eCryptfs: A Stacked Cryptographic Filesystem*

Mike Halcrow
<mhalcrow@us.ibm.com>

1st May 2007

Data Confidentiality Crisis

The media has been delivering a seemingly endless stream of reports of lost or stolen laptops, backup tapes, hard drives, and servers from government and corporate facilities. These devices often contain medical, financial, and other sensitive data. When the storage devices fall into the wrong hands, attackers can access the data directly, completely bypassing the access control mechanisms in place in the organization's network. Reports indicate that millions of people have already been affected by such compromises. As a result, customers and citizens are at an increasing risk of identity fraud and loss of privacy.

While the cryptographic technology to protect data confidentiality has existed for decades, many organizations have failed to integrate this technology into their processes for handling sensitive data. In cases where cryptography is included in process, it is frequently obtrusive, costly, and complicated. Organizations sometimes neglect to establish data encryption policies, and employees often ignore such policies once they are in place.

In cases where employees attempt to utilize cryptography, they often use it ineffectively. For instance, they often select weak keys, and it is easy to inadvertently save or transfer data in unencrypted form through insecure media (such as web email or a USB flash drive). Security strategies that depend on individual applications performing their own encryption often fail when the user copies and pastes sensitive information to other applications that do not have cryptographic capability.

Data encryption needs to be made ubiquitous, transparent, flexible, easily deployable, integrated into data handling process, and, of course, secure enough to counter sophisticated attacks. These properties need to be in effect regardless of the particular applications accessing the data. To make encryption services application-agnostic, the operating system kernel itself should provide

*The content of this article was originally published on or before March 15, 2007, on page 54 of the April 2007 edition of Linux Journal magazine.

a system-wide data encryption service for sensitive information written to secondary storage.

Popular Cryptographic Filesystem Solutions

Several options exist for filesystem encryption under Linux(tm), all with various advantages and disadvantages. Device mapper crypt (dm-crypt) ships with the Linux kernel and provides block device layer encryption. Loop-AES and TrueCrypt(tm), which must be obtained separately from the official Linux kernel, also provide encryption at the block device layer. With block device layer encryption, the user creates the filesystem on the block device, and the encryption layer transparently encrypts the data before writing it to the actual lower block device.

The main advantage of block device layer encryption is that it is simple in concept and implementation. Another advantage of block device layer encryption is that an attacker learns nothing about the filesystem unless he has the key; for instance, the attacker will not even know the type of filesystem or the directory structure. Sparse files can be securely and efficiently supported in filesystems on encrypted block devices.

Block device encryption can have disadvantages that stem from the lack of integration with the filesystem itself:

- A fixed region of storage must be pre-allocated for the entire filesystem. Resizing the partition later is often an inconvenient process.
- It can be difficult to change encryption keys or ciphers.
- There is no flexibility for the block device encryption mechanism to encrypt different files with different keys or ciphers.
- Applications such as incremental backup utilities need access to the unencrypted data.
- All content in the filesystem incurs the overhead of encryption and decryption, including data that does not require secrecy.
- Files must be re-encrypted with a userspace application before they are transmitted through another medium.

EncFS is a userspace cryptographic filesystem that operates via FUSE. Userspace filesystems are easier to implement than kernel-native filesystems, and they have the advantage of being able to easily utilize userspace libraries. This makes it easy to implement feature-rich filesystems with less time and effort on the part of the developer. Unlike block device encryption solutions, EncFS operates as an actual filesystem. EncFS encrypts and decrypts individual files. Disadvantages of userspace filesystems based on FUSE include performance overhead from frequent kernel/userspace context switches and a current lack of support for shared writable memory mappings.

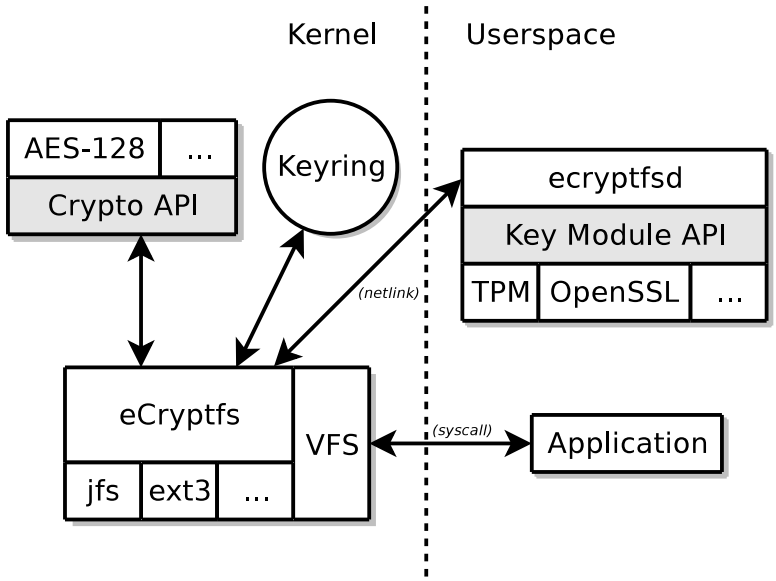

Figure 1: Application file operations go through eCryptfs, which communicates with the kernel crypto API, the kernel keyring, and the userspace eCryptfs daemon to perform encryption and decryption. eCryptfs manipulates files in lower filesystems such as jfs or ext3.

eCryptfs

eCryptfs is a kernel-native stacked cryptographic filesystem for Linux. Stacked filesystems layer on top of existing mounted filesystems that are referred to as "lower" filesystems. eCryptfs is a stacked filesystem that encrypts and decrypts the files as they are written to or read from the lower filesystem.

Applications in userspace make filesystem system calls which go through the kernel Virtual Filesystem (VFS). Both eCryptfs and the lower filesystem (e.g., ext3, jfs, nfs, etc.) are registered in the kernel VFS. The operations under the eCryptfs mount point first go to eCryptfs. eCryptfs retrieves key material from the user session keyring and uses the kernel cryptographic API to perform encryption and decryption of the file contents. eCryptfs may make key management requests with the userspace eCryptfs daemon (*ecryptfsd*). eCryptfs reads and writes encrypted content stored in files in the lower filesystem (see Figure 1).

eCryptfs aims to provide the flexibility of a Pretty Good Privacy (PGP) application as a transparent kernel service. For that reason, the OpenPGP (RFC 2440) specification inspires the basic key handling techniques in eCryptfs. This includes the common procedure of using a hierarchy of keys when performing cryptographic operations (see Figure 2).

The cryptographic metadata is in the header region of the encrypted lower

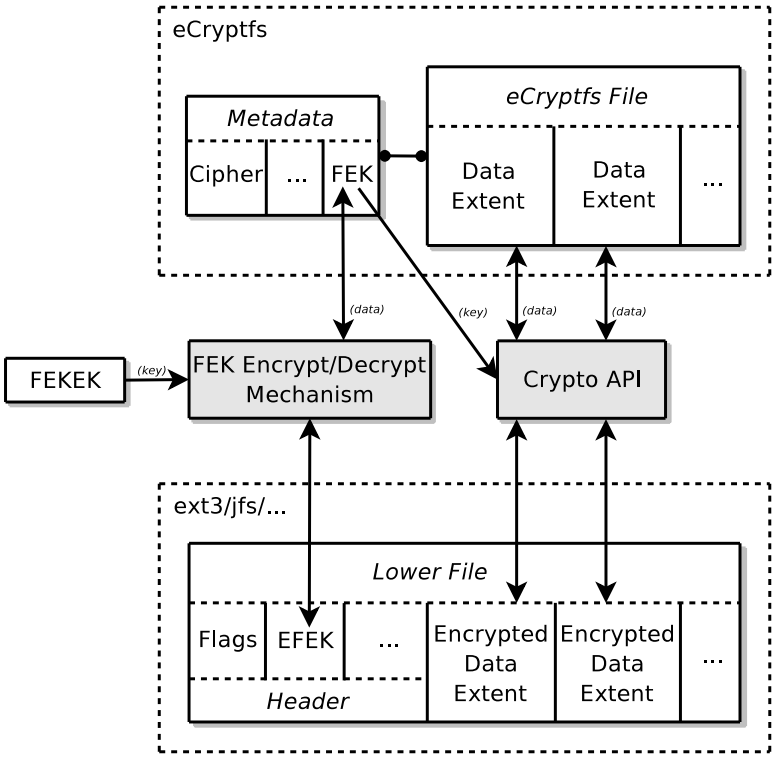

Figure 2: eCryptfs encrypts and decrypts individual data extents in each file using a unique randomly generated File Encryption Key (FEK). The FEK is encrypted with the File Encryption Key Encryption Key (FEKEK), and the resulting Encrypted File Encryption Key (EFEK) is stored in the header of each lower file.

```
Code maturity level options  --->
  [*] Prompt for development and/or incomplete code/drivers

Security options  --->
  <M> Enable access key retention support

Cryptographic options  --->
  <M>   MD5 digest algorithm
  <M>   AES cipher algorithms

File systems  --->
  Miscellaneous filesystems  --->
    <M> eCrypt filesystem layer support (EXPERIMENTAL)
```

Figure 3: Kernel options needed for eCryptfs.

file. The user can transmit the lower file as-is to another user, and the recipient can access the decrypted contents of the file through eCryptfs, so long as he has the proper key. This provides a high degree of flexibility in how the files can be handled while maintaining strong security.

Using eCryptfs

eCryptfs requires a kernel component and a userspace component. The kernel component ships in the current mainline Linux kernel. See Figure 3 for the minimum kernel options necessary to enable eCryptfs. By default, eCryptfs will use the AES cipher. eCryptfs can use other ciphers available in the kernel if you build them.

Newer versions of the Linux kernel contain more feature-rich versions of eCryptfs. For instance, Linux kernel version 2.6.19 is the first official kernel version that contains eCryptfs, and only passphrase mode of operation is available in that kernel. As of the writing of this article, the development kernel branch version 2.6.20-mm contains public key support, and so that feature may be now available in more recent mainline kernel versions. You can determine the features available in your kernel by loading the *ecryptfs* module and viewing the contents of *fs/ecryptfs/version_str* under your sysfs mount point.

Popular Linux distributions carry the eCryptfs userspace packages; follow the software package installation procedure for your distribution to install the *ecryptfs-utils* package. If the eCryptfs userspace tools are not yet available from your distribution, then you can download, build, and install the source tarball. You can obtain the userspace components from the eCryptfs SourceForge site `http://ecryptfs.sourceforge.net`.

If eCryptfs is built as a kernel module, then you will need to load the module:

```
# modprobe ecryptfs
```

At this point, you can begin using eCryptfs with whatever filesystem you are currently using. To mount eCryptfs, specify the lower directory for the encrypted files and the eCryptfs mount point for the decrypted view of the files:

```
# mount -t ecryptfs /secret /secret
```

The first path is the lower directory, and the second path is the eCryptfs mount point. Note that the lower directory and the mount point have the same path in this example. These paths can be different, but I recommend doing a layover mount in order to help assure that only eCryptfs has access to the files in the lower filesystem. This command transforms the given path from the lower directory into the eCryptfs mount point for the duration of the mount.

When performing a mount, the eCryptfs mount helper will first attempt to read in options from the .ecryptfsrc file in the current user home directory, and then it will read options provided via the command line. The mount helper will interactively prompt for any mandatory options that are not specified in the .ecryptfsrc file or the command line. For instance, you may be asked to choose a passphrase and a cipher.

Once the mount has successfully completed, files written to the */secret* mount point will be transparently encrypted and written to the */secret* directory in the lower filesystem. Encrypted files that exist in the lower */secret* directory and that are able to be decrypted with the key specified at the time of the mount will be accessible in their unencrypted form when read from the */secret* eCryptfs mount point.

When you unmount eCryptfs and look in */secret*, you will see the encrypted lower files. You may first notice that the lower files are larger than the files viewed under the eCryptfs mount point. The exact size of the lower files will depend on the page size of your host and on the amount of data written. In general, the minimum lower filesize is either 12KB or your host page size plus 4KB, whichever is larger. This helps ensure page alignment between the eCryptfs file and the lower file, which helps performance. The lower file then grows in 4KB increments as data spills into new 4KB data extents.

The extra space at the front of each lower file contains cryptographic metadata about the file, such as attribute flags and an encrypted file encryption key. Having this information in the file contents makes it convenient to transfer or backup the files while preserving all the information necessary to access the files later. However, the headers can take up a disproportionally large amount of space if there are many small files. Newer releases of eCryptfs can store the data in the extended attribute region instead, reducing the size of the lower encrypted files; refer to the eCryptfs online documentation at `http://ecryptfs.sourceforge.net` for more information on using this feature.

If your kernel has public key support, you can utilize one of the eCryptfs key modules to manage your key. You can check for support in the version of eCryptfs in your kernel by viewing the contents of *fs/ecryptfs/version_str* under your sysfs mount point; if there is support, you will see "pubkey" listed as one of the supported features.

Key modules can be selected and parameterized via mount options. If you wish to use the OpenSSL key module, first you need to generate a public/private keypair to use in eCryptfs. To generate a keypair:

- Run *ecryptfs-manager*.
- Select menu option '3'.
- Select the 'openssl' key module.

You also need to run the eCryptfs daemon in order to manage kernel-userspace communications; the daemon can be started by simply running the executable:

```
# ecryptfsd
```

Note that running the daemon is not necessary if you are only using the passphrase mode of operation. Then, assuming you created your key in */usb-drive/mykey.pem*, you would mount with the following options:

```
# mount -t ecryptfs \
  -o key=openssl:keyfile=/usb-drive/mykey.pem \
  /secret /secret
```

Given these options, the eCryptfs mount helper will prompt you for a passphrase that protects the private key contained in the key file.

You can mount the same lower directory with many different combinations of keys and ciphers (known as a "mount context"); that particular context will apply to any new files created under the mount point. For current versions of eCryptfs, files created under any given mount context will only be accessible when the mount is performed with that same context.

Notes on Security

As with any filesystem, you should make regular backups of your data when using eCryptfs. This is easily and securely done by unmounting eCryptfs and reading the lower encrypted files.

eCryptfs only protects the confidentiality of data at rest that is outside the control of the trusted host environment. You should properly use access control mechanisms such as SELinux on the trusted host in order to regulate access to the decrypted files.

eCryptfs will, by default, preserve all of the information necessary to access the decrypted contents of the files in the contents of the lower files themselves. All that is required is the key used to create the files in the first place. You should take measures to protect this key. If applications such as incremental backup utilities are configured to only read the lower encrypted files, then these utilities do not need to apply any further encryption to the files in order to ensure data confidentiality.

If you are using a passphrase, follow common best practices in selecting and protecting your passphrase (for instance, see `http://www.iusmentis.com/security/passphrasefaq/`). I recommend using the public key mode of operation instead of passphrase whenever possible. When using a public key module, make a backup copy of your key file and store it in a physically secure location. Should you lose your key, nobody will be able to retrieve your data. Do not store unprotected copies of your passphrase or your public key file on the same media as your encrypted data.

You are free to choose among the symmetric encryption ciphers that are available through the Linux kernel cryptographic API. eCryptfs will recommend AES-128 as the default cipher. If you have hardware acceleration available on your machine and if it is supported by the selected cipher in the kernel cryptographic API, then eCryptfs encryption and decryption operations will automatically be hardware accelerated.

You should take measures to ensure that sensitive data is not written to secondary storage in unencrypted form. Applications that write out sensitive temporary data should be configured so that they only write under an eCryptfs mount point. You should also use dm-crypt to encrypt the swap space with a random key. The details are beyond the scope of this article, but commands to set it up will take the form:

```
# cryptsetup -c aes-cbc-plain -d /dev/random create \
  swap /dev/SWAPDEV
# mkswap /dev/mapper/swap
# swapon -p 1 /dev/mapper/swap
```

SWAPDEV is the swap block device on your machine (refer to your */etc/fstab* file if you are not sure which device is currently used for swap). You can create simple boot scripts to automatically set up the encrypted swap space, run *ecryptfsd*, and perform eCryptfs mounts. Consult your distribution's documentation for more details on writing boot scripts and using dm-crypt with a random key to encrypt your swap space.

Note that current releases of eCryptfs only encrypt the file contents. Metadata about the file – for instance, the size, the name, permissions, and extended attributes – are all readable by anyone with access to the lower encrypted file. Future work on eCryptfs will include encryption or obfuscation of some of this metadata.

Using block device encryption together with eCryptfs can combine the security provided by both mechanisms while offering the flexibility of having seamless access to individual encrypted lower files, although this will roughly double the processing overhead of encrypting and decrypting the data. If only the contents of the files on secondary storage require confidentiality, then eCryptfs by itself is, in most cases, sufficient.

The Stony Brook University (SUNY) File Systems and Storage Labs (FSL) (`http://filesystems.org/`) has developed a stacked filesystem framework called FiST. eCryptfs is derived from Cryptfs, which is one of the example filesystems implemented in FiST. Unionfs is another popular stacked filesystem written by the SUNY FSL.

Future Work

eCryptfs was designed to support a host of advanced key management and policy features. The development roadmap for eCryptfs includes multiple keys per file, different keys and ciphers for different files depending on the application creating the file and the location where the file is being written, integrity enforcement, and more extensive interoperability with existing key infrastructures and key management devices. These features will become available as they are implemented in future versions of the Linux kernel.

Conclusion

eCryptfs is a flexible kernel-native solution that cryptographically enforces data confidentiality on secondary storage devices. eCryptfs can be deployed on existing filesystems with minimal effort. The individual encrypted files can be transferred to other hosts running eCryptfs and transparently accessed using the proper key. The eCryptfs key management mechanism is highly extensible. eCryptfs is suitable to use as a strong and convenient data confidentiality enforcement component to help secure data managed in Linux environments.

Legal Statement

About The Author

Mike Halcrow <mhalcrow@us.ibm.com> is a Security Software Engineer at the IBM Linux Technology Center and is the lead architect and developer of eCryptfs. He is also pursuing a Master's degree in Computer Science at UT Austin. In the past, he has maintained the openCryptoki PKCS#11 application, contributed to Common Criteria CAPP/EAL security certification efforts for Linux, and authored the BSD Secure Levels Linux Security Module (LSM) that shipped in previous versions of the Linux kernel.