Demands, Solutions, and Improvements for Linux Filesystem Security

Michael Austin Halcrow
International Business Machines, Inc.
`mike@halcrow.us`

Abstract

Securing file resources under Linux is a team effort. No one library, application, or kernel feature can stand alone in providing robust security. Current Linux access control mechanisms work in concert to provide a certain level of security, but they depend upon the integrity of the machine itself to protect that data. Once the data leaves that machine, or if the machine itself is physically compromised, those access control mechanisms can no longer protect the data in the filesystem. At that point, data privacy must be enforced via encryption.

As Linux makes inroads in the desktop market, the need for transparent and effective data encryption increases. To be practically deployable, the encryption/decryption process must be secure, unobtrusive, consistent, flexible, reliable, and efficient. Most encryption mechanisms that run under Linux today fail in one or more of these categories. In this paper, we discuss solutions to many of these issues via the integration of encryption into the Linux filesystem. This will provide access control enforcement on data that is not necessarily under the control of the operating environment. We also explore how stackable filesystems, Extended Attributes, PAM, GnuPG web-of-trust, supporting libraries, and applications (such as GNOME/KDE) can all be orchestrated to provide robust encryption-based access control over filesystem content.

1 Development Efforts

This paper is motivated by an effort on the part of the IBM Linux Technology Center to enhance Linux filesystem security through better integration of encryption technology. The author of this paper is working together with the external community and several members of the LTC in the design and development of a transparent cryptographic filesystem layer in the Linux kernel. The "we" in this paper refers to immediate members of the author's development team who are working together on this project, although many others outside that development team have thus far had a significant part in this development effort.

2 The Filesystem Security

2.1 Threat Model

Computer users tend to be overly concerned about protecting their credit card numbers from being sniffed as they are transmitted over the

Internet. At the same time, many do not think twice when sending equally sensitive information in the clear via an email message. A thief who steals a removable device, laptop, or server can also read the confidential files on those devices if they are left unprotected. Nevertheless, far too many users neglect to take the necessary steps to protect their files from such an event. Your liability limit for unauthorized charges to your credit card is $50 (and most credit card companies waive that liability for victims of fraud); on the other hand, confidentiality cannot be restored once lost.

Today, we see countless examples of neglect to use encryption to protect the integrity and the confidentiality of sensitive data. Those who are trusted with sensitive information routinely send that information as unencrypted email attachments. They also store that information in clear text on disks, USB keychain drives, backup tapes, and other removable media. GnuPG[7] and OpenSSL[8] provide all the encryption tools necessary to protect this information, but these tools are not used nearly as often as they ought to be.

If required to go through tedious encryption or decryption steps every time they need to work with a file or share it, people will select insecure passwords, transmit passwords in an insecure manner, fail to consider or use public key encryption options, or simply stop encrypting their files altogether. If security is overly obstructive, people will remove it, work around it, or misuse it (thus rendering it less effective). As Linux gains adoption in the desktop market, we need integrated file integrity and confidentiality that is seamless, transparent, easy to use, and effective.

2.2 Integration of File Encryption into the Filesystem

Several solutions exist that solve separate pieces of the problem. In one example highlighting transparency, employees within an organization that uses IBM™ Lotus Notes™ [9] for its email will not even notice the complex PKI or the encryption process that is integrated into the product. Encryption and decryption of sensitive email messages is seamless to the end user; it involves checking an "Encrypt" box, specifying a recipient, and sending the message. This effectively addresses a significant file in-transit confidentiality problem. If the local replicated mailbox database is also encrypted, then it also addresses confidentiality on the local storage device, but the protection is lost once the data leaves the domain of Notes (for example, if an attached file is saved to disk). The process must be seamlessly integrated into *all* relevant aspects of the user's operating environment.

In Section 4, we discuss filesystem security in general under Linux, with an emphasis on confidentiality and integrity enforcement via cryptographic technologies. In Section 6, we propose a mechanism to integrate encryption of files at the filesystem level, including integration of GnuPG[7] web-of-trust, PAM[10], a stackable filesystem model[2], Extended Attributes[6], and libraries and applications, in order to make the entire process as transparent as possible to the end user.

3 A Team Effort

Filesystem security encompasses more than just the filesystem itself. It is a team effort, involving the kernel, the shells, the login processes, the filesystems, the applications, the administrators, and the users. When we speak of "filesystem security," we refer to the security of the files in a filesystem, no matter what ends up providing that security.

For any filesystem security problem that exists, there are usually several different ways of solving it. Solutions that involve modifications in the kernel tend to introduce less overhead. This is due to the fact that context switches and copying of data between kernel and user memory is reduced. However, changes in the kernel may reduce the efficiency of the kernel's VFS while making it both harder to maintain and more bug-prone. As notable exceptions, Erez Zadok's stackable filesystem framework, FiST[3], and Loop-aes, require no change to the current Linux kernel VFS. Solutions that exist entirely in userspace do not complicate the kernel, but they tend to have more overhead and may be limited in the functionality they are able to provide, as they are limited by the interface to the kernel from userspace. Since they are in userspace, they are also more prone to attack.

4 Aspects of Filesystem Security

Computer security can be decomposed into several areas:

- Identifying who you are and having the machine recognize that identification (*authentication*).
- Determining whether or not you should be granted access to a resource such as a sensitive file (*authorization*). This is often based on the permissions associated with the resource by its owner or an administrator (*access control*).
- Transforming your data into an encrypted format in order to make it prohibitively costly for unauthorized users to decrypt and view (*confidentiality*).
- Performing checksums, keyed hashes, and/or signing of your data to make unauthorized modifications of your data detectable (*integrity*).

4.1 Filesystem Integrity

When people consider filesystem security, they traditionally think about access control (file permissions) and confidentiality (encryption). File integrity, however, can be just as important as confidentiality, if not more so. If a script that performs an administrative task is altered in an unauthorized fashion, the script may perform actions that violate the system's security policies. For example, many rootkits modify system startup and shutdown scripts to facilitate the attacker's attempts to record the user's keystrokes, sniff network traffic, or otherwise infiltrate the system.

More often than not, the value of the data stored in files is greater than that of the machine that hosts the files. For example, if an attacker manages to insert false data into a financial report, the alteration to the report may go unnoticed until substantial damage has been done; jobs could be at stake and in more extreme cases even criminal charges against the user could result . If trojan code sneaks into the source repository for a major project, the public release of that project may contain a backdoor.[1]

Many security professionals foresee a nightmare scenario wherein a widely propagated Internet worm quietly alters the contents of word

[1] A high-profile example of an attempt to do this occurred with the Linux kernel last year. Luckily, the source code management process used by the kernel developers allowed them to catch the attempted insertion of the trojan code before it made it into the actual kernel.

processing and spreadsheet documents. Without any sort of integrity mechanism in place in the vast majority of the desktop machines in the world, nobody would know if any data that traversed vulnerable machines could be trusted. This threat could be very effectively addressed with a combination of a kernel-level mandatory access control (MAC)[11] protection profile and a filesystem that provides integrity and auditing capabilities. Such a combination would be resistant to damage done by a root compromise, especially if aided by a Trusted Platform Module (TPM)[13] using attestation.

One can approach filesystem integrity from two angles. The first is to have strong authentication and authorization mechanisms in place that employ sufficiently flexible policy languages. The second is to have an auditing mechanism, to detect unauthorized attempts at modifying the contents of a filesystem.

4.1.1 Authentication and Authorization

The filesystem must contain support for the kernel's security structure, which requires stateful security attributes on each file. Most GNU/Linux applications today use PAM[10] (see Section 4.1.2 below) for authentication and process credentials to represent their authorization; policy language is limited to what can be expressed using the file owner and group, along with the owner/group/world read/write/execute attributes of the file. The administrator and the current owner have the authority to set the owner of the file or the read/write/execute policies for that file. In many filesystems, files may also contain additional security flags, such as an immutable or append-only flag.

Posix Access Control Lists (ACL's)[6] provide for more stringent delegations of access authority on a per-file basis. In an ACL, individual read/write/execute permissions can be assigned to the owner, the owning group, individual users, or groups. Masks can also be applied that indicate the maximum effective permissions for a class.

For those who require even more flexible access control, SE Linux[15] uses a powerful policy language that can express a wide variety of access control policies for files and filesystem operations. In fact, Linux Security Module (LSM)[14] hooks (see Section 4.1.3 below) exist for most of the security-relevant filesystem operations, which makes it easier to implement custom filesystem-agnostic security models. Authentication and authorization are pretty well covered with a combination of existing filesystem, kernel, and user-space solutions that are part of most GNU/Linux distributions. Many Linux distributions could, however, do a better job of aiding both the administrator and the user in understanding and using all the tools that they have available to them.

Policies that safeguard sensitive data should include timeouts, whereby the user must periodically re-authenticate in order to continue to access the data. In the event that the authorized users neglect to lock down the machine before leaving work for the day, timeouts help to keep the custodial staff from accessing the data when they comes in at night to clean the office. As usual, this must be implemented in such a way as to be unobtrusive to the user. If a user finds a security mechanism overly imposing or inconvenient, he will usually disable or circumvent it.

4.1.2 PAM

Pluggable Authentication Modules (PAM)[10] implement authentication-related security policies. PAM offers discretionary access control

(DAC)[12]; applications must defer to PAM in order to authenticate a user. If the PAM module function that is called returns an affirmative answer, then the application considers the action to be authenticated, and vice versa. The exact mechanism that the PAM function uses to evaluate the authentication is dependent on the module called.[2]

In the case of filesystem security and encryption, PAM can be employed to obtain and forward keys to a filesystem encryption layer in kernel space. This would allow seamless integration with any key retrieval mechanism that can be coded as a Pluggable Authentication Module.

4.1.3 LSM

Linux Security Modules (LSM) can provide customized security models. One possible use of LSM is to allow decryption of certain files only when a physical device is connected to the machine. This could be, for example, a USB keychain device, a Smartcard, or an RFID device. Some devices of these classes can also be used to house the encryption keys (retrievable via PAM, as previously discussed).

4.1.4 Auditing

The second angle to filesystem integrity is auditing. Auditing should only fill in where the authentication and authorization mechanisms fall short. In a utopian world, where security systems are perfect and trusted people always act trustworthily, auditing does not have much of a use. In reality, code that implements security has defects and vulnerabilities. Passwords can be compromised, and authorized people can act in an untrustworthy manner. Auditing can involve keeping a log of all changes made to the attributes of the file or to the file data itself. It can also involve taking snapshots of the attributes and/or contents of the file and comparing the current state of the file with what was recorded in a prior snapshot.

[2]This is parameterizable in the configuration files found under */etc/pam.d/*

Intrusion detection systems (IDS), such as Tripwire[16], AIDE[17], or Samhain[18], perform auditing functions. As an example, Tripwire periodically scans the contents of the filesystem, checking file attributes, such as the size, the modification time, and the cryptographic hash of each file. If any attributes for the files being checked are found to be altered, Tripwire will report it. This approach can work fairly well in cases where the files are not expected to change very often, as is the case with most system scripts, shared libraries, executables, or configuration files. However, care must be taken to assure that the attacker cannot also modify the Tripwire's database when he modifies a system file; the integrity of the IDS system itself must also be assured.

In cases where a file changes often, such as a database file or a spreadsheet file in an active project, we see a need for a more dynamic auditing solution - one which is perhaps more closely integrated with the filesystem itself. In many cases, the simple fact that the file has changed does not imply a security violation. We must also know who made the change. More robust security requirements also demand that we know what parts of the file were changed and when the changes were made. One could even imagine scenarios where the context of the change must also be taken into consideration (i.e., who was logged in, which processes were running, or what network activity was taking place at the time the change was made).

File integrity, particularly in the area of auditing, is perhaps the security aspect of Linux filesystems that could use the most improvement. Most efforts in secure filesystem development have focused on confidentiality more so than integrity, and integrity has been regulated to the domain of userland utilities that must periodically scan the entire filesystem. Sometimes, just knowing that a file has been changed is insufficient. Administrators would like to know exactly how the attacker made the changes and under what circumstances they were made.

Cryptographic hashes are often used. These can detect unauthorized circumvention of the filesystem itself, as long as the attacker forgets (or is unable) to update the hashes when making unauthorized changes to the files. Some auditing solutions, such as the Linux Auditing System (LAuS)[3] that is part of SuSE Linux Enterprise Server, can track system calls that affect the filesystem. Another recent addition to the 2.6 Linux kernel is the Light-weight Auditing Framework written by Rik Faith[28]. These are implemented independently of the filesystem itself, and the level of detail in the records is largely limited to the system call parameters and return codes. It is advisable that you keep your log files on a separate machine than the one being audited, since the attacker could modify the audit logs themselves once he has compromised the machine's security.

4.1.5 Improvements on Integrity

Extended Attributes provide for a convenient way to attach metadata relating to a file to the file itself. On the premise that possession of a secret equates to authentication, every time an authenticated subject makes an authorized write to a file, a hash over the concatenation of that secret to the file contents (keyed hashing; HMAC is one popular standard) can be written as an Extended Attribute on that file. Since this action would be performed on the filesystem level, the user would not have to conscientiously re-run userspace tools to perform such an operation every time he wants to generate an integrity verifier on the file.

[3]Note that LAuS is being covered in more detail in the 2004 Ottawa Linux Symposium by Doc Shankar, Emily Ratliff, and Olaf Kirch as part of their presentation regarding CAPP/EAL3+ Certification.

This is an expensive operation to perform over large files, and so it would be a good idea to define extent sizes over which keyed hashes are formed, with the Extended Attributes including extent descriptors along with the keyed hashes. That way, a small change in the middle of a large file would only require the keyed hash to be re-generated over the extent in which the change occurs. A keyed hash over the sequential set of the extent hashes would also keep an attacker from swapping around extents undetected.

4.2 File Confidentiality

Confidentiality means that only authorized users can read the contents of a file. Sometimes the names of the files themselves or a directory structure can be sensitive. In other cases, the sizes of the files or the modification times can betray more information than one might want to be known. Even the security policies protecting the files can reveal sensitive information. For example, "Only employees of Novell and SuSE can read this file" would imply that Novell and SuSE are collaborating on something, and neither of them may want this fact to be public knowledge as of yet. Many interesting protocols have been developed that can address these sorts of issues; some of then are easier to implement than others.

When approaching the question of confidentiality, we assume that the block device that contains the file is vulnerable to physical compromise. For example, a laptop that contains sensitive material might be lost, or a database server might be stolen in a burglary. In either event, the data on the hard drive must not be readable by an unauthorized individual. If any individual must be authenticated before he is able to access to the data, then the data is protected against unauthorized access.

Surprisingly, many users surrender their own data's confidentiality (and more often than not they do so unwittingly). It has been my personal observation that most people do not fully understand the lack of confidentiality afforded their data when they send it over the Internet. To compound this problem, comprehending and even using most encryption tools takes considerable time and effort on the part of most users. If sensitive files could be *encrypted by default*, only to be decrypted by those authorized at the time of access, then the user would not have to expend so much effort toward protecting his confidentiality.

By putting the encryption at the filesystem layer, this model becomes possible without any modifications to the applications or libraries. A policy at that layer can dictate that certain processes, such as the mail client, are to receive the encrypted version any files that are read from disk.

4.2.1 Encryption

File confidentiality is most commonly accomplished through encryption. For performance reasons, secure filesystems use symmetric key cryptography, like AES or Triple-DES, although an asymmetric public/private keypair may be used to encrypt the symmetric key in some key management schemes. This hybrid approach is in common use through SSL and PGP encryption protocols.

One of our proposals to extend Cryptfs is to mirror the techniques used in GnuPG encryption. If the symmetric key that protects the contents of a file is encrypted with the public key of the intended recipient of the file and stored as an Extended Attribute of the file, then that file can be transmitted in multiple ways (e.g., physical device such as removable storage); as long as the Extended Attributes of the file are preserved across filesystem transfers, then the recipient with the corresponding private key has all the information that his Cryptfs layer needs to transparently decrypt the contents of the file.

4.2.2 Key Management

Key management will make or break a cryptographic filesystem.[5] If the key can be easily compromised, then even the strongest cipher will provide weak protection. If your key is accessible in an unencrypted file or in an unprotected region of memory, or if it is ever transmitted over the network in the clear, a rogue user can capture that key and use it later. Most passwords have poor entropy, which means that an attacker can have pretty good success with a brute force attack against the password. Thus the weakest link in the chain for password-based encryption is usually the password itself. The Cryptographics Filesystem (CFS)[22] mandates that the user choose a password with a length of at least 16 characters[4].

Ideally, the key would be kept in password-encrypted form on a removable device (like a USB keychain drive) that is stored separately from the files that the key is used to encrypt. That way, an attacker would have to both compromise the password and gain physical access

[4]The subject of secure password selection, although an important one, is beyond the scope of this article. Recommended reading on this subject is at `http://www.alw.nih.gov/Security/Docs/passwd.html`.

to the removable device before he could decrypt your files.

Filesystem encryption is one of the most exciting applications for the Trusted Computing Platform. Given that the attacker has physical access to a machine with a Trusted Platform Module, it is significantly more difficult to compromise the key. By using secret sharing (otherwise known as *key splitting*)[4], the actual key used to decrypt a file on the filesystem can be contained as both the user's key and the machine's key (as contained in the TPM). In order to decrypt the files, an attacker must not only compromise the user key, but he must also have access to the machine on which the TPM chip is installed. This "binds" the encrypted files to the machine. This is especially useful for protecting files on removable backup media.

4.2.3 Cryptanalysis

All block ciphers and most stream ciphers are, to various degrees, vulnerable to successful cryptanalysis. If a cipher is used improperly, then it may become even easier to discover the plaintext and/or the key. For example, with certain ciphers operating in certain modes, an attacker could discover information that aids in cryptanalysis by getting the filesystem to re-encrypt an already encrypted block of data. Other times, a cryptanalyst can deduce information about the type of data in the encrypted file when that data has predictable segments of data, like a common header or footer (thus allowing for a known-plaintext attack).

4.2.4 Cipher Modes

A block encryption mode that is resistant to cryptanalysis can involve dependencies among chains of bytes or blocks of data. Cipher-block-chaining (CBC) mode, for example, provides adequate encryption in many circumstances. In CBC mode, a change to one block of data will require that all subsequent blocks of data be re-encrypted. One can see how this would impact performance for large files, as a modification to data near the beginning of the file would require that all subsequent blocks be read, decrypted, re-encrypted, and written out again.

This particular inefficiency can be effectively addressed by defining chaining extents. By limiting regions of the file that encompass chained blocks, it is feasible to decrypt and re-encrypt the smaller segments. For example, if the block size for a cipher is 64 bits (8 bytes) and the block size, which is (we assume) the minimum unit of data that the block device driver can transfer at a time (512 bytes) then one could limit the number of blocks in any extent to 64 blocks. Depending on the plaintext (and other factors), this may be too few to effectively counter cryptanalysis, and so the extent size could be set to a small multiple of the page size without severely impacting overall performance. The optimal extent size largely depends on the access patterns and data patterns for the file in question; we plan on benchmarking against varying extent lengths under varying access patterns.

4.2.5 Key Escrow

The proverbial question, "What if the sysadmin gets hit by a bus?" is one that no organization should ever stop asking. In fact, sometimes no one person should alone have independent access to the sensitive data; multiple passwords may be required before the data is decrypted. Shareholders should demand that no single person in the company have full access to certain valuable data, in order to mitigate the damage to the company that could be done by a single corrupt administrator or executive. Methods for secret sharing can be employed to assure that multiple keys be required for file access, and (m,n)-threshold schemes [4] can ensure that the data is retrievable, even if a certain number of the keys are lost. Secret sharing would be easily implementable as part of any of the existing cryptographic filesystems.

4.3 File Resilience

The loss of a file can be just as devastating as the compromise of a file. There are many well-established solutions to performing backups of your filesystem, but some cryptographic filesystems preclude the ability to efficiently and/or securely use them. Backup tapes tend to be easier to steal than secure computer systems are, and if unencrypted versions of secure files exist on the tapes, that constitutes an often-overlooked vulnerability.

The Linux 2.6 kernel cryptoloop device[5] filesystem is an all-or-nothing approach. Most backup utilities must be given free reign on the unencrypted directory listings in order to perform incremental backups. Most other encrypted filesystems keep sets of encrypted files in directories in the underlying filesystem, which makes incremental backups possible without giving the backup tools access to the unencrypted content of the files.

The backup utilities must, however, maintain backups of the metadata in the directories containing the encrypted files in addition to the files themselves. On the other hand, when the filesystem takes the approach of storing the cryptographic metadata as Extended Attributes for each file, then backup utilities need only worry about copying just the file in question to the backup medium (preserving the Extended Attributes, of course).

[5]Note that this is deprecated and is in the process of being replaced with the Device Mapper crypto target.

4.4 Advantages of FS-Level, EA-Guided Encryption

Most encrypted filesystem solutions either operate on the entire block device or operate on entire directories. There are several advantages to implementing filesystem encryption at the filesystem level and storing encryption metadata in the Extended Attributes of each file:

- Granularity: Keys can be mapped to individual files, rather than entire block devices or entire directories.
- Backup Utilities: Incremental backup tools can correctly operate without having to have access to the decrypted content of the files it is backing up.
- Performance: In most cases, only certain files need to be encrypted. System libraries and executables, in general, do not need to be encrypted. By limiting the actual encryption and decryption to only those files that really need it, system resources will not be taxed as much.
- Transparent Operation: Individual encrypted files can be easily transfered off of the block device without any extra transformation, and others with authorization will be able to decrypt those files. The userspace applications and libraries do not need to modified and recompiled to support this transparency.

Since all the information necessary to decrypt a file is contained in the Extended Attributes of the file, it is possible for a user on a machine that is not running Cryptfs to use userland utilities to access the contents of the file.

This also applies to other security-related operations, like verifying keyed hashes. This addresses compatibility issues with machines that are not running the encrypted filesystem layer.

5 Survey of Encrypted Filesystems

5.1 Encrypted Loopback Filesystems

5.1.1 Loop-aes

The most well-known method of encrypting a filesystem is to use a loopback encrypted filesystem[6]. Loop-aes[20] is part of the 2.6 Linux kernel (CONFIG_BLK_DEV_CRYPTOLOOP). It performs encryption at the block device level. With Loop-aes, the administrator can choose whatever cipher he wishes to use with the filesystem. The *mount* package on most popular GNU/Linux distributions contains the *losetup* utility, which can be used to set up the encrypted loopback mount (you can choose whatever cipher that the kernel supports; we use blowfish in this example):

```
root# modprobe cryptoloop
root# modprobe blowfish
root# dd if=/dev/urandom of=encrypted.img \
      bs=4k count=1000
root# losetup -e blowfish /dev/loop0 \
      encrypted.img
root# mkfs.ext3 /dev/loop0
root# mkdir /mnt/unencrypted-view
root# mount /dev/loop0 /mnt/unencrypted-view
```

The loopback encrypted filesystem falls short in the fact that it is an all-or-nothing solution. It is impossible for most standard backup utilities to perform incremental backups on sets of encrypted files without being given access to the unencrypted files. In addition, remote users will need to use IPSec or some other network encryption layer when accessing the files, which must be exported from the unencrypted mount point on the server. Loop-aes is, however, the best performing encrypted filesystem that is freely available and integrated with most GNU/Linux distributions. It is an adequate solution for many who require little more than basic encryption of their entire filesystems.

[6]Note that Loop-aes is being deprecated, in favor of Device Mapping (DM) Crypt

5.1.2 BestCrypt

BestCrypt[23] is a non-free product that uses a loopback approach, similar to Loop-aes.

5.1.3 PPDD

PPDD citeppdd is a block device driver that encrypts and decrypts data as it goes to and comes from another block device. It works very much like Loop-aes; in fact, in the 2.4 kernel, it uses the loopback device, as Loop-aes does. PPDD has not been ported to the 2.6 kernel. Loop-aes takes the same approach, and Loop-aes ships with the 2.6 kernel itself.

5.2 CFS

The Cryptographic Filesystem (CFS)[22] by Matt Blaze is a well established transparent encrypted filesystem, originally written for BSD platforms. CFS is implemented entirely in userspace and operates similarly to NFS. A userspace daemon, cfsd, acts as a pseudo-NFS server, and the kernel makes RPC calls to the daemon. The CFS daemon performs transparent encryption and decryption when writing and reading data. Just as NFS can export a directory from any exportable filesystem, CFS can do the same, while managing the encryption on top of that filesystem.

In the background, CFS stores the metadata necessary to encrypt and decrypt files with the files being encrypted or decrypted on the filesystem. If you were to look at those directories directly, you would see a set of files with encrypted values for filenames, and there would be a handful of metadata files mixed in. When accessed through CFS, those metadata files are hidden, and the files are transparently encrypted and decrypted for the user applications (with the proper credentials) to freely work with the data.

While CFS is capable of acting as a remote NFS server, this is not recommended for many reasons, some of which include performance and security issues with plaintext passwords and unencrypted data being transmitted over the network. You would be better off, from a security perspective (and perhaps also performance, depending on the number of clients), to use a regular NFS server to handle remote mounts of the encrypted directories, with local CFS mounts off of the NFS mounts.

Perhaps the most attractive attribute of CFS is the fact that it does not require any modifications to the standard Linux kernel. The source code for CFS is freely obtainable. It is packaged in the Debian repositories and is also available in RPM form. Using apt, CFS is perhaps the easiest encrypted filesystem for a user to set up and start using:

```
root# apt-get install cfs
user# cmkdir encrypted-data
user# cattach encrypted-data unencrypted-view
```

The user will be prompted for his password at the requisite stages. At this point, anything the user writes to or reads from */crypt/unencrypted-view* will be transparently encrypted to and decrypted from files in *encrypted-data*. Note that any user on the system can make a new encrypted directory and attach it. It is not necessary to initialize and mount an entire block device, as is the case with Loop-aes.

5.3 TCFS

TCFS[24] is a variation on CFS that includes secure integrated remote access and file integrity features. TCFS assumes the client's workstation is trusted, and the server cannot necessarily be trusted. Everything sent to and from the server is encrypted. Encryption and decryption take place on the client side.

Note that this behavior can be mimicked with a CFS mount on top of an NFS mount. However, because TCFS works within the kernel (thus requiring a patch) and does not necessitate two levels of mounting, it is faster than an NFS+CFS combination.

TCFS is no longer an actively maintained project. The last release was made three years ago for the 2.0 kernel.

5.4 Cryptfs

As a proof-of-concept for the FiST stackable filesystem framework, Erez Zadok, et. al. developed Cryptfs[1]. Under Cryptfs, symmetric keys are associated with groups of files within a single directory. The keys are generated with password that is entered at the time that the filesystem is mounted. The Cryptfs mount point provides an unencrypted view of the directory that contains the encrypted files.

The authors of this paper are currently working on extending Cryptfs to provide seamless integration into the user's desktop environment (see Section 6).

5.5 Userspace Encrypted Filesystems

EncFS[25] utilizes the Filesystem in Userspace (FUSE) library and kernel module to implement an encrypted filesystem in userspace. Like CFS, EncFS encrypts on a per-file basis.

CryptoFS[26] is similar to EncFS, except it uses the Linux Userland Filesystem (LUFS) library instead of FUSE.

SSHFS[27], like CryptoFS, uses the LUFS kernel module and userspace daemon. It limits itself to encrypting the files via SFTP as they are transfered over a network; the files stored on disk are unencrypted. From the user perspective, all file accesses take place as though they were being performed on any regular filesystem (opens, read, writes, etc.). SSHFS transfers the files back and forth via SFTP with the file server as these operations occur.

5.6 Reiser4

ReiserFS version 4 (Reiser4)[29], while still in the development stage, features pluggable security modules. There are currently proposed modules for Reiser4 that will perform encryption and auditing.

5.7 Network Filesystem Security

Much research has taken place in the domain of networking filesystem security. CIFS, NFSv4, and other networking filesystems face special challenges in relation to user identification, access control, and data secrecy. The NFSv4 protocol definition in RFC 3010 contains descriptions of security mechanisms in section 3[30].

6 Proposed Extensions to Cryptfs

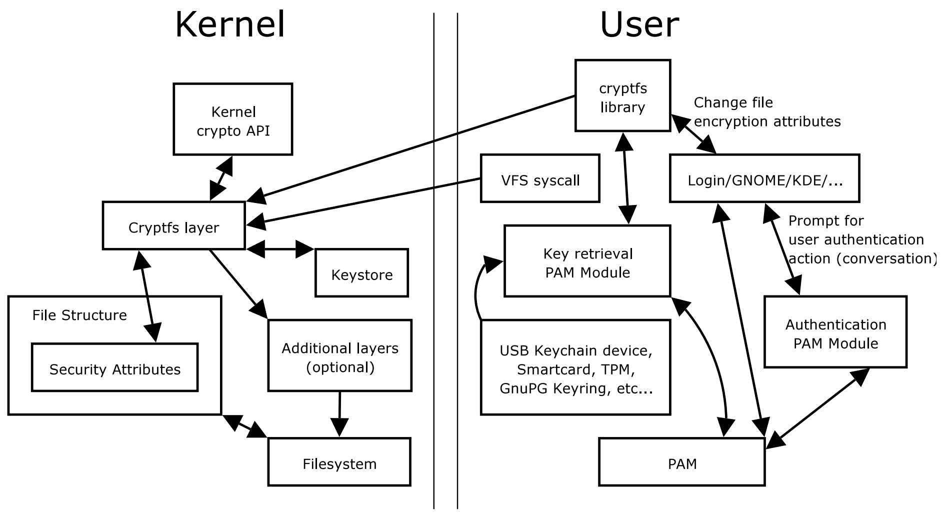
Our proposal is to place file encryption metadata into the Extended Attributes (EA's) of the file itself. Extended Attributes are a generic interface for attaching metadata to files. The Cryptfs layer will be extended to extract that information and to use the information to direct the encrypting and decrypting of the contents of the file. In the event that the filesystem does not support Extended Attributes, another filesystem layer can provide that functionality. The stackable framework effectively allows Cryptfs to operate on top of *any* filesystem.

The encryption process is very similar to that of GnuPG and other public key cryptography programs that use a hybrid approach to encrypting data. By integrating the process into the filesystem, we can achieve a a greater degree of transparency, without requiring any changes to userspace applications or libraries.

Under our proposed design, when a new file is created as an encrypted file (file creation policy enacted by Cryptfs can be dictated by directory attributes or globally defined behavior), the Cryptfs layer generates a new symmetric key K_s for the encryption of the data that will be written. The owner of the file is automatically authorized to access the file, and so the symmetric key is encrypted with the public key of the owner of the file K_u, which was passed into the Cryptfs layer at the time that the user logged in by a Pluggable Authentication Module linked against libcryptfs. The encrypted symmetric key is then added to the Extended Attribute set of the file:

$\{K_s\}K_u$

Suppose that the user at this point wants to grant Alice access to the file. Alice's public key, K_a, is in the user's GnuPG keyring. He



Kernel
User
Kernel crypto API
Cryptfs layer
Keystore
File Structure
Security Attributes
Additional layers (optional)
Filesystem
cryptfs library
Change file encryption attributes
VFS syscall
Login/GNOME/KDE/...
Prompt for user authentication action (conversation)
Key retrieval PAM Module
Authentication PAM Module
USB Keychain device, Smartcard, TPM, GnuPG Keyring, etc...
PAM

Figure 1: Overview of proposed extended Cryptfs architecture

can run a utility that selects Alice's key, extracts it from the GnuPG keyring, and passes it to the Cryptfs layer, with instructions to add Alice as an authorized user for the file. The new key list in the Extended Attribute set for the file then contains two copies of the symmetric key, encrypted with different public keys:

$\{K_s\}K_u$
$\{K_s\}K_a$

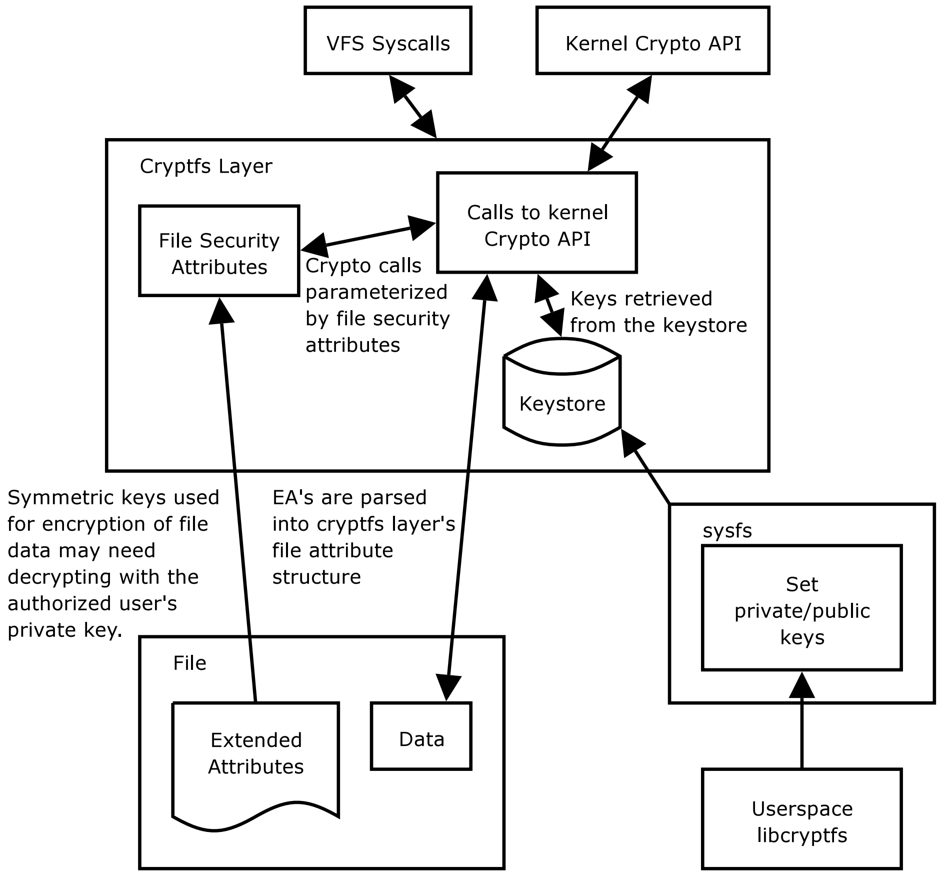
Note that this is not an access control directive; it is rather a confidentiality enforcement mechanism that extends beyond the local machine's access control. Without either the user's or Alice's private key, no entity will be able to access the decrypted contents of the file. The machine that harbors such keys will enact its own access control over the decrypted file, based on standard UNIX file permissions and/or ACL's.

When that file is copied to a removable media or attached to an email, as long as the Extended Attributes are preserved, Alice will have all the information that she needs in order to retrieve the symmetric key for the file and decrypt it. If Alice is also running Cryptfs, when she launches an application that accesses the file, the decryption process is entirely transparent to her, since her Cryptfs layer received her private key from PAM at the time that she logged in.

If the user requires the ability to encrypt a file for access by a group of users, then the user can associate sets of public keys with groups and refer to the groups when granting access. The userspace application that links against libcryptfs can then pass in the public keys to Cryptfs for each member of the group and instruct Cryptfs to add the associated key record to the Extended Attributes. Thus no special support for groups is needed within the Cryptfs layer itself.

6.1 Kernel-level Changes

No modifications to the 2.6 kernel itself are necessary to support the stackable Cryptfs layer. The Cryptfs module's logical divisions include a sysfs interface, a keystore, and the VFS operation routines that perform the encryption and the decryption on reads and writes.



VFS Syscalls
Kernel Crypto API
Cryptfs Layer
File Security Attributes
Calls to kernel Crypto API
Crypto calls parameterized by file security attributes
Keys retrieved from the keystore
Keystore
Symmetric keys used for encryption of file data may need decrypting with the authorized user's private key.
EA's are parsed into cryptfs layer's file attribute structure
sysfs
Set private/public keys
File
Extended Attributes
Data
Userspace libcryptfs

Figure 2: Structure of Cryptfs layer in kernel

By working with userspace daemon, it would be possible for Cryptfs to export public key cryptographic operations to userspace. In order to avoid the need for such a daemon while using public key cryptography, the kernel cryptographic API must be extended to support it.

6.2 PAM

At login, the user's public and private keys need to find their way into the kernel Cryptfs layer. This can be accomplished by writing a Pluggable Authentication Module, pam_cryptfs.so. This module will link against libcryptfs and will extract keys from the user's GnuPG keystore. The libcryptfs library will use the sysfs interface to the Cryptfs layer to pass the user's keys in.

6.3 libcryptfs

The libcryptfs library works with the Cryptfs's sysfs interface. Userspace utilities, such as pam_cryptfs.so, GNOME/KDE, or stand-alone utilities, will link against this library and use it to communicate with the kernel Cryptfs layer.

6.4 User Interface

Desktop environments such as GNOME or KDE can link against libcryptfs to provide users with a convenient interface through which to work with the files. For example, by right-clicking on an icon representing the file and selecting “Security”, the user will be presented with a window that can be used to control the encryption status of the file. Such options will include whether or not the file is encrypted, which users should be able to encrypt and decrypt the file (identified by their public keys from the user’s GnuPG keyring), what cipher is used, what keylength is used, an optional password that encrypts the symmetric key, whether or not to use keyed hashing over extents of the file for integrity, the hash algorithm to use, whether accesses to the file when no key is available should result in an error or in the encrypted blocks being returned (perhaps associated with UID’s - good for backup utilities), and other properties that are controlled by the Cryptfs layer.

6.5 Example Walkthrough

When a file’s encryption attribute is set, the first thing that the Cryptfs layer will do will be to generate a new symmetric key, which will be used for all encryption and decryption of the file in question. Any data in that file is then immediately encrypted with that key. When using public key-enforced access control, that key will be encrypted with the process owner’s private key and stored as an EA of the file. When the process owner wishes to allow others to access the file, he encrypts the symmetric key with the their public keys. From the user’s perspective, this can be done by right-clicking on an icon representing the file, selecting “Security→Add Authorized User Key”, and having the user specify the authorized user while using PAM to retrieve the public key for that user.

When using password-enforced access control, the symmetric key is instead encrypted using a key generated from a password. The user can then share that password with everyone who he authorized to access the file. In either case (public key-enforced or password-enforced access control), revocation of access to future versions of the file will necessitate regeneration and re-encryption of the symmetric key.

Suppose the encrypted file is then copied to a removable device and delivered to an authorized user. When that user logged into his machine, his private key was retrieved by the key retrieval Pluggable Authentication Module and sent to the Cryptfs keystore. When that user launches any arbitrary application and attempts to access the encrypted file from the removable media, Cryptfs retrieves the encrypted symmetric key correlating with that user’s public key, uses the authenticated user’s private key to decrypt the symmetric key, associates that symmetric key with the file, and then proceeds to use that symmetric key for reading and writing the file. This is done in an entirely transparent manner from the perspective of the user, and the file maintains its encrypted status on the removable media throughout the entire process. No modification to the application or applications accessing the file are necessary to implement such functionality.

In the case where a file’s symmetric key is encrypted with a password, it will be necessary for the user to launch a daemon that listens for password queries from the kernel cryptfs layer. Without such a daemon, the user’s initial attempt to access the file will be denied, and the user will have to use a password set utility to send the password to the cryptfs layer in the kernel.

6.6 Other Considerations

Sparse files present a challenge to encrypted filesystems. Under traditional UNIX semantics, when a user seeks more than a block beyond the end of a file to write, then that space is not stored on the block device at all. These missing blocks are known as “holes.”

When holes are later read, the kernel simply fills in zeros into the memory without actually reading the zeros from disk (recall that they do not exist on the disk at all; the filesystem “fakes it”). From the point of view of whatever is asking for the data from the filesystem, the section of the file being read appears to be all zeros. This presents a problem when the file is supposed to be encrypted. Without taking sparse files into consideration, the encryption layer will naïvely assume that the zeros being passed to it from the underlying filesystem are actually encrypted data, and it will attempt to decrypt the zeros. Obviously, this will result in something other that zeros being presented above the encryption layer, thus violating UNIX sparse file semantics.

One solution to this problem is to abandon the concept of “holes” altogether at the Crypfs layer. Whenever we seek past the end of the file and write, we can actually encrypt blocks of zeros and write them out to the underlying filesystem. While this allows Cryptfs to adhere to UNIX semantics, it is much less efficient. One possible solution might be to store a “hole bitmap” as an Extended Attribute of the file. Each bit would correspond with a block of the file; a “1” might indicate that the block is a “hole” and should be zero’d out rather than decrypted, and a “0” might indicate that the block should be normally decrypted.

Our proposed extensions to Cryptfs in the near future do not currently address the issues of directory structure and file size secrecy. We recognize that this type of confidentiality is important to many, and we plan to explore ways to integrate such features into Cryptfs, possibly by employing extra filesystem layers to aid in the process.

Extended Attribute content can also be sensitive. Technically, only enough information to retrieve the symmetric decryption key need be accessible by authorized individuals; all other attributes can be encrypted with that key, just as the contents of the file are encrypted.

Processes that are not authorized to access the decrypted content will either be denied access to the file or will receive the encrypted content, depending on how the Cryptfs layer is parameterized. With this behavior, it would be possible for incremental backup utilities to function properly, without requiring access to the unencrypted content of the files they are backing up.

At some point, we would like to include file integrity information in the Extended Attributes. As previously mentioned, this can be accomplished via sets of keyed hashes over extents within the file:

$$
\begin{aligned}
&H_0 = H\{O_0, D_0, K_s\}\\
&H_1 = H\{O_1, D_1, K_s\}\\
&\ldots\\
&H_n = H\{O_n, D_n, K_s\}\\
&H_f = H\{H_0, H_1, \ldots, H_n, n, s, K_s\}
\end{aligned}
$$

Where n is the number of extents in the file, s is the extent size (also contained as another EA), O_i is the offset number i within the file, D_i is the data from offset O_i to $O_i + s$, K_s is the key that one must possess in order to make authorized changes to the file, and H_f is the hash of the hashes, the number of extents, the extent size, and the secret key, to help detect when an attacker swaps around extents or alters the extent size.

Keyed hashes prove that whoever modified the data had access to the shared secret, which is, in this case, the symmetric key. Digital signatures can also be incorporated into Cryptfs. Executables downloaded over the Internet can often be of questionable origin or integrity. If you trust the person who signed the executable, then you can have a higher degree of certainty that the executable is safe to run if the digital signature is verifiable. The verification of the digital signature can be dynamically performed at the time of execution.

As previously mentioned, in addition to the extensions to the Cryptfs stackable layer, this effort is requiring the development of a cryptfs library, a set of PAM modules, hooks into GNOME and KDE, and some utilities for managing file encryption. Applications that copy files with Extended Attributes must take steps to make sure that they preserve the Extended Attributes.[7]

7 Conclusion

Linux currently has a comprehensive framework for managing filesystem security. Standard file security attributes, process credentials, ACL, PAM, LSM, Device Mapping (DM) Crypt, and other features together provide good security in a contained environment. To extend access control enforcement over individual files beyond the local environment, you must use encryption in a way that can be easily applied to individual files. The currently employed processes of encrypting and decrypting files, however, is inconvenient and often obstructive.

By integrating the encryption and the decryption of the individual files into the filesystem itself, associating encryption metadata with the individual files, we can extend Linux security to provide seamless encryption-enforced access control and integrity auditing.

[7]See `http://www.suse.de/~agruen/ea-acl-copy/`

8 Recognitions

We would like to express our appreciation for the contributions and input on the part of all those who have laid the groundwork for an effort toward transparent filesystem encryption. This includes contributors to FiST and Cryptfs, GnuPG, PAM, and many others from which we are basing our development efforts, as well as several members of the kernel development community.

9 Legal Statement

References

[1] E. Zadok, L. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. *Technical Report CUCS-021-98, Computer Science Department*, Columbia University, 1998.

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

[3] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the Annual USENIX Technical Conference*, pp. 55-70, San Diego, June 2000.

[4] S. C. Kothari, Generalized Linear Threshold Scheme, *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 231-241.

[5] Matt Blaze. "Key Management in an Encrypting File System", Proc. *Summer '94 USENIX Tech. Conference*, Boston, MA, June 1994.

[6] For more information on Extended Attributes (EA's) and Access Control Lists (ACL's), see `http://acl.bestbits.at/` or `http://www.suse.de/~agruen/acl/chapter/fs_acl-en.pdf`

[7] For more information on GnuPG, see `http://www.gnupg.org/`

[8] For more information on OpenSSL, see `http://www.openssl.org/`

[9] For more information on IBM Lotus Notes, see `http://www-306.ibm.com/software/lotus/`. Information on Notes security can be obtained from `http://www-10.lotus.com/ldd/today.nsf/f01245ebfc115aaf8525661a006b86b9/232e604b847d2cad88256ab90074e298?OpenDocument`

[10] For more information on Pluggable Authentication Modules (PAM), see `http://www.kernel.org/pub/linux/libs/pam/`

[11] For more information on Mandatory Access Control (MAC), see `http://csrc.nist.gov/publications/nistpubs/800-7/node35.html`

[12] For more information on Discretionary Access Control (DAC), see `http://csrc.nist.gov/publications/nistpubs/800-7/node25.html`

[13] For more information on the Trusted Computing Platform Alliance (TCPA), see `http://www.trustedcomputing.org/home`

[14] For more information on Linux Security Modules (LSM's), see `http://lsm.immunix.org/`

[15] For more information on Security-Enhanced Linux (SE Linux), see `http://www.nsa.gov/selinux/index.cfm`

[16] For more information on Tripwire, see `http://www.tripwire.org/`

[17] For more information on AIDE, see `http://www.cs.tut.fi/~rammer/aide.html`

[18] For more information on Samhain, see `http://la-samhna.de/samhain/`

[19] For more information on Logcrypt, see `http://www.lunkwill.org/logcrypt/`

[20] For more information on Loop-aes, see `http://sourceforge.net/projects/loop-aes/`

[21] For more information on PPDD, see `http://linux01.gwdg.de/~alatham/ppdd.html`

[22] For more information on CFS, see `http://sourceforge.net/projects/cfsnfs/`

[23] For more information on BestCrypt, see `http://www.jetico.com/index.htm#/products.htm`

[24] For more information on TCFS, see `http://www.tcfs.it/`

[25] For more information on EncFS, see `http://arg0.net/users/vgough/encfs.html`

[26] For more information on CryptoFS, see `http://reboot.animeirc.de/cryptofs/`

[27] For more information on SSHFS, see `http://lufs.sourceforge.net/lufs/fs.html`

[28] For more information on the Lightweight Auditing Framework, see `http://lwn.net/Articles/79326/`

[29] For more information on Reiser4, see `http://www.namesys.com/v4/v4.html`

[30] NFSv4 RFC 3010 can be obtained from `http://www.ietf.org/rfc/rfc3010.txt`