SecurityFocus™

Infocus

< http://www.securityfocus.com/infocus/1673 >

Cryptographic Filesystems, Part One: Design and Implementation

by Ido Dubrawsky
last updated March 7, 2003

As security becomes a greater focus in networks, every aspect of online information needs a level of protection from the network-level use of firewalls and IDS to the host-level use of IDS. However, an additional level of security has recently come to the forefront of security - cryptographic filesystems. While the technology for cryptographic filesystems has been available for quite a while, the deployment of cryptographic filesystems in production environments has not taken hold. This article will discuss some of the background and technology of cryptographic filesystems and will then cover some example implementations of these filesystems including Microsoft's Encrypting File System for Windows 2000, the Linux CryptoAPI, and the Secure File System.

Cryptographic Filesystem Types

There are several approaches to cryptographic filesystems. These vary from volume encryptors to filesystem encryptors to file encryptors. Each of these approaches has its merits as well as its drawbacks and is discussed in more detail below.

Volume Encryptors

Volume encryptors use the device driver layer to encrypt and decrypt information to and from a physical disk. Such systems include systems as PGPDisk, the Secure File System (SFS), the Linux CryptoAPI and ScramDisk. Volume encryptors encrypt whole drives and are convenient to use since they are transparent to the end user. However, volume encryptors do not provide fine-grained access control to individual directories or files.

File Encryptors

File encryptors operate at the application or presentation layer to provide true end-to-end file encryption. In order to provide some sort of transparency to the end-user file, encryptors typically require some measure of application rewrite in order to support encryption. File encryptor systems include such tools as PGP. For small numbers of files these types of encryptors are adequate but they do not scale well to storage systems.

File System Encryptor

File system encryptors allow the encryption of files on a per-file or a per-directory basis using a single key. Systems such as these include the Cryptographic File System (CFS) developed by Matt Blaze, the Transparent Cryptographic File System (TCFS) supported under Linux and BSD, CryptFS, as well as Microsoft's EFS under Windows 2000.

Cryptographic Filesystem Design

CryptFS

One implemention of a cryptographic filesystem is through the use of a kernel-resident filesystem. This implementation model is used in CryptFS. Using this implementation model, the file system can be mounted on any directory as well as on top of another file system such as UFS or NFS[1]. This model also removes the need for additional daemon processes that can possibly be exploited to gain access to the system or possibly to the files. The interface used by CryptFS is through a stackable V-node (Virtual Node). Unix-based operating systems use vnodes to represent an open file, directory, device, or other objects. The vnodes do not expose what type of physical file system they implement. CryptFS uses a concept called V-node stacking, which allows for filesystem function modularization where one V-node interface calls another. V-node stacking is shown in Figure 1 below.

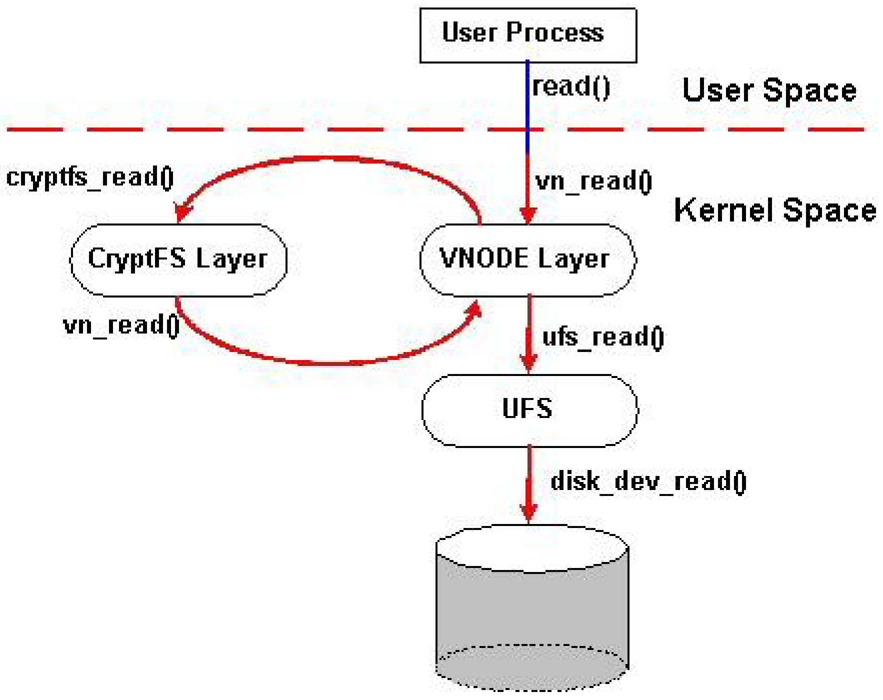


Figure 1: Stackable V-node Interface Design

Each V-node operation in CryptFS calls its next-lower layer function for filesystem specific operations. Originally developed for Solaris, CryptFS has also been ported to Linux and FreeBSD. CryptFS inserts itself on top of any directory, encrypts file data before it is passed to the file system, and decrypts it in the reverse direction. The designers of CryptFS also provide for a key management scheme where only the root user can actually mount an instance of CryptFS and user keys would be associated not only with a UID but also a session ID. An attacker would not only have to break into an account but also have his processes use the same session ID as the users process in order to acquire or change a user's key. CryptFS uses blowfish in CBC mode as its encryption algorithm.

The CryptFS is available to the general public as part of the File Systems Software Package (fistgen). It can be found here.

Linux CryptoAPI

While the Linux CryptoAPI is not, strictly speaking, a filesystem encryption system, it does provide for that capability. The original kernelint patch was designed to provide filesystem encryption capabilities in the 2.2 series kernels. The development of the kernelint patch into a general purpose API for kernel-space encryption represents the next logical evolutionary step. The loopback device allows for a level of indirection when mounting a filesystem whereby system calls can be intercepted to provide encryption and decryption of filesystem data. Instead of mounting the filesystem directly on a given directory the filesystem can be mounted on the loopback device. The loopback device is then mounted on the directory mount point. This configuration has the effect of sending all kernel commands to the filesystem through the loopback device. The process is shown in Figures 2 and 3.

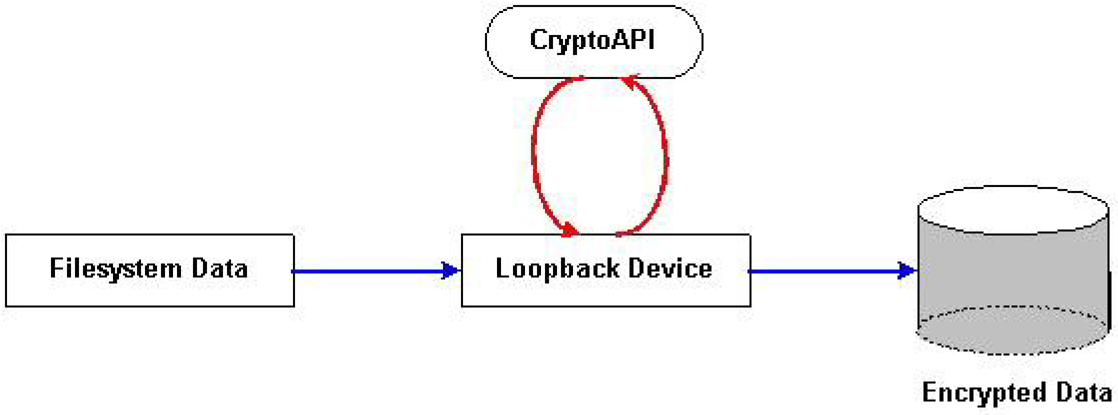


Figure 2: CryptoAPI Encryption

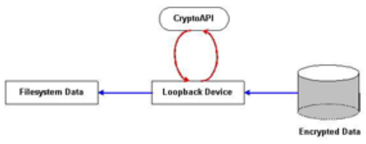


Figure 3: CryptoAPI Decryption

The CryptoAPI software can be downloaded from the GNU/Linux CryptoAPI site.

Microsoft's Encrypted FileSystem (EFS)

Microsoft's EFS is implemented using a public key-based scheme. File data is encrypted using a fast symmetric algorithm with a file encryption key that is randomly generated. This key is encrypted itself using one or more public keys obtained from a user's X.509 version 3 certificate. The private portion of the private-public key is used to decrypt the file encryption key, which is then used to decrypt the file. Microsoft's EFS does not support using a symmetric algorithm using a password-based key because of the concern that these password based schemes are weaker due to their susceptibility to dictionary attacks. EFS also provides for the encryption of the File Encryption Key with one or more recovery key public keys, this allows for the possibility of the recovery of the file data should an individual leave an organization or lose their key. The encryption and decryption processes are shown in Figures 4 and 5 below.

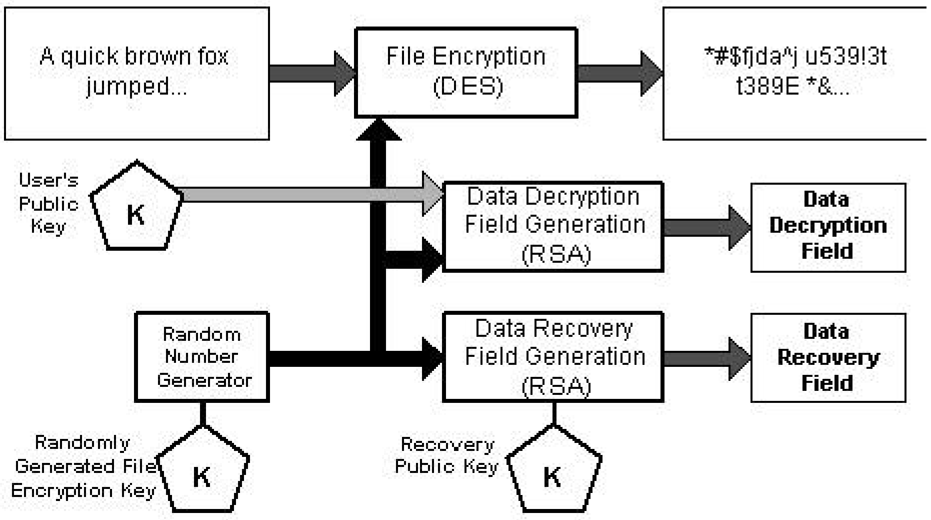

A quick brown fox jumped...
File Encryption (DES)
*#$fjda^j u539!3t t389E *&...
User's Public Key
K
Data Decryption Field Generation (RSA)
Data Decryption Field
Random Number Generator
Data Recovery Field Generation (RSA)
Data Recovery Field
Randomly Generated File Encryption Key
K
Recovery Public Key
K

Figure 4: Microsoft EFS Encryption (from [3])

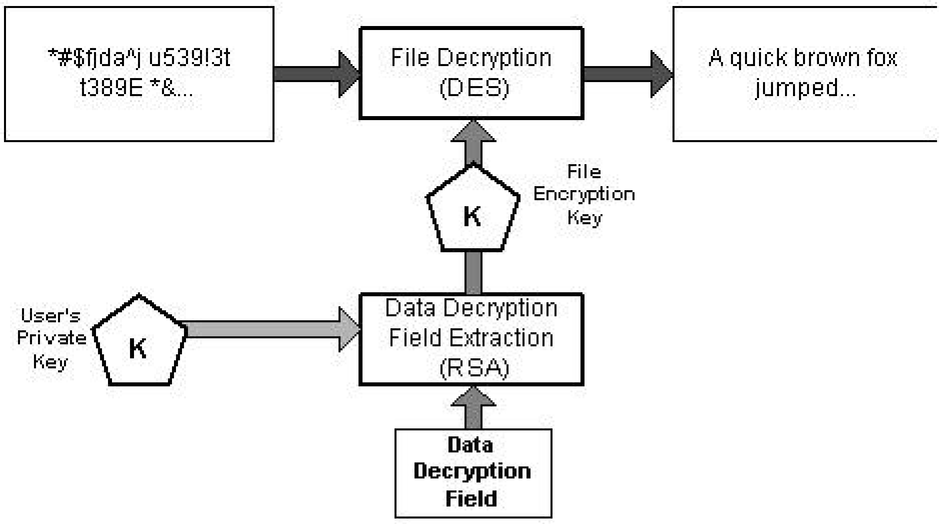

*#$fjda^j u539!3t t389E *&...
File Decryption (DES)
A quick brown fox jumped...
File Encryption Key
K
User's Private Key
K
Data Decryption Field Extraction (RSA)
Data Decryption Field

Figure 5: Microsoft EFS Decryption (from [3])

EFS is designed to be transparent to the end-user under normal circumstances. When a user attempts to access one of their encrypted files, the EFS will locate the private key used to encrypt the File Encryption Key that encrypted the file, decrypt the File Encryption Key, and decrypt the file. If an attempt is made to access a file encrypted by another user, EFS will fail to find the private key that can be used to decrypt the File Encryption Key and the user will be presented with an "Access Denied" condition.

Microsoft's implementation of EFS is shown in Figure 6. This architecture is broken up into four primary categories: the EFS driver, FileSystem Run-Time Library, the EFS service, and various Win32 APIs. The driver is layered on top of the NTFS filesystem. It communicates with the EFS service to request file encryption keys and other services. The driver passes this information to the run-time library in order to perform various file system operations. The EFS driver and run-time library do not communicate directly but rather use the NTFS file control call-out mechanism to communicate. The EFS service is part of the security subsystem and uses the port between the Local Security Authority (LSA) and the kernel-mode security reference monitor in order to communicate with the EFS driver. The EFS service interfaces with the CryptoAPI (not the same CryptoAPI for Linux) in user-mode to provide file encryption keys and other services. It also provides support for Win32 APIs that provide the programming interfaces for encrypting, decrypting or recovering files, as well as the importing and exporting of encrypted files.

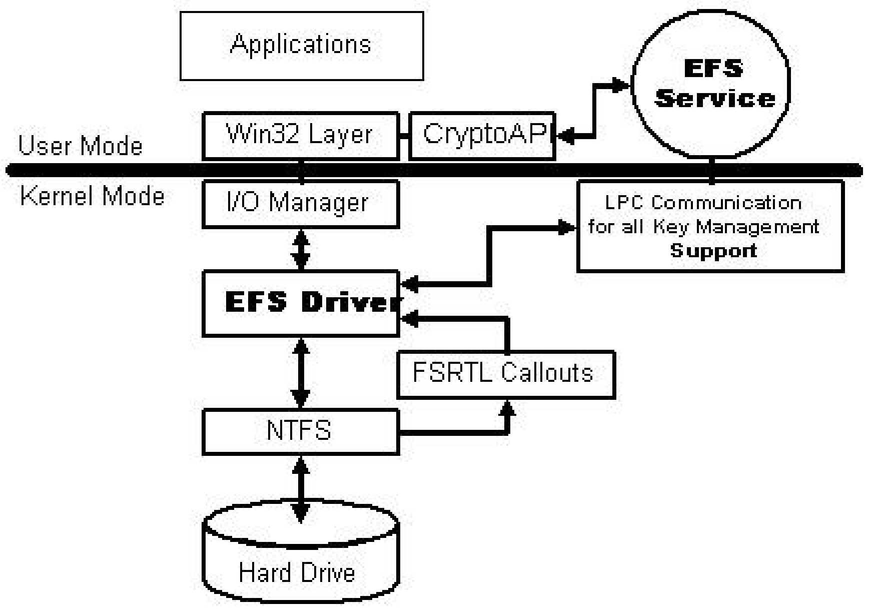


Figure 6: EFS Architecture (from [3])

EFS currently only supports the DESX encryption algorithm, which is based on a 128-bit encryption key. Microsoft says that future releases of EFS will support alternate encryption algorithms.

Summary

Cryptographic filesystems have made significant progress in the past few years. The capability of providing transparent encryption services to users in order to protect the data stored on a filesystem has become more enticing as networks are increasingly coming under attack from both external and internal sources. One of the key tenets of a solid security policy is the capability to ensure the integrity of data stored on network systems. Cryptographic filesystems make this increasingly possible by easing the effort of encrypting files.

References

[1] Zadok, Erez, Ion Badulescu and Alex Shender, "Cryptfs: A Stackable Vnode Level Encryption File System", CUCS-021-98, http://www.cs.columbia.edu/~ezk/research/cryptfs/cryptfs.pdf, 1998

[2] Bryson, David, "Using CryptoAPI", http://www.kerneli.org/howto/node3.php, 2002.

[3] Microsoft, "Encrypting File System for Windows 2000", http://www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp, 1998.

Infocus
< http://www.securityfocus.com/infocus/1685 >

Cryptographic File Systems, Part Two: Implementation

by Ido Dubrawsky
last updated April 14, 2003

This is the second article in a two-part series looking at cryptographic filesystems. The first article in this series covered the background on cryptographic filesystems from the underlying concepts to some of the mechanics of those systems. This article will cover implementation. The focus will be on implementing the Microsoft's EFS under Windows 2000 and the Linux CryptoAPI.

One point to clarify from the first article involves the note that Microsoft's EFS does not support using a password-based symmetric algorithm. This is due to the concern that such schemes are weaker because of their susceptibility to dictionary attacks. While technically accurate, the fact remains that the public portion of the user's X.509v3 certificate (which is used to encrypt the File Encryption Key, or FEK, used by EFS) is used to encrypt the FEK. To decrypt the FEK requires the use of password or passphrase and unless password-based logon is disabled completely this password or passphrase is typically the user's domain password.

Microsoft Windows 2000 EFS

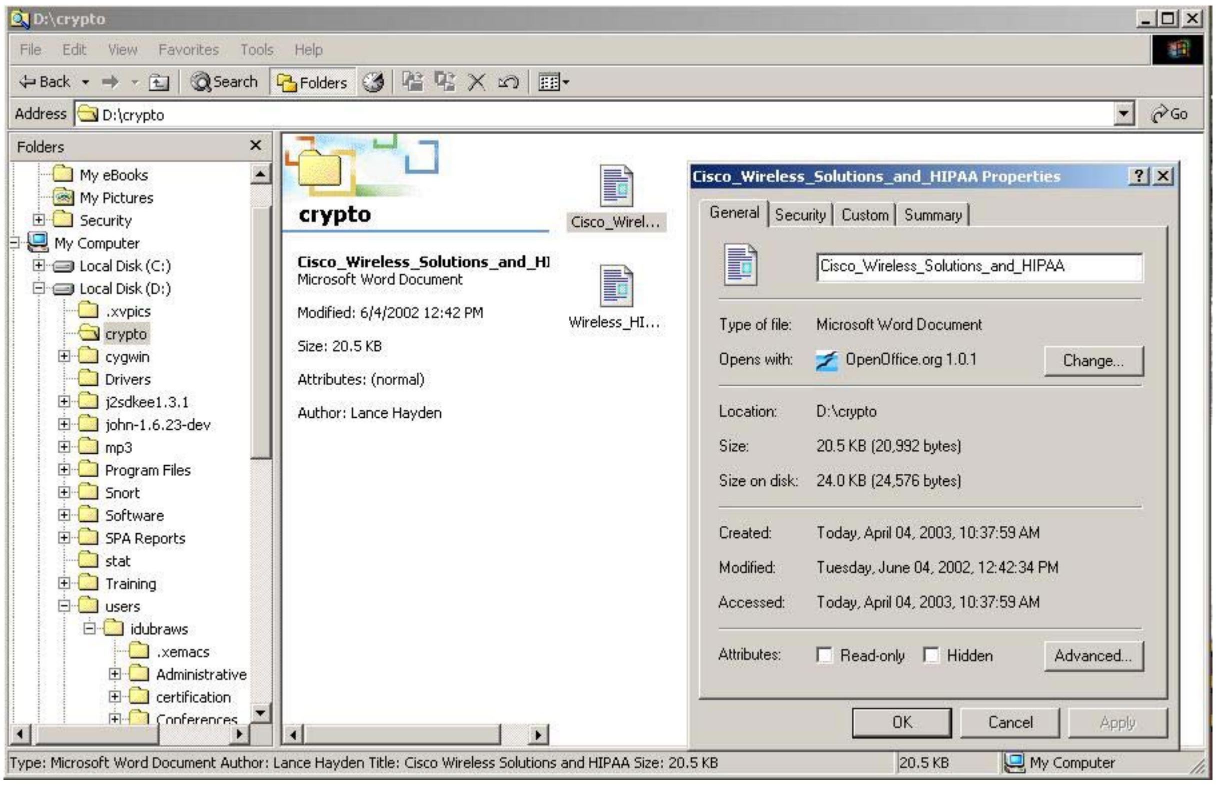
Microsoft's Windows 2000 product introduced the encrypting file system (EFS) to the Windows product line. While third party add-on encrypting software has been available for some time, EFS was the integration of such a concept into Windows 2000. Under Windows 2000, EFS supports the DESX algorithm only. With Windows XP that encryption algorithm now includes 3DES as well and will eventually include the Advanced Encryption Standard (AES) algorithm.

Set-Up

Using EFS is very easy. Windows 2000 supports both a command line method as well as a graphical method of encrypting files with EFS. Encrypting individual files and folders is very easy under Windows 2000. The tricky part is when those files and folders are moved either to different computers or stored on another computer because the appropriate certificate and private keys must also be stored with the files on that computer in order to be able to decrypt the files.

Encrypting/Decrypting a File: Graphical Method

Encrypting a file is as simple as clicking on the file with the right mouse button and opening up the file properties as shown below.



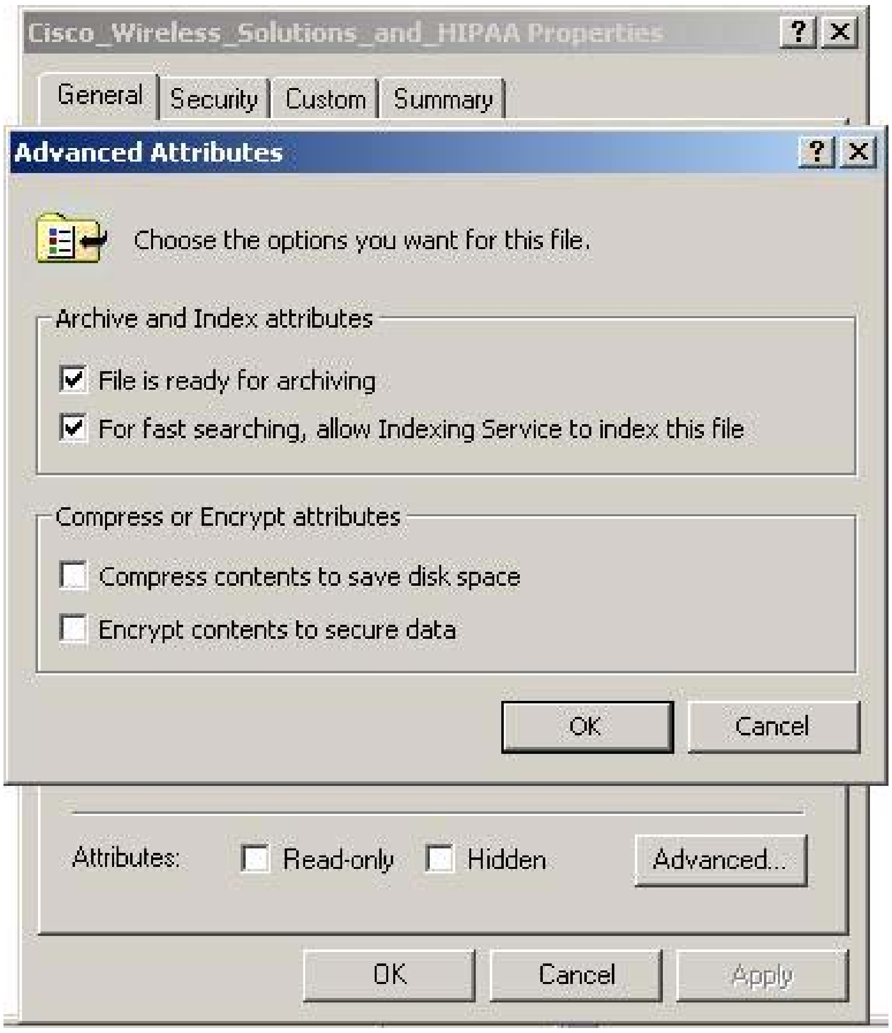
Cisco_Wireless_Solutions_and_HI
Microsoft Word Document
Modified: 6/4/2002 12:42 PM
Size: 20.5 KB
Attributes: (normal)
Author: Lance Hayden

Cisco_Wireless_Solutions_and_HIPAA

Type of file:	Microsoft Word Document
Opens with:	OpenOffice.org 1.0.1
Location:	D:\crypto
Size:	20.5 KB (20,992 bytes)
Size on disk:	24.0 KB (24,576 bytes)
Created:	Today, April 04, 2003, 10:37:59 AM
Modified:	Tuesday, June 04, 2002, 12:42:34 PM
Accessed:	Today, April 04, 2003, 10:37:59 AM
Attributes:	☐ Read-only ☐ Hidden

Type: Microsoft Word Document Author: Lance Hayden Title: Cisco Wireless Solutions and HIPAA Size: 20.5 KB 20.5 KB

Next, select the "Advanced" button at the bottom right of the properties panel to bring up the "Advanced Attributes" window. At the bottom of this window is a checkbox entitled "Encrypt Contents to secure data". Select the checkbox and click "OK".



Choose the options you want for this file.

Archive and Index attributes

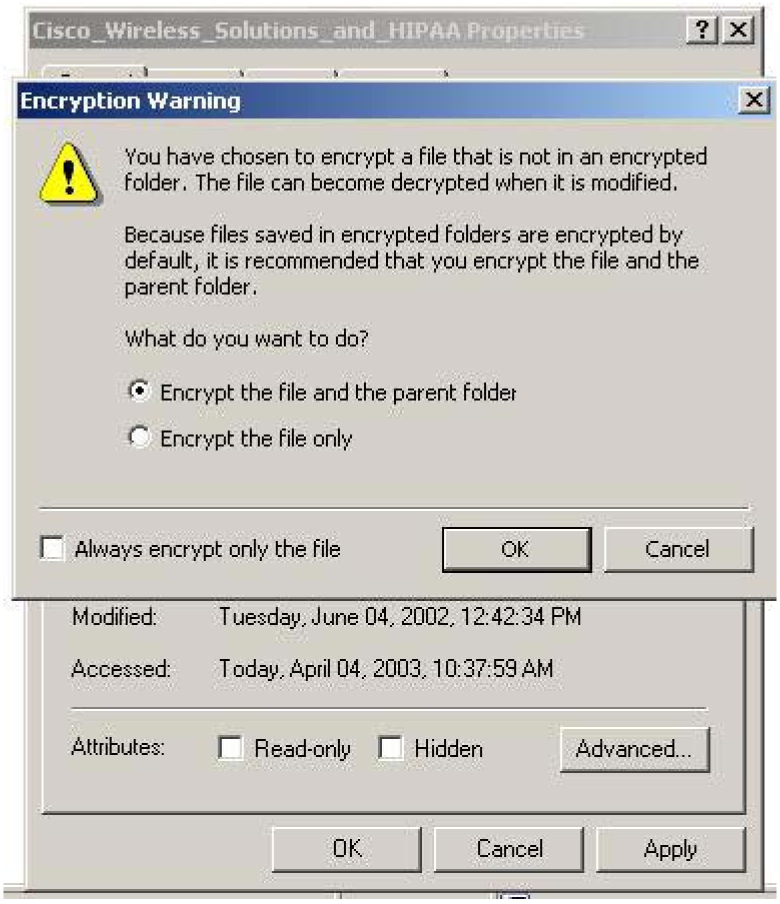
- [✓] File is ready for archiving
- [✓] For fast searching, allow Indexing Service to index this file

Compress or Encrypt attributes

- [ ] Compress contents to save disk space
- [ ] Encrypt contents to secure data

Attributes: [ ] Read-only [ ] Hidden

When the "Advanced Attributes" windows has closed, select "Apply" at the lower right of the properties window. This will bring up a new dialog window up asking if you want to encrypt only the selected file or the selected file and its parent folder. Select one of these two options and click the "OK" button at the lower right.


Cisco_Wireless_Solutions_and_HIPAA Properties

Encryption Warning

You have chosen to encrypt a file that is not in an encrypted folder. The file can become decrypted when it is modified.

Because files saved in encrypted folders are encrypted by default, it is recommended that you encrypt the file and the parent folder.

What do you want to do?

[✓ Encrypt the file and the parent folder]

Encrypt the file only

Always encrypt only the file

Modified: Tuesday, June 04, 2002, 12:42:34 PM

Accessed: Today, April 04, 2003, 10:37:59 AM

Attributes: Read-only Hidden

Once the file has been encrypted it can still be accessed just like any other file. The file does not have to be decrypted to view it because the entire encryption/decryption process is automatic and transparent. To decrypt a file completely (that is where it is not stored on the disk encrypted) simply open the file properties panel by right clicking on the file or folder. Select the Advanced Attributes button and de-select the "Encrypt Contents to secure data" checkbox. Apply the properties to the file and it reverts back to a normal, unencrypted file which can be viewed by other users (depending on the NTFS attributes of the file).

Encrypting/Decrypting a File: Command Line Method

Encrypting and decrypting a file using the Windows command line is easier than with the graphical method. The **cipher.exe** program is used to encrypt/decrypt a file or a folder and its contents. An example is shown below. To encrypt a file the command is:

```
c:cipher.exe /e "D:\crypto\file.txt"
```

To encrypt a folder and all of its subfolders and files:

```
c:cipher.exe /e /s "D:\crypto"
```

To decrypt, simply change the **/e** flag to a **/d** as shown below:

```
c:cipher.exe /d /s "D:\crypto"
```

Remember, to just decrypt a file don't use the **/s** flag.

There are significantly more complex challenges with regards to EFS when files are moved from one computer to another. In order to provide encryption/decrytion capabilities with the same FEK on different computers, users will have to export their keys using the Microsoft Management Console on one system and import them on other systems. Microsoft provides a step-by-step guide to using EFS including how to export and import FEKs. It is available here.

Linux CryptoAPI Implementation

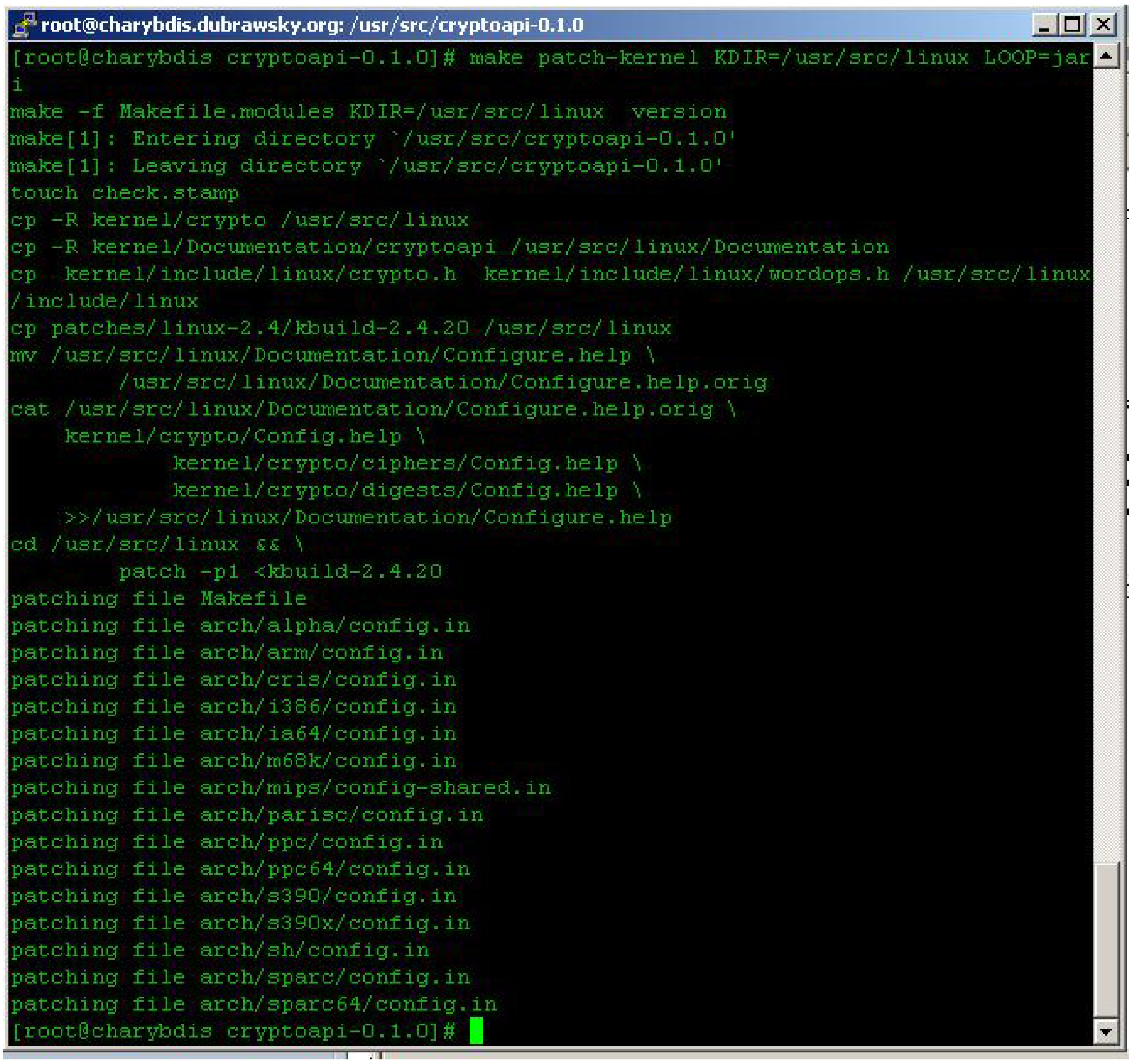
Implementing a cryptographic filesystem under Linux is not much harder than the implementation under Windows 2000. Three (and possibly four, depending on your Linux software) items are required to implement CryptoAPI under Linux -- a current version of the Linux Kernel the CryptoAPI source code and the cryptoloop software. Both of these should be downloaded, their signatures checked, and then unpacked. The CryptoAPI version should match relatively closely, if not exactly, with the kernel version.

```
[root@charybdis cryptoapi-0.1.0]# bzip2 -dc /opt/software/kernel/linux-2.4.20.tar.bz2 | tar -
xvf -
[root@charybdis cryptoapi-0.1.0]# bzip2 -dc cryptoapi-0.1.0.tar.bz2 | tar -xvf -
[root@charybdis cryptoapi-0.1.0]# bzip2 -dc cryptoloop-0.0.1-pre1.tar.bz2 | tar -xvf -
```

Two patches to the loopback filesystem device driver are provided in the CryptoAPI sources. The first is the loopiv (iv stands for initialization vector) patch, which provides minimal support for the CryptoAPI, and the second is the loop-jari patch, which also provides support for CryptoAPI as well as some bug fixes.

Once the kernel and the CryptoAPI source code have been unpacked, you will need to configure the kernel *before* applying the CryptoAPI patches. The process can be done for an existing kernel by using the current configuration in place. The following instructions are for building a new kernel with the CryptoAPI. Once the kernel has been configured you can apply the CryptoAPI patches using the command:

```
[root@charybdis cryptoapi-0.1.0]# make patch-kernel KDIR=< kernel dir > LOOP= < iv|jari
>
```



```
[root@charybdis cryptoapi-0.1.0]# make patch-kernel KDIR=/usr/src/linux LOOP=jar
i
make -f Makefile.modules KDIR=/usr/src/linux  version
make[1]: Entering directory `/usr/src/cryptoapi-0.1.0'
make[1]: Leaving directory `/usr/src/cryptoapi-0.1.0'
touch check.stamp
cp -R kernel/crypto /usr/src/linux
cp -R kernel/Documentation/cryptoapi /usr/src/linux/Documentation
cp  kernel/include/linux/crypto.h  kernel/include/linux/wordops.h /usr/src/linux
/include/linux
cp patches/linux-2.4/kbuild-2.4.20 /usr/src/linux
mv /usr/src/linux/Documentation/Configure.help \
        /usr/src/linux/Documentation/Configure.help.orig
cat /usr/src/linux/Documentation/Configure.help.orig \
    kernel/crypto/Config.help \
            kernel/crypto/ciphers/Config.help \
            kernel/crypto/digests/Config.help \
    >>/usr/src/linux/Documentation/Configure.help
cd /usr/src/linux && \
        patch -p1 <kbuild-2.4.20
patching file Makefile
patching file arch/alpha/config.in
patching file arch/arm/config.in
patching file arch/cris/config.in
patching file arch/i386/config.in
patching file arch/ia64/config.in
patching file arch/m68k/config.in
patching file arch/mips/config-shared.in
patching file arch/parisc/config.in
patching file arch/ppc/config.in
patching file arch/ppc64/config.in
patching file arch/s390/config.in
patching file arch/s390x/config.in
patching file arch/sh/config.in
patching file arch/sparc/config.in
patching file arch/sparc64/config.in
[root@charybdis cryptoapi-0.1.0]#
```

[root@charybdis cryptoloop-0.0.1-pre1]# make patch-kernel KDIR=< kernel dir > LOOP= < iv|jari>

Once the patches have been applied, change to the kernel source directory, build the kernel and install it. With the new kernel built and installed it is recommended that the system be rebooted with the new kernel. After the new kernel is functioning build the CryptoAPI modules using the following syntax:

[root@charybdis cyrptoapi-0.1.0]# make modules KDIR=< kernel dir >

Once the CryptoAPI modules are built, install them:

[root@charybdis cyrptoapi-0.1.0]# make modules_install

One final piece of the puzzle is a patch for **losetup** program. Depending on the Linux distribution the losetup program may or may not need to be patched. This patch provides the losetup program the ability to tell the kernel which cipher and key to use for the encrypted devices that will be configured. The patch is available at SourceForge. Some newer Linux versions contain an losetup with the patch already installed.

Once the system is set up it is very easy to bring the CryptoAPI into play. What needs to be determined is how the encrypted filesystem will be implemented. There are two ways to handle this: a physical filesystem or a virtual filesystem. A physical filesystem would entail creating an encrypted filesystem on a disk partition. A virtual filesystem is essentially a very large empty file that can be mounted through the loopback device. The example below focuses on the latter. Creating an empty file simply requires setting aside space within a physical filesystem as a large file. To do this:

```
[root@charybdis cyrptoapi-0.1.0]# dd if=/dev/zero if=/oracle/testfs bs=1M count=50
```

It is just as valid to use /dev/urandom to fill the empty file with random data. Once that is done, the next step is to load the cryptoapi, cryptoloop, and the cipher kernel modules.

```
[root@charybdis cyrptoapi-0.1.0]# modprobe cryptoloop
[root@charybdis cyrptoapi-0.1.0]# modprobe cryptoapi
[root@charybdis cyrptoapi-0.1.0]# modprobe cipher-des
```

Once that is done the filesystem device can be mounted through the loopback device:

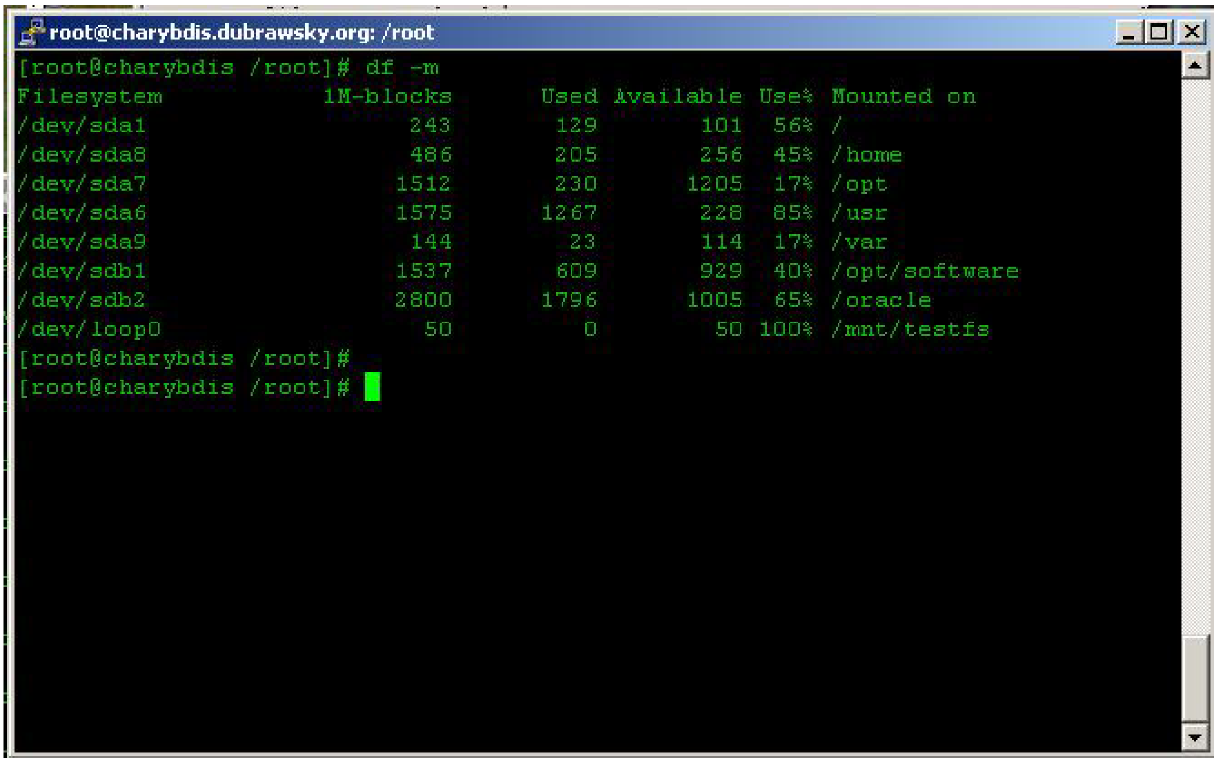
```
[root@charybdis cyrptoapi-0.1.0]# losetup -e des /dev/loop0 /oracle/testfs
Password:
Init (up to 16 hex digits):
```

Once the loopback device is set up the filesystem needs to be made on it:

```
[root@charybdis cyrptoapi-0.1.0]# mkfs -t ext3 /dev/loop0
```

Now the filesystem is ready to be mounted:

```
[root@charybdis cyrptoapi-0.1.0]# mount -t ext3 /dev/loop0 /mnt/testfs
```



```
[root@charybdis /root]# df -m
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/sda1                  243       129       101  56% /
/dev/sda8                  486       205       256  45% /home
/dev/sda7                 1512       230      1205  17% /opt
/dev/sda6                 1575      1267       228  85% /usr
/dev/sda9                  144        23       114  17% /var
/dev/sdb1                 1537       609       929  40% /opt/software
/dev/sdb2                 2800      1796      1005  65% /oracle
/dev/loop0                  50         0        50 100% /mnt/testfs
[root@charybdis /root]#
[root@charybdis /root]#
```

All files placed in this filesystem will be encrypted. Be aware that it is not recommended that the filesystem be too large because of the potential performance hit that will occur. Remember that anything written to and read from must be encrypted and decrypted respectively. CryptoAPI provides the capability to create encrypted filesystems that are transparent to system users.

Summary

The main benefit of using encrypted filesystems comes from the ability to secure sensitive data beyond what is normally capable in a system. This additional security is dependent not only on the strength of the encryption algorithm but also on its proper use by users and administrators. An encrypted filesystem is of no use if the user or the administrator do not properly handle the keys used to encrypt the data on the filesystem.

Relevant Links

Cryptographic Filesystems, Part One: Design and Implementation
Ido Dubrawsky