On the fly encryption in Linux using CryptoLoop

On the fly encryption, or OTFE, is a method to have your data safely tucked away in a filesystem hidden inside an encrypted container (either file, or disk partition, or another block device) and be able to mount this encrypted filesystem in such a way that access to the data is transparent - the computer user does not even have to know the data in the container is de/encrypted on the fly when it is accessed. You will need a kernel device driver for this kind of transparent manipulation of filesystem data. In another article I describe *TrueCrypt*, a cross-platform tool that lets you work with the same encrypted files on Windows and Linux computers. That requires the TrueCrypt package (of course), but if you don't care about Windows, you do not need additional software at all.

Slackware has all the tools on board that enable you to create, configure and use encrypted containers (files) or block devices (partitions) with ease.

The important building blocks are the 2.6 kernel’s device mapper infrastructure and the device-mapper userland tools that interface with the kernel. The 2.4 kernels can not be used with cryptoloop, which is not a problem since Slackware has a ready-to-use 2.6 kernel in `/extra`. The Slackware 2.6 kernels contain the device mapper (dm) as a module, and the `device-mapper` package containing the corresponding userland tools can be found in `/extra/lvm2`.

Common scenario

Imagine you have a USB pendrive which contains valuable information - passwords, sensitive documents, GPG key, your CV… and imagine you lose that pendrive. Not only will it be troublesome to find copies of what you just lost (unless you're good at making timely backups) but your data could fall into the wrong hands.
There are real-life examples of people working for the Government, NGO's and security companies compromising their work by allowing other people to access and publish sensitive data, merely as the result of carelessness.

Preparing your computer

You will need to have cryptoloop and verious cyphers built as modules for (or directly built into) your kernel. The Slackware 2.6 kernel in `/extra` fulfills these requirements. Also you need to have installed the `device-mapper` package from the `/extra/lvm2` directory.

You can check which cyphers your running kernel supports by examining the output from the command

```
cat /proc/crypto
```

You will probably notice that Slackware by default supports MD5 encryption only:

```
$ cat /proc/crypto
```

```
name         : md5
driver       : md5-generic
module       : kernel
priority     : 0
type         : digest
blocksize    : 64
digestsize   : 16
```

For our pusposes we need at least AES (Rijndael) and Twofish encryption as additional options. These are loaded as kernel modules:

```
modprobe aes
modprobe twofish
modprobe cryptoloop
```

I added support for cryptoloop as well by loading the module - we will need that if we want to be able to loop-mount an image file that is using AES or Twofish encryption.

```
$ cat /proc/crypto
name         : md5
driver       : md5-generic
module       : kernel
priority     : 0
type         : digest
blocksize    : 64
digestsize   : 16

name         : aes
driver       : aes-generic
module       : aes
priority     : 100
type         : cipher
blocksize    : 16
min keysize  : 16
max keysize  : 32

name         : twofish
driver       : twofish-generic
module       : twofish
priority     : 0
type         : cipher
blocksize    : 16
min keysize  : 16
max keysize  : 32
```

To make this stick, we will need to add these lines to the `rc.modulkes` file as well. Slackware 11.0 has several of those, and rc.modules-$(uname -r) will take precedence. For the stock Slackware 11.0 “*2.6.17.13*” kernel you should therefore edit the file

```
/etc/rc.d/rc.modules-2.6.17.13
```

and add these lines to the bottom:

```
/sbin/modprobe aes
/sbin/modprobe twofish
/sbin/modprobe cryptoloop
```

Now, we're all set and good to go!

Setting up a container file

First example: we are going to create a container file of 10 MB in size which we are going to fit with an encrypted filesystem. This could be a file on a USB stick, or on the hard drive, and this section will detail the steps required to prepare the file for daily use. The Usage section will show you how to use this encrypted file .

- Create the 10 MB image file:

  ```
  mkdir $HOME/crypto
  dd if=/dev/urandom of=$HOME/crypto/container.aes bs=1k count=10240
  ```

  This command will generate 10 MB of random data and fill the file `$HOME/crypto/container.aes` with that. I don't use `/dev/zero` here because I do not want a file that contains only zeroes.
  The crux in using random bytes lies in the following:
  suppose a malicious 3rd party gets hold of your encrypted container file. Using computer forensics tools, they will try to determine where you stored “real” data in the container. When the file originally contains only zeroes, it will be quite easy to pinpoint the blocks where your data was written. On the other hand, if the original file already contained random data, then your “real” data that was written to the file will be indistinguishable from that random data (well, almost at least).

- Loop-mount the container file with transparent AES-encryption and equip it with a ext3 filesystem:

  ```
  LOOPDEV=$( losetup -f )
  echo "Our loop device is '$LOOPDEV'"
  losetup -e aes $LOOPDEV $HOME/crypto/container.aes
  mkfs.ext3 -m 0 $LOOPDEV
  tune2fs -i 0 $LOOPDEV
  ```

  The first three commands look for the first unused loop device (normally that is `/dev/loop0` but I can not know in advance if you already have some other active loopmounted files, so `losetup -f` will tell us the device to use).
  After the completion of the `losetup` command, we have a loop device that we can treat as a real block device - which means we can use it as the argument to the `mkfs.ext3` and `tune2fs` commands which create the filesystem and tune it so that it will never need a forced `fsck` when it is mounted.

  The “`losetup -e aes`” command will prompt for a passphrase - this is the password that will be used to encrypt/decrypt the data that will be written to and read from the container file in future operations. **Never** forget this password!

- With a filesystem in place, try to mount it! Then, unmount it again because we are finished with the preparations:

  ```
  mkdir -p /mnt/crypto
  mount $LOOPDEV /mnt/crypto/
  ls -la /mnt/crypto
  umount /mnt/crypto/
  ```

Usage

To use our encrypted container file, all we need to do is mount it like this:

```
mount -o encryption=aes $HOME/crypto/container.aes /mnt/crypto
ls -la /mnt/crypto
```

this triggers a "*password:* " prompt where we enter the password that we typed when creating the AES-encrypted loopmount.

- Automated mount in fstab:
  If you want to make it as simple as possible, add the following line to your `/etc/fstab` file:

  ```
  /home/<yourname>/container.aes /mnt/crypto auto
  rw,user,noauto,encryption=aes 0 0
  ```

  and change <*yourname*> to your real account name of course. With that line added, mounting the filesystem inside the encrypted container is as easy as typing:

  ```
  mount /mnt/crypto
  ```

  and this can be done by you without being root. The container's protecting passprase has to be entered to get access to the encrypted data.

The beauty of the device-mapper is that (once mounted) the embedded filesystem in that container file is transparently accessible to you. All the encryption/decryption is done behind the scenes and as long as it is mounted, you will see "normal" files and directories below `/mnt/crypto`. Once you `umount` the filesystem, the container file will appear to be filled with random data again. This is the way you should deal with sensive data stored on a USB drive!

Setting up an encrypted partition

🔧 Fix Me! this section needs my attention… 🔧 Fix Me!

From:
https://wiki.alienbase.nl/ - **Alien's Wiki**

Permanent link:
https://wiki.alienbase.nl/doku.php?id=linux:cryptoloop

Last update: **2006/11/06 20:21**