Design and Implementation of a Transparent Cryptographic File System for Unix

G. Cattaneo and G. Persiano
Dip. Informatica ed Appl.
Università di Salerno
Baronissi(SA) – Italy
{cattaneo,giuper}@dia.unisa.it

A. Del Sorbo, A. Celentano, A. Cozzolino, E. Mauriello, R. Pisapia

1 Introduction

Recent advances in hardware and communication technologies have made possible and cost effective to share a file system among several machines over a local (but possibly also a wide) area network. One of the most successful and widely used such applications is Sun's Network File System (NFS). NFS is very simple in structure but assumes a very strong trust model: the user trusts the remote file system server (which might be running on a machine in different country) and a network with his/her data. It is easy to see that neither assumption is a very realistic one. The server (or anybody with superuser privileges) might very well read the data on its local filesytem and it is well known that the Internet or any local area network (e.g, Ethernet) is very easy to tap (see for example, Berkeley's *tcpdump*[7, 5] application program). Impersonification of users is also another security drawback of NFS. In fact, most of the permission checking over NFS are performed in the kernel of the client. In such a context a *pirate* can temporarely assign to his own workstation the Internet address of victim. Without secure RPC [9] no further authentication procedure is requested. From here on, the pirate can issue NFS requests presenting himself with any (false) uid and therefore accessing for reading and writing any private data on the server, even protected data.

Given the above, a user seeking a certain level of security should take some measures. Possible solutions are to use either user-level cryptography or application level cryptography. A discussion of the drawbacks of these approaches is found in [4]. A better approach is to push encryption services into the operating system as done by M. Blaze in the design of his CFS [4].

In this paper, we propose a new cryptographic file system, which we call TCFS , as a suitable solution to the problem of privacy for distributed file system (see section 2.1). Our work improves on CFS by providing a deeper integration between the encryption service and the file system which results in a complete transparency of use to the user applications.

2 Design goals

In designing TCFS we were interested in providing robust security mechanism at the lowest possible cost to the user. The security mechanisms must guarantee that secure files should not be readable

- by users other than the legitimate owner;
- by tapping the communication lines between the user and remote file system server;
- by the superuser of the file system server.

In addition to the above we require that all sensitive meta data should be hidden. This means that for each file not only the content but also the filename is encrypted. Moreover on the same file system space should be possible for the users to easily distinguish *private* data from scratch data, such us object files which do not require encryption. The security of the content of files is guaranteed by means of DES [1]. We employ a variations of DES so to hide dependencies among various parts of the text (see Section 4). A feature of TCFS is that users need not to remember the decryption password associated with the files. This allows the use of random and difficult to guess passwords. We just generate a secret key and than we encrypt it with the user password storing the resulting string in a file indexed by the user id.

To maximize the level of security achievable we keep to the minimum the number of trusted entities. User need to trust only the kernel (and the superuser) of the machine on which their application programs accessing the secure files run. We stress that this minimal level of trust is necessary as the kernel can read the user space anyway and thus get the secure files after they have been decrypted. Our trust model fits perfectly the typical scenario in which TCFS will be employed: a network of diskless (or with very limited disk space) workstations with one or more remote file system servers. Here, typically each user (or small group of users) is assigned its own workstation and remote login among workstation is a very limited practice.

Keeping to the minimum the cost of using TCFS has various aspects which we now briefly discuss. First of all, TCFS should be completely transparent to the file system server and it should not modify data structures of the file system itself. This way the server system administrators would still be able to perform backups and run tools for fixing the file system (i.e. *fsck*) independently from TCFS clients. Our TCFS interfaces perfectly with a NFS server supporting extended file attributes. The issue of interoperability between TCFS and non cryptographic file system has arisen several times during the design of TCFS . For example, directories need special attention in the decryption and encryption phases. We encrypt only the filenames and not the related data. For example, if i-node numbers were encrypted, *fsck* would signal the anomaly at first run.

Second the access semantics to secure files should not change. Secure files should be accessible using the same standard systems calls. No additional flags in the `open()` `create()` system calls need to be used. This is necessary to make possible the use of software unaware of TCFS , without recompiling. When new files are created in a secure directory, they are by default created secure; similarly, for files created in a non secure directory. Thus, all temporary files created by applications running in a secure directory are by default created secure.

We also stress that TCFS has no need of creating special hidden files.

2.1 Blaze's CFS

The need for pushing cryptography at the operating system level has been advocated by M. Blaze [4] that proposed the Cryptographic File System. In the CFS, the user can associate a password to a secure directory. A secure directory is created by using the *ckmdir* and needs to be attached to a special directory (`/crypt`) before its files can be accessed. The user needs to supply the password relative to a directory each time the directory is attached. Once the directory is attached to `/crypt`, the user can access its files (which now appear in clear in the directory `/crypt`). Moreover, CFS suffers also of the following minor drawbacks.

- Each directory is associated with a different key. This makes key management from the user's side quite cumbersome.
- Once a directory is encrypted. all of its file are encrypted. This is not always necessary. For example, it might be desirable to have only the source files of a project in encrypted form but not the executable.
- CFS keeps all the vital management information relative to the directory in special hidden files in the directory itself.

TCFS instead is completely transparent to the user; accessing an encrypted file system involves no special operation and no password (other than the login password) needs to remembered and typed in by the user. Actually, the user might even ignore that he is operating on an encrypted file system.

3 Prototype implementation

The TCFS prototype has been implemented entirely at user level and communicates with the user applications through the kernel NFS data structures. Our prototype has been implemented on x86-based machines running Linux 1.2.13. Our work builds on the PD implementation of the NFS package V 2.0 [8].

3.1 Client side

The client side of TCFS consists of the following programs: `tcfsmount, tcfsumount` (utilities for mounting and unmounting the TCFS directories), `mountd` (modified mountd daemon to handle local mount requests), `tcfsd` (the TCFS daemon), `tcfslogin tcfslogout, genkey, passwd` (utilities for key management).

The full set of utilities and application programs will be presented in the final version of the paper.

3.2 Implementation issues

As already pointed out, TCFS has been implemented as a user process. This approach does not need to modify the kernel and thus has the following obvious advantages :

1. installation does not require kernel re-compilation and thus it is more likely to be trusted by system managers. In most cases system managers apply kernel patches very cautiously and accept patches only from a restricted set of well known kernel developers;
2. makes the code portable and readable;
3. makes the debugging process easier;
4. big speed up in the development time.

On the other hand our choice suffers of the following drawback. Communication between TCFS and kernel takes place by means a filter which requires each RPC call to traverse twice the kernel data structure. See figure 1.

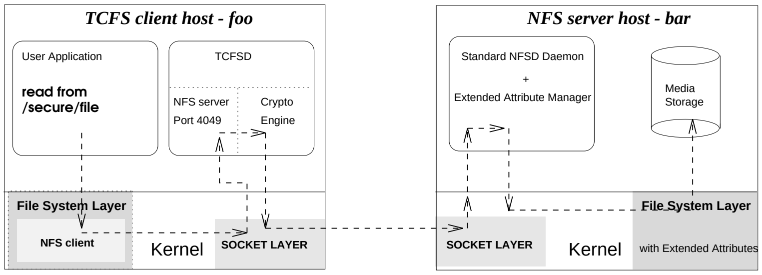

Figure 1: The main TCFS implementation scheme.

An alternative approach could be based on System V stream concept. We could push a new stream acting as a TCFS file system. Unfortunately streams are not supported by our environment and might be less portable through different operating systems. Moreover, with our approach we entirely reused the NFS file system primitives reducing the amount of code to be written. Also observe that streams are less portable and more difficult to debug (fixing streams inside the kernel is not a simple task).

3.2.1 An Example

Suppose that machine `foo` mounts the file system `remote` of machine `bar` on the directory `/local` and suppose we want to access `/local` using TCFS . We let `foo` export a dummy directory `/null` to be mounted locally on `/secure` using TCFS well-known port (4049 in our implementation) as an option of the mount program. In TCFS the two directories `/secure` and `/local` are identified; that is, the component `secure` of any access path is replaced with `/local`. This mechanism is very similar to the one employed by CFS. In CFS, when a directory is `cattach`ed to a subdirectory of the cryptographic file system `/crypt` an entry is created in a table. A substitution is then performed at each lookup function. TCFS instead works transparently and the substitution is performed once when directories are mounted by `tcfsmount` through RPCcalls to our modified `mountd` daemon. For example, when the following command is executed:

```
tcfsmount /null@localhost /secure /remote@bar /local
```

the following steps are performed:

1. A new specialized RPC call `ADMPROC_Stick` is issued to the `mountd` server. The effect of this call is to store the four strings in a table of daemon address space so to allow him to keep track of the substitution patterns.

2. `/null@localhost` is mounted to `/secure@localhost` using TCFS port (port 4049) through an RPC call to `ADMPROC_Mount`. In the created file handler the prefix `/null` is replaced with the local NFS mount point `/local`

3. The directory `/remote@bar` is mounted on `/local@localhost` using NFS port.

The final result is a virtual mount of `/remote@bar` on `/secure@localhost`. In fact, notice that all accesses to file systems mounted through NFS are realized through the `lookup` function that constructs a new file handler from the file handler associated with the mount point and its relative path.After the above three steps have been performed, all accesses on `/secure` will be performed through RPC calls to TCFS daemon which, in turn, will forward the calls to the remote server running on `bar`.

3.3 The Modified `mountd`

In order to activate the above described mechanism, and to keep it transparent to the user, we had to extend the function of the standard NFS daemon `mountd`. This daemon shares file system table with the kernel to associate a NFS device to each remotely mounted file system.

A new RPC program number has been declared in the RPC sources of mountd with the following RPC functions:

- `ADMPROC_STICK(tcfs_stickargs)`. Calling this procedure has the effect of creating a new entry in the table which stores the mounting points.
- `ADMPROC_UNSTICK(tcfs_unstickargs)`. This procedure is called to remove the specified entry when a TCFS file system is unmounted.
- `ADMPROC_LOOKUP(tcfs_lookupargs)`. This is an administrative procedure which is called to retrieve the stored information.

3.4 The TCFS daemon

The TCFS daemon includes the code to handle the RPC generated by the kernel and the management of extended attributes. The RPC are syntactically compatible to the NFS RPC but the one relative to read, write, directory handling and fh_compose have been extended to perform security operations.

Each time a new file handler is created, the extended attribute "`secure`" is tested. If the file is secure, then all successive read and write operation will be filtered through the encryption/decryption layer. If the bit is not set, the daemon will behave exactly as an NFS daemon.

DES encrypts and decrypts in blocks of 8 bytes. In order to make the read and write operations more efficient, we implemented a smart cache inside `tcfsd`[3]. This made the daemon faster even for non secure file, reducing the number of remote reads.

Special handling is required for directories. In fact, a block of directory data cannot be entirely encrypted, otherwise the server file system consistency will be corrupted. For each directory inode we encrypt filenames but not the other fields. This strategy allows us to reuse the standard file system depend `fsck` command.

To make easier the user interface, the attribute secure is inherited by files created in a secure directory. In this way the basic behaviour is CFS compatible.

3.5 The Server Side

The server used to test our prototype runs Linux with NFS daemon to export an `ext2` type file system. We only use one bit of the 16 bits reserved for the extended attributes provided by the `ext2` file system. Unfortunately, the NFS RPC procedures for reading and setting file attributes (`NFSPROC_GETATTR(nfs_fh)`, `NFSPROC_SETATTR(sattrargs)`) have not been designed to handle extended attributes. To let our client code to interact with a standard NFS server software, we provide a daemon for the server side to perform this task.

We defined two RPC procedures `NFSPROC_GETEXTAT(getextatargs)` and `NFSPROC_SETEXTAT(setextatargs)` which issue the necessary calls to `ioctl`) to read and set the extended attributes on the server. To summarize we can say that the server side is constituted by a standard NFS server plus a new daemon for managing the extended attributes over NFS.

4 File encryption

TCFS can be used with any block cipher (DES, IDEA) [1, 6] and in our prototype files are encrypted using the widely available implementation of DES of SECUDE project [2]. Each user is associated a file system key denoted by FSK and all files of a user are encrypted using the user's FSK. The encryption and key generation mechanisms will be discussed in the final version of the paper.

5 Performance Evaluation

In order to evaluate the overhead introduced by encryption of the data sent over the network, we performed a set of tests. To reduce the influence of external factors, such as network traffic and physical disk access time, we run the test in the following framework:

- The client machine running TCFS is an Intel 80486 100 Mhz processor running Linux.
- The server is a fast file server SPARCstation10 running Linux, with a 2 Gb fast SCSI disk.

The choice of this configuration for testing performance is based on the following observations. The major bottleneck of any cryptographic application is the encryption/decryption process which is a typical CPU bound task. Thus having a fast client to perform encryption could result in a situation in which CPU is idle waiting for the data to arrive from the network. In this setting no meaningful performance evaluation for TCFS can be carried-out: any test will just measure the throughput of the network [1]. On the other hand, if, as in our case, encryption is performed by a slower machine, we estimate the impact of cryptography on file system operations which is exactly what we are interested in.

SFS caches blocks after reading so to improve efficiency. This feature would give TCFS an unfair advantage over NFS (which does not cache the blocks). This explains why TCFS real time for reading operation without encryption is close to the one of NFS despite the fact that TCFS goes through the kernel twice.

This advantage disappears for write operations. This is due to the fact that we choose not to cache write so to guarantee the consistency of remote file. In any case the real time measured is affected by the size of the internal operating system caches.

In order to evaluate any extra CPU time spent in the local operation of `tcfsd` we inserted a global variable in the daemon to accumulate the CPU time measured as difference between the end time and initial time obtained from the field `ru_utime` of the structure returned by the function `getrusage`. Of course, this variable has been reset at the beginning of each test and is read at the end by means of two dedicated signal handlers. The values of this variable for our tests are reported in the columns 3, 5, 7, and 9 of table 1.

The real time values have been obtained by means of the csh built in command `time` with the following command: `time dd bs=1024` $\cdots$.

All the values reported in the table are average values over ten different runs, and the execution of all test are mixed to avoid caching side effect. The times are expressed in seconds.

We observe that summing encryption and transfer do not necessarily give the total real time. In fact, encryption and transfer can interleave and thus get a better real time. As the degree of the interleaving increases, the throughput of the cryptographic file system nears the throughput of a standard network file system. The optimal situation is when the rate of encryption of the CPU is equal to the the network transfer rate, in which case the cryptographic file system has a performance very close to a standard remote file system. The tests show that our implementation is very close to such a situation.

In sums the table 1 shows the following results:

- The overall performance of TCFS for writing without encryption is reasonably close to half the NFS throughput. Reading instead is closer to NFS performance thanks to TCFS caches.

[1] In our configuration the maximal network throughput for UDP remote connections has been stated to about 250 Kbytes/sec.

- The CPU time measured inside the `tcfsd` execution of the read/write RPC is slightly more than the CPU time required to encrypt/decrypt a memory stream of the same size.

- Even with such a slow client the time spent for encryption is completely absorbed by the transfer time. We tested TCFS also with faster DES implementations, getting a 50% of processor idle time for such kind of tests.

- No extra cost is incurred into for key manipulation or path substitution.

file size	TCFS secure = 0				TCFS secure = 1				NFS		DES perf
	read		write		read		write		read	write	
	real	CPU	real	CPU	real	CPU	real	CPU	real	real	CPU
10 Kb	0.39	0.05	0.61	0.03	0.58	0.19	0.53	0.20	0.29	0.40	0.19
100 Kb	2.52	0.13	4.15	0.20	4.37	1.89	4.59	1.99	2.03	2.22	1.73
1 Mb	21.98	1.72	43.11	2.75	42.34	19.41	45.77	19.55	21.99	21.48	17.58
10 Mb	269.66	20.28	450.48	18.46	451.72	192.76	462.64	196.22	220.50	227.00	176.98

Table 1: Table summarizing the results of our tests

References

[1] Data encryption standard. Technical Report Publication 46, Federal Information Processing Standards, January, 15 1977.

[2] SECUDE manual: Low level cryptographic functions. Technical report, Gesellschaft fur Mathematik und Datenverarbeitung (GMD), July 1994.

[3] Maurice J. Bach. *The Design of The Unix Operating System*. Prentice-Hall International Editions, 1987.

[4] M. Blaze. A cryptographic file system for unix. In *First ACM Conference on Communications and Computing Security*, pages 158–165, Fairfax, VA, 1993.

[5] Douglas Comer. *Internetworking with TCP/IP*. Prentice-Hall International Editions, 1988.

[6] X. Lai and J. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology – EUROCRYPT 90*, pages 389–404, 1990.

[7] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Winter Usenix Conference*, 1993.

[8] M. Shand, D. J. Becker, R. Sladkey, O. Zborowski, and F. N. van Kempen. The LINUX User-Space NFS server. Technical report, Joint Microelectronics Research Centre, School of Electrical Engineering, University of NSW, Kensington, NSW 2033, Australia.

[9] Bradley Taylor and David Goldberg. Secure networking in the SUN environment.