are stored should not affect the group of information consumers that are allowed to access the information. It should be possible to specify cryptographically-enforced permissions on a per-file basis. If everything on a disk drive is encrypted with the same key, in order to let someone read one of your encrypted files the key to the entire hard drive must be revealed to them. The same is true for directories. In order to let someone read an encrypted file in a directory with all of the files encrypted under the same key; the key for the entire directory must be revealed. SFS uses separate keys for each file to provide fine-grain control of file accesses.

Should the encryption be end-to-end or link-by-link? In communications you can encrypt a message and then send the encrypted message through an unsecured link. Or you can encrypt the link and not worry about encrypting the message. In communications, link-by-link encryption is straightforward and highly effective between two nodes. However, there are drawbacks: (1) all links have to be encrypted to ensure security, (2) any nodes along the channel

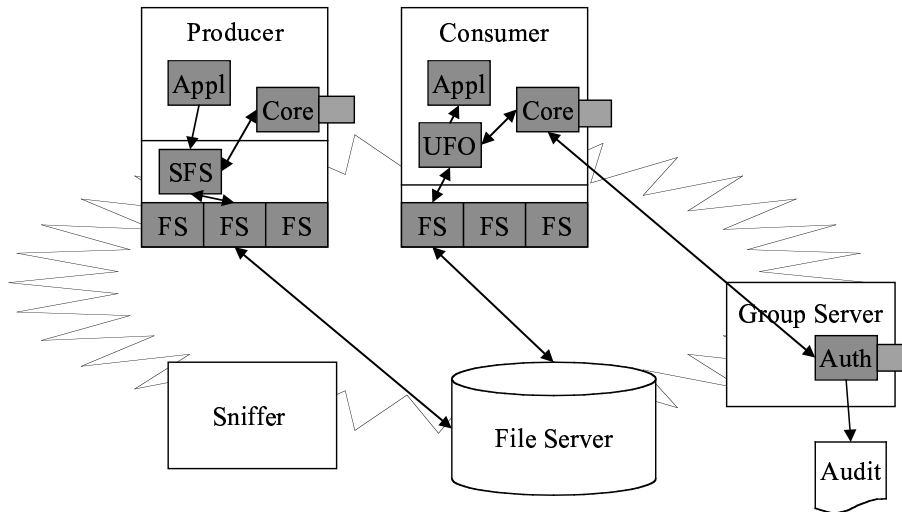


Figure 4.2

get access to all of the information, and (3) many processing cycles are wasted encrypting information that does not need to be encrypted.

In some ways, storage can be considered as a high-latency communications endpoint. The drawbacks to link-by-link encryption are essentially the same in a storage system as in a communications system. In link-by-link encryption all nodes and links in the data path need to be secured. Physically securing all disks and tapes (for backups) is costly and error-prone.

SFS is designed to provide end-to-end encryption. The information is encrypted at the information producer and remains encrypted until it reaches the information consumer. This increases security substantially since there is no need to rely on the storage subsystems to provide security.

The major disadvantage of end-to-end encryption is that it allows traffic analysis[14]. File sizes and the ownership of files are not hidden by SFS. SFS does encrypt filenames, but the filenames are still open to some amount of analysis. SFS treats directories like files. A directory has a unique key and an access control list like a file. When an information consumer is given access to read a directory, they can read the names of all the files in that directory. This allows them to see the names of files they may not have permission to access. Read access to the directory also gives them permission to read the access control lists of all of the files in that directory. This provides information on which groups of people can access certain files in that directory.

In storage systems the presentation layer is part of the filesystem. Adding the encryption to individual file sys-

tems eliminates the duplication of this functionality in every program. But each file system to use this functionality will have to be modified to support the encryption. UFO represents is a middleware layer between the applications and the original file system.

UFO catches system calls after they leave the application, but before they reach the file system. Once the system calls are caught UFO can and does modify them. This allows UFO to act as the translator between the application and the filesystem.

4.1.2 Improvements to the UFO Implementation

The current implementation of UFO [9] was adequate to provide a proof of concept for our SFS architecture (and we are grateful to the original UFO developers for making their code available to us), but it needs improvement to be useful in production versions of SFS.

For example, there are some system calls that should be handled by UFO, but are not. This causes minor errors in some applications running on top of UFO. Also, the read and write system calls are not caught, but must be to support partial encryption and decryption of files.

In the original UFO implementation, read and write system calls did not need to be modified. Once the open call was modified, the file descriptor returned from the open call referred to the cached copy of the file so all reads and writes were already directed at the cached copy of the file. Therefore, there was no reason to catch and modify the read and write system calls.

The current implementation requires that the entire file be encrypted or decrypted. This gives rise to extra copies

of the file in the cache, one for the encrypted version and another for the decrypted version. With large files this becomes a problem. Also, decrypting only the portion of the file that is being read takes less time than decrypting the entire file. In UFO the read and write system calls need to be caught and modified to support partial encryption and decryption of files.

4.1.3 Future improvements to UFO

Internally, UFO represents files as *unodes*, these are analogous to inodes in a traditional file system. These unodes are identified by their filename, unlike traditional filesystems that identify inodes with numbers. Since SFS will encrypt filenames as well as file data, every file has two names, the encrypted name and the cleartext name. Using the filename to identify the files internally can get confusing. This can be simplified by following the traditional approach to identifying inodes with inode numbers.

The original authors divided UFO up into the catcher module, the filesystems module, and the protocols module [9]. A fourth module (running inside the kernel) should be added that takes care of the kernel-like functions of UFO, such as path name resolution, managing subject processes, and keeping track of mounted filesystems. In the current implementation, this functionality is mostly in the OS-dependent catcher module but this separate kernel module would allow much of it to be made to be system-independent.

In the original version of UFO there were three types of file systems: local, FTP, and HTTP. Adding encrypted versions of each of these adds three more file systems which results in code duplication. Using the vfs layer, the interface between the UFO kernel and the individual filesystems can be generalized and file systems can be implemented as separate libraries that can be compiled in separately if desired.

Wrapfs [13] is an implementation of stackable file system layer concept in Linux. It runs as a kernel loadable module and can sit on top of one or more filesystems. Each layer in a vfs (virtual file system) stack has the opportunity to manipulate the data and then pass the data to the next layer in the stack. The top layer in the stack talks to the application and the bottom layer in the stack talks to the storage device. A vfs stack is just a stack of translation layers. SFS could be implemented as a layer in the vfs stack.

UFO runs completely in user space, which allows anyone to install and use it without root privileges. While wrapfs has a performance advantage over UFO, because it runs in kernel space and does not have to context switch

or use the slow debugging interface. We would like to implement SFS using both approaches and have chosen to do UFO first because it is easier to implement a user level program and UFO is easier to install than wrapfs.

SFS could also be integrated into the Intermezzo distributed file system [10]. Intermezzo has much more sophisticated sharing and client caching semantics than UFO, but at present has no mechanisms to provide secure access to encrypted files.

4.2 Smart Cards

In this section we describe Linux smart card technology and how we are applying it the problem of authentication and key management in the SFS implementation.

A smart card [11] looks like a credit card and can even be used as one, but it has a major advantage over credit cards magnetic strips: its own processor. On a Schlumberger Cyberflex Access smart card there is 64 kilobytes (kB) of total memory. Most of this memory is used for the smart card OS and the Java Virtual Machine. There is 16 kB of EEPROM left over for the user. This memory is non-volatile. There is also a small amount of RAM (1 kB). Smart cards have no batteries: the processor is activated by placing the smart card in a card reader which can be attached to a PC, an ATM, or whatever.

The smart card can transfer information at 55,800 bits per second to and from the card reader. The smart card processor allows a higher level of security than a simple magnetic strip (but not perfect security – see [15]). The processor gives us the ability to do computations internal to the smart card (primarily decryption). It also allows us to block out certain parts of the card by PIN numbers. Smart cards don't allow direct access to stored information (as with a magnetic strip). With the processor controlling access to the memory on the card, commands have to be sent to the processor on the card itself to get at the information on the card. This allows the processor to require the user to authenticate themselves to the smart card before information is returned.

4.2.1 PC/SC and MUSCLE

PC/SC [16] is an evolving, industry-standard API that is meant to provide a card reader standard so that any PC/SC-compliant smart card can work across multiple platforms. Its purpose is to create interoperability at all levels of the smart card and the reader through the use of a resource manager (which can be thought of as a server) and a service provider (which can be thought of as a client). These two layers of abstraction allow systems

to treat different smart cards and different readers in the same way. This allows multiple applications to talk to the server, and thus the smart card. PS/SC is basically a way to translate the byte commands which vary from manufacturer to manufacturer so one can make programs that will work on multiple card/reader combinations.

The MUSCLE (Movement for the Use of Smart Cards in a Linux Environment) project [12] is an implementation of the PC/SC standard in Linux. MUSCLE implements this with a client/server scheme. PC/SC defines a resource manager. This is the server. The resource manager manages the smart card readers. PC/SC also defines a service provider. This is the client to the resource manager. Applications connect to the resource manager through the service provider. The server and client communicate through the use of MICO, a free CORBA implementation [17].

We are using MUSCLE because (1) it is open source, (2) it works on Linux, and (3) it can ideally be used with any PC/SC-compliant smart card although right now MUSCLE only supports the newer Schlumberger cards. MUSCLE supports readers from numerous companies including Towitoko, Schlumberger, American Biometrics, and GemPlus though.

PS/SC defines the following classes as the base API for accessing the smart card:

SCARD This gives applications a way to connect to the smart card. In PC/SC, this class is required because it is how the client connects to a smart card. This class is the only class required by PC/SC. All other classes are optional.

FILEACCESS This class allows basic file access to files on the smart card. It allows you to do things such as delete, create, read and write files on the smart card.

CHVERIFICATION This allows management of CHVerification (Card Holder Verification) keys on the smart card. This can only be accomplished by someone who has the master key to the card.

CARDAUTH This class allows the card to authenticate itself cryptographically.

CRYPTPROV This class gives access to cryptographic features of the smart card.

These are the classes as defined by PC/SC. MUSCLE has implemented enough to create files, read them, and do some PIN verification. MUSCLE has not implemented CHVERIFICATION, CARDAUTH, or CRYPTPROV. These classes are not necessary, though, as we are planning to use Java for these functions (PC/SC does not define standards for Java). What we are planning to do is to send byte codes that will control the Java commands. We are using MUSCLE to be as PC/SC-compliant as

possible. Sending byte commands to a smart card is not really PC/SC-compatible, but it is the only way currently to run the Java. By using PC/SC as much as possible, we are giving people the chance to pick their reader (byte commands are card-specific, not reader-specific). MUSCLE's PC/SC implementation is also the cleanest way to send byte commands to the smart card.

The Java classes we use are basic I/O classes to send and receive information to and from the card. These functions are in the `javacard.framework.APDU` class. We also use the `javacard.framework.PIN` class for PIN verification. This can be used instead of the PC/SC class `CHVERIFICATION` if we choose. We will also be using `javacardx.crypto.RSA.PrivateKey` and `RSA.PublicKey`. These classes implement RSA algorithms and store private keys and will be used to do the actual decryption of the file key sent to the smart card by the group server.

Since a Java class exists as long as the card is functioning, once our program has been instantiated on the card and the key has been initialized, the card will take a file key encrypted to it (the card), decrypt it, and return it to the SFS Client so it can decrypt the file. The important thing is that the smart card allows us to store private keys on the card and, since the card can do the decryption internally, the file key will never leave the card.

4.3 SFS Client and XML Header

The SFS Client ties together the three parts of the Secure File System: the client file system (UFO in this case), the Group Server, and Smart Cards. It handles all of the cryptographic functions including the encryption and decryption of files and the encryption and decryption of all communication over the network.

SFS Client is first called when the client file system sends a file reference to it. SFS Client then determines if the file is encrypted and, if so, it obtains an XML header for the file. Using the smart card to digitally sign and encrypt all communications, SFS Client sends the Group Server the XML header and, if the current user has access, according to the XML header, the key to the file is then encrypted to the user's smart card and sent back to SFS Client. SFS Client then forwards the encrypted key to the smart card which decrypts it and sends the key back. SFS Client is then able to decrypt the file and return to the client file system, which transfers the file to the user application.

The header is a human-readable ASCII text file using XML (eXtensible Markup Language). This allows us to examine the XML header and see exactly what it contains, reducing the possibility of covert channels in the header.

It contains the author of the file, the owning group and the Access Control List (ACL). The ACL allows the user to explicitly define who can access the data. It also allows the use of key escrow techniques, such as requiring at least two people out of a three person group to access the data. Embedded in the ACL is the key for the file encrypted to the Group Server using public/private key cryptography. An example XML file is shown in Figure 4.1.

4.4 Detailed Example

First, a user accesses a file while running UFO. The file system call is caught and interpreted by UFO which accesses the file and then passes the file and its header to SFS Core. SFS Client then examines the header to determine which Group Server is involved. SFS Client then sends the header to the Group Server encrypting and signing the communications channel with the public key encryption routines on the client smart card. The Group Server, with a smart card of its own, is able to verify who the request is coming from and by examining the XML header it has received is able to grant or deny access to

the user. If access is granted, the Group Server takes the encrypted key embedded in the XML, decrypts it, and then re-encrypts it to the client's smart card. The Group Server then sends this re-encrypted key back to SFS Client. The SFS Client sends the encrypted key to the client smart card which decrypts and returns the key. The decrypted key is used by SFS Client to decrypt the file. Once the file has been decrypted SFS Client then returns control back to UFO and the user application is able to access the file transparently.

5 Conclusions and Future Work

SFS provides a mechanism for securing information from the information producer to the information consumer. We plan to port the SFS Client and Group Server code to other distributed and local file systems once the UFO implementation is complete.

Source code, papers, and information about new SFS developments will be made available at our web site: <http://www.securefilesystem.com>.

6 Acknowledgments

This work was supported by Storage Technology. In particular, we would like to thank Alope Guha, Walt Hinton, and Joan Wrabetz for their support of this project.

References

- [1] D. P. Reed & D. D. Clark J. H. Saltzer. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [2] Peter Gutmann, University of Auckland, New Zealand. The secure filesystem (sfs) for dos/windows. <http://www.cs.auckland.ac.nz/pgut001/sfs/index.html>, September 1996.
- [3] Alex Tormasov. The tordisk project. <http://www.cs.auckland.ac.nz/pgut001/sfs/index.html>, November 1997.
- [4] Phil Zimmerman. Pgp home page. <http://web.mit.edu/pgp/>.
- [5] Matt Blaze. A cryptographic file system for unix. In *First ACM Conference on Communications and Computing Security*, pages 33–43, Fairfax,VA, November 1993.
- [6] Matt Blaze. Key management in an encrypting file system. Boston,MA, June 1994.
- [7] University of Salerno. Tcfs home page. <http://www.globenet.it/ermmau/tcfs/index.html>, April 1997.
- [8] Parallel Data Lab. Nasd home page. <http://www.pdl.cs.cmu.edu/NASD/>.
- [9] Albert Alexandrov, Maximilian Ibel, Klaus Schausser, and Chris Scheiman. Extending the operating system at the user level: the ufo global file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 77–90, 1997.
- [10] Peter Braam and Philip Nelson. Removing bottlenecks in distributed filesystems. In *Proceedings of the 5th Annual Linux Expo*, pages 131–139, Raleigh, North Carolina, May 1999.
- [11] Scott Guthery and Timothy Jurgensen. *Smart Card Developer's Kit*. Macmillan Technical Publishing, 1998.
- [12] David Corcoran. <http://www.linuxnet.com>.
- [13] Erez Zadok and Ion Badulescu. A stackable file system interface for linux. In *Proceedings of the 5th Annual Linux Expo*, page 141=151, Raleigh, North Carolina, May 1999.

- [14] Bruce Schneier. *Applied Cryptography*. Wiley, 1996.
- [15] Bruce Schneier and Adam Shostack. Breaking up is hard to do: Modeling security threats for smart cards. In *First USENIX Symposium on Smart Cards*, 1999.
- [16] PC/SC Workgroup. Pc/sc home page. <http://www.smartcardsys.com/>.
- [17] <http://www.mico.org/>.