Linux Encryption HOWTO

by Marc Mutz, *Marc@Mutz.com* v0.2.1, 28 September 2000

How to set up a Linux 2.2 system to use encryption in both disk and network accesses. This document describes how you can use the International Kernel Patch and other packages to make harddisk contents and network traffic inaccessible to others by encrypting them.

Contents

1 Introduction

1.1 Legal Stuff

Encryption HOWTO for Linux Systems

1.2 I've got nothing to hide

In this section I will briefly remind you of basic rules that must be taken care of if you want to use strong encryption, as all of the presented packages in this document support (or even require) it.

As the fear of governments of cryptography being employed by criminal circles has produced some strange regulations concerning the use of strong encryption, it cannot be over-emphasized how important it is to know the law in your own country:

1. Am I allowed to use strong cryptography?
2. Am I allowed to use a particular cipher algorithm either for private or commercial use? Do I have to pay licence fees?
3. Am I allowed to participate in the development of software that contains strong cryptography?

Maybe I will extend this discussion in a later version of this paper. As for now, check the following resources to find answers to the above questions:

- The Crypto Law Survey web page by Bert-Jaap Koops at *http://cwis.kub.nl/~frw/people/koops/ lawsurvy.htm*.
- The RSA crypto FAQ at *http://www.rsasecurity.com/rsalabs/faq/*
- The `alt.security.*`, `alt.privacy.*` newsgroups.
- The Department of Justice or lawyers if in doubt (likely only interesting for commercial applications).

1.3 Aims of this document

This document will (eventually, more or less extensively) describe all major development activities around the Linux(tm) operating system that provide encryption features to the kernel.

These effords are currently being collected by Alexander Kjeldaas (*astor@fast.no*) in the so-called International Kernel Patch (see below). If some packages described here are currently not included in this patch, I will state that clearly at the beginning of the section that discusses it.

This document will not speak about other security-related issues. See the excellent *Security HOWTO* for that.

You can always find the latest version and snapshots (pre-releases) of forthcoming versions of this document at it's homepage at *http://marc.mutz.com/Encryption-HOWTO/*

1.4 Mailing List

If you have questions or suggestions regarding this document or Linux and cryptography in general, you can send them to the mailing list *linux-crypto@nl.linux.org*. To subscribe, just send a message with the single line

```
subscribe linux-crypto
```

in the *body* of a mail to *majordomo@nl.linux.org* and follow the instructions in the reply.

1.5 Notations and Conventions used

I would like to give a few notations I will use throughout this document here.

When listing commands or command sequences, I will prepend each one with a prompt, either `user$` for an ordinary user's shell, `root#` for a root shell and `user>` and `root>` for continued lines in a user and root shell resp.

Also I would like to mention that links to HOWTO's are meant to work on the *LDP site* and in any directory structure that keeps all HOWTO's in the same directory (e.g. `/usr/doc/howto`). They definitely do not work from within the *Encryption-HOWTO Homepage*.

2 Obtaining and Installing the International Kernel Patch

The International Kernel Patch is the primary souce for encryption support for the Linux kernel. Due to export restrictions in some countries that prohibit the free flow and exchange of software that contains strong cryptography, it is not possible to include crypto support in the main kernel source tree (i.e. Linus' one).

Therefore, you should not consider the packages contained therein unstable or even unusable only because they are not contained in the main kernel. Most of this stuff is actually quite stable and I partly wrote this HOWTO to broaden the base of installed international kernels, so that code development can go on faster.

The International Kernel Patches are maintained by *Alexander Kjeldaas* and can be obtained from any kernel.org mirror in the directory `/pub/linux/kernel/crypto/v2.2`. The files to download are named `patch-int-a.b.c.d.gz`, where `a.b.c` is the kernel version and `d` is the patch level of the International Kernel Patch for that particular kernel version.

I strongly recommend to get at least the 2.2.10.4 version of the patch, as it fixes a problem with moving around files that are used as loopback devices, see 6.1 (FAQ section). As of this writing (Sep 2000) 2.2.17.6 is the latest version. It should apply without rejects to all recent 2.2.1x kernels.

The international kernel is distributed in the form of patches. If you are not familiar with this distribution form, you should consult the *Kernel HOWTO* for how to apply patches to the kernel source. For the purpose of this document it suffices if you issue the following commands as root:

```
root# cd /usr/src/linux
root# cp .config ..           # this is a trick I use
root# make mrproper           # to avoid the changing of
root# mv ../.config .         # the .config file by mrproper
root# zcat ~/patch-int-a.b.c.d.gz | patch -p1
```

assuming that you downloaded the patch to root's home directory. The patch should apply without rejects.

3 The Crypto API

The Crypto API is an approach to unify the interface between kernel modules *using* crypto routines (such as the loop_gen driver or CIPE (eventually?)) and kernel modules *providing* crypto routines (such as cipher or hash modules).

It shows up as the "Crypto Options" menu in `make menuconfig`. At the time of this writing (Sep 2000), only the crypto loop device driver (`CONFIG_BLK_DEV_LOOP_GEN`) uses this facility and not all ciphers contained in the international kernel have been integrated into the Crypto Library. It is obviously desirable that all kernel modules using strong encryption use the Crypto API (where applicable).

To this end, this HOWTO will eventually include a discussion on the "API of the Crypto API", so developers trying to use this library are no longer forced to read the source to find the appropriate functions.

Developers who want to earn their merits will find here a great opportunity to play. Help to implement new ciphers and hashes and put them under the GPL. Help to convert existing ciphers already contained in the international kernel to use the crypto API. Help to enhance the performance of ciphers by implementing them in native assembler code for different platforms! You don't need to be a kernel hacker to support this project, as the modules involved are rather small and similar to each other.

Late International Kernel Patches (from 2.2.12.2 on) extend the concept of the Crypto API from plain cipher modules to digest (i.e. cryptographic hash) modules. Currently MD5 and SHA1 are supported, where MD5 was contributed by Alan Smithee and SHA1 was ported from "public domain code by Steve Reid" by Andrew McDonald. RIPE-MD is in the works from me.

4 Encrypting Disks

In this section I will show how to configure the kernel and some of its utilities to use the loop device mechanism to encrypt block devices (i.e. partitions or regular files containing whole filesystems). This is the preferred method, as it is fast, simple and reliable.

There are, however, some restrictions still not resolved that avoid its use in all and every situation (e.g. swap partitions and root filesystems cannot yet be encrypted reliably), so I will briefly discuss other approaches that may fix some of these issues in section 4.5 (Other Disk Encryption Approaches).

4.1 Configuring the Kernel

After successfully applying the patch to the kernel source tree, we are in the position to configure the kernel to actually use the encryption routines. As usual, this includes re-compilation of the kernel. If this should be the first time you compile a custom kernel yourself, get yourself the help of a friend or read the *Kernel HOWTO*.

`make oldconfig` will prompt you for only the new kernel options, while `make xconfig` or `make menuconfig` will somewhat hide the new features as they are spread over all the menus, as we will see in a minute.

After configuring the options you want (see below), you re-compile and install the kernel as usual and then reboot it. Your kernel has now all it needs to support encryption of disks.

4.1.1 Overview of kernel options

This is a short summary of disk encryption related kernel config options. See section 5 (Encrypting Network Traffic) for options related to network traffic encryption. As of 2.2.11.2, the help texts of most of these options do not contain much information.

CONFIG_CIPHERS (Crypto Ciphers):

This enables the use of a wide variety of crypto ciphers (= routines that de/encrypt data) for the Crypto API. The Crypto API can be employed to disk encryption via the generic crypto driver for the loop device (`CONFIG_BLK_DEV_LOOP_GEN`). That obsoletes the concept that the "direct" drivers `CONFIG_BLK_DEV_LOOP_CAST` and `CONFIG_BLK_DEV_LOOP_FISH2` are using. Use of the generic driver is strongly recommended.

CONFIG_CIPHER_*:

This set of options enables specific ciphers (= routines that de/encrypt data) to be used in CBC mode by the Crypto API (see above).

CONFIG_BLK_DEV_LOOP (Loopback device support):

This options has nothing to do with encryption per se. However, as the current disk encryption implementation uses the loop device, you must say "Y" here.

CONFIG_BLK_DEV_LOOP_USE_REL_BLOCK (Use relative block numbers...):

This options allows you to copy around the file containing your encrypted file system without affecting it's usability. Without this option enabled, you have to make sure it stays at the exact same place on the disk in order to use it, because the (absolute, as opposed to relative) block numbers the file occupies on disk affect the crypto cipher, see the question on copying files in the FAQ section.

CONFIG_BLK_DEV_LOOP_GEN (General encryption support):

This option provides an additional layer of abstraction for crypto ciphers. With this option enabled, you can use any of the ciphers (`CONFIG_CIPHER_*`) the Crypto API supports.

CONFIG_BLK_DEV_LOOP_CAST (CAST128 encryption):

This option provides 128-bit key CAST encryption in ECB mode for use with the loop device. This option does not depend on the general encryption support (`CONFIG_BLK_DEV_LOOP_GEN`) or the Crypto API above. Use of this option is deprecated.

CONFIG_BLK_DEV_LOOP_FISH2 (Twofish encryption):

This option provides Twofish encryption in CBC mode for use with the loop device. This options does not depend on the general encryption support (`CONFIG_BLK_DEV_LOOP_GEN`) or the Crypto API above. Use of this option is deprecated.

4.1.2 Selecting the right options

This is easy. First, you choose which cipher you want to use. Make sure that you know what using a particular cipher implies (licensing fees and/or even break of law). The help texts as well as the glossary at the end of this document will help you with that, although these sources are not to be considered authorative.

Next, if you want to use CAST-128 or Twofish, then say "yes" to `CONFIG_BLK_DEV_LOOP` and `CONFIG_BLK_DEV_LOOP_CAST` or `CONFIG_BLK_DEV_LOOP_FISH2`. Note that the implementation of these ciphers will likely change in future releases, as they will become accessible through the Crypto API. I'm not sure the future implementation will be compatible with existing encryption, so use these only if you have no choice or want to experiment. Also, they conflict with the `loop_gen` driver, so you can only have one of them in the kernel at any given time.

If you want to use one of the ciphers in the Crypto API, then you need to say "yes" to `CONFIG_CIPHERS`, `CONFIG_BLK_DEV_LOOP`, `CONFIG_BLK_DEV_LOOP_GEN` and the cipher you wish to use under "Crypto options". I recommend to also enable `CONFIG_BLK_DEV_LOOP_USE_REL_BLOCKS` (available in `patch-int-2.2.10.4` and later), as this solves problems with physically moving around the encrypted file on disk.

NOTE: Using relative block numbers will probably not help you if you plan to move the file containing the encrypted filesystem between underlying filesystems with different block sizes. I nearly lost my gigabyte of encrypted data while trying to do so. A solution is in the work. It will allow you to choose the encryption block size (currently equal to the block size of the underlying filesystem - this where the trouble stems from) on setup via `losetup`.

If you want to compile crypto ciphers as modules, see the FAQ section for how.

4.2 Patching the `util-linux` source

After re-compiling the kernel and rebooting it, you need to patch some of its helper programs, namely `losetup` and/or `mount`. The patch used resides in the `Documentation/crypto` hierarchy of the kernel source tree.

`losetup` and `mount` need to be patched if you want to use any of the following ciphers:

- Blowfish
- Twofish
- DFC
- 3DES
- Rijndael (broken - kills box when used)
- IDEA

- MARS
- RC5 (missing test vectors)
- RC6
- Serpent

Note that this collection is compiled from my own personal tests and is only valid for `patch-int-2.2.17.6`. As it is one short-term goal to unify the conceptional design of the crypto modules (i.e. all will use the loop_gen abstraction someday), the size of this list is likely to monotonically increase as new releases come out.

In order to patch the mount utilities to support encrypted filesystems over the loop device, you need to fetch a recent source tree of the util-linux package, e.g. from *ftp.kernel.org/pub/linux/utils/util-linux/* or oneof it's mirrors.

As usual, you issue the following commands to apply the patch:

```
user$ tar xfzv util-linux*.tar.gz
user$ cd util-linux*
user$ patch -p1 < /usr/src/linux/Documentation/crypto/util-linux*.patch
```

It should apply cleanly if you use the patch only on util-linux trees that have a higher version number than the patch file shows.

Next you have to make sure that only the mount utilities are built. You can get in all sorts of troubles if you mess around with `init` or the password authentication programs. **You can render your computer unusable if you do not make sure that you only compile `mount` and `losetup`!.**

In order to avoid this mess, you need to install the executable files yourself, so you can gain control over what is being installed. But that is not hard to do:

After paching the util-linux source tree as described above, issue the following commands in the `util-linux*` directory. Make sure you have not currently mounted any filesystems using the loop device.

```
user$ ./configure
user$ make -C lib setproctitle.o    # mount depends on this
user$ cd mount
user$ make losetup mount umount
root# for i in losetup mount umount; do
root> j=$(locate bin/$i)
root> cp $j{,.old} && chattr +i $j.old
root> cp $i $j
root> done
```

If the make step fails, check that `/usr/include/linux` and `/usr/include/asm` resp. are symlinks to `include/linux` and `include/asm-arch` resp. Some distributions (e.g. Debian) only have a *copy* of the kernel include there. If you change the kernel, they become outdated.

Change the permissons and ownership of the new binaries according to your distribution's policy (check the old binaries for the right settings).

4.3 Making an Encrypted Folder

Create a file that will hold the encrypted data. We will call it `.crypto` for now and assume it is placed in the home diretory of user "user". You can use any other name, though. Note that this file will not grow

dynamically as you feed more and more data into it. On the other hand, it will permanently block the amount of disk space you allocate to it, even if there is no data stored in the encrypted folder. However, you can always grow or shrink the filesystem manually, see FAQ section for how. Choose a reasonable size, e.g. 10M in the following command (calculated as *bs * count*):

```
user$ dd if=/dev/urandom of=~/.crypto bs=1024k count=10
```

This will take some time, as the urandom device emits random numbers that need to be created first. The above command takes about one minute to complete on my 300 MHz AMD K6-2. You might want to use `/dev/zero` instead, if you have a slow machine or want a huge crypto filesystem, but then your filesystem is said to be vulnerable to known-plaintext attacks. I cannot follow that argument, as we do not encrypt the zeros, but instead pretend they were the ciphertext. However, using zeros to create the file in the first place makes it possible for the attacker to know which blocks have not been written to. This *may* give him one or another hint as to how to attack your ciphertext.

You may also choose to encrypt a whole partition. Just make sure it contains no data and replace the filename `~/.config` in the following discussion with the name of the block device that you wish to use (e.g. `/dev/sda3`).

Now set up this file as the target of a loop device. You must do this as superuser, unless the loop device files can be written to by everyone (not recommended). Choose a loop device that is not currently in use. Try `mount|grep loop` to see which loop devices are currently used by mounted filesystems.

```
root# losetup -e ciphername /dev/loop0 ~user/.crypto
Password :
```

Note that you should take your time as you enter the password, as you are not prompted to enter it again for typo checking! One way around this, if you have a recent bash (tested to work with bash-2.04, tested to not work with bash-2.01) and util-linux-2.10p or higher (which will hopefully include the `-p` option to `losetup` and `mount`), is to use the following *little script*:

```
#!/bin/bash

# the cipher is the first command line argument
CIPHER=$1
# the loop device to use is the second
LOOPDEV=$2
# the underlying file is third
UNDERLYING=$3

# until the two passphrases match and are not empty...
until [ "$PASS1" = "$PASS2" -a -n "$PASS1" ]; do
        # the bash read buitlin has to support the -s option.
        # Don't use read without -s!!
        read -s -p "Enter Passphrase: " PASS1; echo
        read -s -p "Re-enter Passphrase: " PASS2; echo
done

# setup the loop device using the passphrase given on STDIN.
echo "$PASS1" | losetup -e $CIPHER -p 0 $LOOPDEV $UNDERLYING
```

What you have done so far is to set up a so-called encrypted *block device* `/dev/loop0`. It is functionally equivalent to a partition device (e.g. `/dev/hda1`), so you next need to do what you (or your installation

tool) did with naked partitions: Make a filesystem on it. Ext2 should be your choice, as it is Linux' native filesystem: (**FIXME:** what about journalling?)

```
root# mke2fs /dev/loop0
```

Next, you mount it. We take the mount point to be `~/crypto` (notice the omission of the leading dot!):

```
user$ mkdir ~/crypto
root# mount -t ext2 /dev/loop0 ~user/crypto
```

The last step needs to be done as root, as only the superuser is allowed to (un)mount filesystems, unless otherwise stated in `/etc/fstab`.

If all worked well, you now have a filesystem that encryptedly resides in the file `~/.crypto` and is mounted on `~/crypto`. You can enter it and look at the files in it (not many now as it is just newly created, only the `lost+found` folder should be visible).

Now, all this is nice but really clumsy to do everytime you want to access the encrypted data. But before you learn how to make mounting the encrypted file/folder more user-friendly, you first need to unmount and destroy the loop device. I'd like to make it very clear at this point that encryption does not buy you anything if you leave the filesystem mounted all the time and/or set the permissions not restrictive enough. You should be familiar with what the following commands do, so I will let them speak for themselves:

```
root# grep -q "lost+found" /dev/loop0 && echo 'found!'
root# chmod go= ~user/.crypto
root# chown user -R ~user/crypto
root# chmod go= -R ~user/crypto
```

Now unmount the filesytem:

```
root# umount /dev/loop0
```

and detach the loop device from the file:

```
root# losetup -d /dev/loop0
```

You next edit your `/etc/fstab` to include the following line:

```
/home/user/.crypto  /home/user/crypto  ext2  \
      defaults,noauto,loop,encryption=ciphername,user  0   0
```

where `ciphername` is the same as the one you used with the `losetup` command.

This approach has some advantages over the one previously used to initially create the filesystem:

1. You don't need to be root to (un)mount the filesystem (option `user`; this is not meant to be the example username "user" we used throughout this section!)
2. You don't need to call `losetup` yourself, as `mount` will do that for you (options `loop`, `encryption`)
3. You don't need to keep track of already used loop devices, as `mount` will do this for you, choosing free loop devices by itself as needed.

Warning: `mount` currently has a bug that leads to the assignment of additional loop devices, e.g. when remounting. The symtoms are that the sequence

```
root# mount /my/crypto/fs.file
root# mount -o remount,ro /my/crypto/fs.file
root# umount /my/crypto/fs.file
```

will leave `/dev/loop0` assigned to `/my/crypto/fs.file`. Temporary fix: issue `losetup -d /dev/loop0` after the `umount` command. Util-linux version 2.10p should fix this problem.

4.4 Automagically Encrypting the Home Directory with "EHD"

I cannot yet discuss this (additional) patch to the util-linux tree, as I have not yet successfully installed it on my system.

You may do your own experiments, though: The patch (written by Id Est) and some documentation can be obtained from *http://members.home.net/id-est/*.

Do not expect too much comfort from this patch, however. It does not work with `rsh`, which is OK, because `rsh` is inherently insecure, but also not with `ssh` (although you can instruct `ssh` to use `login` for access control) and `su`, which is not acceptable in a networked environment. Also, I have not seen how it should work for X sessions.

All these restrictions stem from the inherent need to enter the passphrase for decryption. Maybe PAM (pluggable authentication) modules are a better place to implement this functionality, but I have yet to see such an approach being taken. However, there is a quite concrete idea of how to do this floating around in my head. If you are interested, *give my a call.*

4.5 Other Disk Encryption Approaches

The loop device approach has several drawbacks. Some will be resolved sometime soon, some are deeply buried in the design of the loop device mechanism. Some can be overcome with scripts and tricks, some are not that easy.

Major drawbacks—and a good reason for other approaches to exist—include (in order of increasing chance to overcome them):

1. You are not able to encrypt devices that are used for swap. Trying this will lead to deadlocks when swapping heavily: The kernel tries to free memory by swapping out some areas of memory, the device tries to encrypt the data, encryption requires memory—bingo.
2. You are not able to export directories encryptedly via NFS (loop device over NFS-mounted files doesn't work in current kernels, as far as I know). So you must export directories in the clear.
3. Root filesystem encryption is tricky.
4. (Automated) incremental backups are difficult.
5. Changing password. The hashed passphrase is used as the encryption key, so changing the passphrase changes the key, i.e. you are no longer able to access the encrypted data.

Other disk encryption approaches known to me are:

- PPDD (Personal Privacy Disk Driver) by Allan Latham *(alatham@flexsys-group.com)*. It claims to be able to encrypt root and swap devices and be "fool-proof" in its usage. I have downloaded, but not yet installed it. Latest version as of this writing (Sep 2000) is 1.2, anything below it is not recommended by the author due to a severe bug. There is a mailing list where you can ask questions, just send a

message with the single word *subscribe* in the body of the message to *ppdd-request@linux01.gwdg.de*. You'll receive a message telling you what to do next. *This is what Doobee writes about it in his document (see below for an URL):*

> You can read more about the design of PPDD in the specification or in the PPDD man page (available at the homepage, see below). One of the very nice features of PPDD is that you can decrypt devices without kernel support, if needed. And the author has thought extensively about ways to backup encrypted media and shows several solutions to solve this problem. He even made it possible to have an encrypted root partition. One of the drawbacks of ppdd is that it doesn't use the now evolving Linux Crypto API. But you can use it together with the International Kernel Patch without big hassle. Perhaps you get a few rejects while patching the two together, but these are easy to resolve by hand.
>
> You can find PPDD at *http://linux01.gwdg.de/~alatham* and *ftp://ftp.gwdg.de/pub/linux/misc/ppdd*. If you want to know how to install PPDD, check out the file ppddhow.txt.
>
> I have done some performance testing on a dual PII/266 MHz/256 MB using `bonnie`. Writing on an encrypted volume takes about twice the time as writing to a plain volume. But reading from an encrypted volume takes 4 times as long as reading from an unencrypted one:

```
              -------Sequential Output-------- ---Sequential Input--
              -Per Char- --Block--- -Rewrite-- -Per Char- --Block---
Machine    MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU
plain     100  4022 95.0 13628 28.7  4036 17.8  3782 84.7 11663 20.4
encrypted 100  2609 75.5  5719 17.2   808 23.8   968 44.9  1165 28.1
              --Random-- -----Time------
              --Seeks--- -user- -system-
Machine    MB  /sec %CPU    sec     sec
plain     100 286.3  7.0  42.98   13.82
encrypted 100  18.2  5.1  43.40  103.28
```

> While doing these tests my PS/2 mouse reacted like I was using Windows :-(but this should be tweakable by skewing with the buffer cache.

The rest of this section is copied with kind permission of the author from Doobee R. Tzeck's (drt@ailis.de) *document "Encrypting your Disks with Linux" at* http://drt.ailis.de/crypto/linux-disk.html.

- CFS (Cryptographic Filesystem) CFS is the first free UNIX disk encryption program hacked by well-known Matt Blaze. It hooks into NFS so one feature of CFS is the fact that you don't have to fiddle with the kernel to get it running and CFS is more portable among UNIXes than the other solutions. Another nice thing is that you can use CFS over NFS so that your files won't be transmitted in clear text over the wire. You can find more about the working of CFS by reading the *Cryptographic File System under Linux HOWTO* or "A Cryptographic File System for Unix" by Matt Blaze at *ftp://research.att.com/dist/mab/*.

  CFS supports DES, which is insecure because the key is too short, 3DES, which can be considered secure but painfully slow, MacGuffin, which is broken, and SAFER-SK128, which has an unusual design and is designed by some NSA buddys at Cylink—enough reason to not fully trust this algorithm. But *darkstar@frop.org* was kind enough to hack Blowfish into CFS and Matt Blaze integrated it into CFS 1.3.4.

  The main problem of CFS even with Blowfish is the lack of speed. This results from CFS being an user space daemon forcing the data to be copied several times between kernel and user space. If you want to encrypt large amounts of data expect a significant performance penalty when using CFS.

  – *cfs source code* `<http://drt.ailis.de/crypto/cfs-1.3.3.tar.gz>`

- *cfs with blowfish source code and bf_tab.h* <http://drt.ailis.de/crypto/cfs-1.3.3bf.1.tar.gz>

- TCFS (Transparent CFS) which is being developed at the University of Salerno, Italy, claims to improve Matt Blaze's CFS by providing deeper integration between the encryption service and the filesystem, which results in a complete transparency of use to the user applications. But the developers seem to focus much more than Matt Blaze on substituting NFS. A nice feature of TCFS is that it will allow you to securely share files among the members of a group. One big misfeature of TCFS is the fact that it needs kernel patches and that the patches are still made for the now obsolete 2.0 kernels. Nevertheless TCFS is under active development. Another problem with TCFS is that it only supports minimal (read: *no*) key management. There is some placebo-key management delivered with TCFS, but that is next to nothing, using only the login password to decrypt the key.

  To learn more about TCFS, read the TCFS-FAQ at *http://tcfs.dia.unisa.it/tcfs-faq.html*. Since it is from Italy, which is part of the free world, you can download it without any problems, from the TCFS Homepage at *http://tcfs.dia.unisa.it/*.

- SFS (Steganographic File System) The theoretical background of SFS was laid by Ross Anderson, Roger Needham and Adi Shamir in their paper "The Steganographic File System". You can grab it at *http://drt.ailis.de/crypto/sfs3.ps.gz*.

  The first implementation was done by Carl van Schaik and Paul Smeddle in a Project called vs3fs. They describe the code they have hacked this way:

  > Ok, firstly we would like to note that this project evolved from a computer science project that had specific goals and deadlines and thus the code may not be complete in specific areas and still has many bugs to be stomped out. Briefly, a steganographic filesystem aims to be more than just your every day encrypted file system. It tries to hide the information in such a way as to discredit its very existence. This has been a very difficult task to perform given such a short development time, but we have succeeded in attaining this goal despite a few alternative methods of doing things (read: kludges :).
  >
  > We present this filesystem as is. We take no responsibility for any damage to disks or data while using this program. I repeat: This is still *very* experimental and you will probably lose data stored on steganographic partitions. We also hope that some of you will be able to work out some of our problems and perhaps try to modify the structure to be more flexible and to provide better security. After all... thats the beauty of open source :)
  >
  > We must also note that we use encryption methods that may be stronger than those allowed in your country. We will accept no responsibility for your actions involving your country's regulations.

  They had a homepage at *http://leg.uct.ac.za/~carl/vs3fs/* with patches for Linux 2.0 and 2.1, but this page is now defunct. Seems they have left university and also left SFS alone.

  Peter Schneider-Kamp *(peter@schneider-kamp.de)* updated the stuff to 2.2. This stuff can be found at *http://www.linux-security.org/sfs/*. But since his pages seem to be down, too, you can also get it at *http://drt.ailis.de/crypto/sfspatch-2.2.10.tar.gz*.

  SFS still has a lot of rough edges and Peter doesn't seem to maintain it actively.

- StegFS (Steganographic File System) Andrew McDonald and Markus Kuhn did another implementation of the ideas outlaid in the paper of Anderson, Needham and Shamir. They claim that SFS is flawed and this claim seems reasonable.

  StegFS seems to be really elaborate and looks much more usable to me than SFS. Since McDonald and Kuhn come from the same University as Anderson, it seems obvious that they tried hard to meet the criteria of the Steganographic Filesystem paper.

StegFS has a homepage at *http://ban.joh.cam.ac.uk/~adm36/StegFS/*.

- CryptFS There is another good-looking crypto file system called CryptFS. It is described in *"CryptFS: A Stackable Vnode Level Encryption File System"* <http://www.cs.columbia.edu/~ezk/research/cryptfs/index.html> and can be downloaded from *http://www.cs.columbia.edu/~ezk/research/software/cryptfs/*.

  But since this site is in the USA, which is a country keeping its scientists from publishing research worldwide, CryptFS is unusable for people from the free world until some dark figure smuggles it over here.

- BestCrypt and Virtual Private Disk *As they are commercial products, I (Marc Mutz) will not include a discussion of them here.*

5 Encrypting Network Traffic

This section is in preparation. See the *Release Notes* for a planned roadmap. I included the CIPE documentation as a subsection, but the section layout is not complete, so this may change (Why doesn't linuxdoc provide a *part* commamd like LaTeX? :-(

Anyway, this section includes descriptions of Linux kernel add-ons that allow you to encrypt the flow of network traffic (at least IP traffic, don't know about IPX and such).

5.1 The Concept of IP Tunnels

All network encryption packages described in this document work on the same basis: A *private* network, which should be inaccessible to the outside world, is spread over many (physical) locations (read: company centres and/or roaming users) that are connected via the "evil" internet. This setup is commonly called VPN (virtual private network), because the private network shares its physical structure with the "real" internet.

The task is to route IP packets originating in the private network and destined for other hosts in the private network through the internet while preserving privacy.

This section describes problems and concepts common to all of the VPN packages to be described later on.

5.1.1 What are IP Tunnels?

The basic networking protocol nowadays is TCP/IP, mostly because it is the protocol the internet uses. But there were other network protocols and ever will be. When the internet began to grow, everyone was tempted to switch to the TCP/IP protocol, regardless of what he was using before. But many applications were not TCP/IP aware and it proved difficult to convert certain subnets to TCP/IP. However, one still wanted to benefit from the TCP/IP protocol and use it for communicating with the internet.

This was one major reason to invent *tunnels*. Just like TCP/IP packets are packed in Ethernet frames to send them over Ethernet or in PPP frames to send them over PPP links, one now wrappes TCP/IP packets with e.g. IPX frames to *tunnel* TCP/IP through IPX, or vice versa.

Thus, one was able to use TCP/IP in IPX or ATM—you name it—environments without fundamental changes to routers and so on.

The next picture shows an IP packet contained in an IPX frame:

```
| IPX header || IP header |      payload     |
+------------++-----------+------------------+
           <----- IP packet -------------->
 <----- IPX packet containing IP packet ------>
```

Of cource, no-one prevents you from tunneling IP packets in IP packets. While this does not seem very useful at first sight, there are indeed situations where this concept makes your network admin's life much easier.

One typical example of ordinary IP-in-IP tunneling is the connection of a roaming user to a LAN. The roaming user can plug his laptop into whatever network is available (provided it has a connection to the internet), establish a tunnel to a gateway in the home LAN and route everything through the tunnel. This means that the roaming user will always find the same configuration of networking services available once he gets the tunnel to the home LAN up and running. Another typical example is LAN-LAN coupling.

You can find out more about ordinary tunneling (as opposed to encrypted tunnels) in the *NET-3-HOWTO*. We will instead concentrate on tunnels that encrypt everything that is put into them.

5.1.2 Private vs. Carrier Networks

The main thing to keep in mind when discussing VPN setups is the existence of *two* logical networks sharing the same physical network hardware. It is important not to confuse those two.

On the one hand, one has the *private network*, consisting of possibly many sites and hosts, which have no (inherently) secure connection between each other. One has to provide for a secured, transparent connection between them in order to make the pieces a whole.

On the other hand, one has the *carrier network* (often the internet) that is used to connect the different sites the private network is physically located at.

Typically, the private network uses IP subnets that are reserved for use in closed networks with no connection to the internet. Those subnets include the class A 10.0.0.0/8, the class B 172.[16-31].0.0/16 and the class C 192.168.[0-255].0/8 subnets. Those IP addresses will not be routed by internet routers and thus can only be used for their intended use: private networks. (There are, however, examples where the carrier network uses these addresses and the "private" network uses ordinary, routable addresses, see Section 5.1.4 (Encrypt the Local Ethernet).)

All VPN packages establish an encrypted point-to-point connection on top of an existing network between two hosts. This connection belongs to the *private* network, while the existing network serves as the *carrier* of the encrypted datagrams.

5.1.3 Routing Issues

The toughest part in setting up a VPN seems to be the correct routing. But this is only so if one confuses the two distinct networks described above. Indeed, one has to maintain two disjoint routing tables:

One for routing the private traffic within the private network (usually through the tunnel interfaces) and one for routing the encrypted datagrams over the carrier network (usually through the physical interfaces like `eth0`, `ppp0`).

Be careful, though: Not all physical network interfaces belong to the carrier network! The internal Ethernet of a company may well belong to the private network.

The problem is now that Linux knows nothing of our ambition to create a VPN setup. Also, it does not know about private and carrier networks and their distinction. All Linux can do is, based on its routing table, send an IP packet destined to a particular IP address through an appropriate network interface.

Therefore, the logical distinction of the two networks has to be expressed in terms of routing rules. The best practice proves ever so often to be to consider the carrier network alone first. Once you have the routing table for the carrier network, you start creating the tunnels and add routing rules for the private network, independent of the carrier networks' rules.

5.1.4 Example: Encrypt the Local Ethernet

The simplest class of a VPN is where the private and the carrier network coincide. As an example, I picked the case where a company wants to encrypt all traffic in the local Ethernet. This can be useful, if the LAN carries sensible data and one wants to make the EM radiation of the ethernet cables unusable to possible spies.

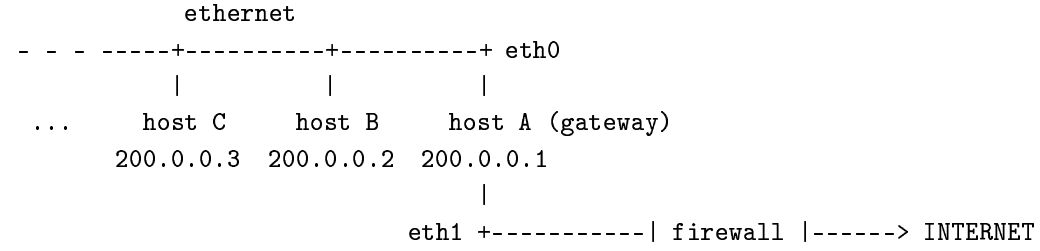
In this setup one hits the limit of most current VPN packages: They only know of point-to-point connections. Only FreeS/WAN has/will eventually have the ability to connect two arbitrary hosts transparently.

So, the only way out is to either select a host as internal gateway, and route all packets through it or to set up point-to-point connections for every single combination of hosts on the network.

While the first method minimizes configuration overhead, it also minimizes performance, because each packet has to be sent from the originating host to the gateway and back to the destination host. So network performance is roughly halved, not even counting the additional de/encryption process on the gateway host.

The second method maximizes both network performance and configuration overhead, because each change in the network topology needs to be reflected in the local (w.r.t. each host) configuration files.

Assume that the company's LAN is a single Ethernet segment and an IP subnet with net address 200.0.0.0 and subnet mask 255.255.255.0, i.e. the class C network 200.0.0.0/24. It has a gateway to the internet (to exchange mail, etc) with IP address 200.0.0.1. Security is enforced by a firewall in front of the gateway, so our task is only to encrypt all traffic in the local part of the ethernet.



As the hosts behind the firewall have transparent access to the internet (within the rules of the firewall), it is necessary that the private network has IP addresses that are routable. The carrier network does not need to have routable IP addresses, so we choose to assign IP address 200.0.0.0/24 to the private network and IP address 10.0.0.0/24 to the carrier network.

In the configuration-friendly model where all traffic is routed via the gateway (200.0.0.1 in our setup), host A's routing table would look like this:

Routes for the carrier network and the connection to the firewall:

```
 Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
 10.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
 firewall        0.0.0.0         255.255.255.255 UH    0      0        0 eth1
 127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
[200.0.0.0       0.0.0.0         255.255.255.0   U!    0      0        0 eth0]
 0.0.0.0         firewall        0.0.0.0         UG    0      0        0 eth1
```

The routing table entry in parenthesis is a *reject route* preventing that local traffic is sent unencryptedly via the default route if the point-to-point route for a particular host is not set or there is no host corresponding to a given IP address.

Routes for the private network:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
200.0.0.2       0.0.0.0         255.255.255.255 UH    0      0        0 tunl0
200.0.0.3       0.0.0.0         255.255.255.255 UH    0      0        0 tunl1
200.0.0.4       0.0.0.0         255.255.255.255 UH    0      0        0 tunl2
  ...             ...                 ...       ...  ...    ...      ...  ...
```

Here `tunl`x stands as a placeholder for the interface name of the particular software used (e.g. `cipcb`x for CIPE or `ipsec`x for FreeS/WAN).

The routing table on the other hosts (B, C, etc) would look simply thus:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
10.0.0.1        0.0.0.0         255.255.255.255 UH    0      0        0 eth0
200.0.0.1       0.0.0.0         255.255.255.255 UH    0      0        0 tunl0
0.0.0.0         200.0.0.1       0.0.0.0         UG    0      0        0 tunl0
```

Here, 10.0.0.1 is the IP address of host A in the carrier network (this is all any given host needs to know for its routing job, but the first route can safely be changed to be a *net* route).

In the performance-friendly model, host A's routing table would look the same, but the other hosts would have a routing table like the following:

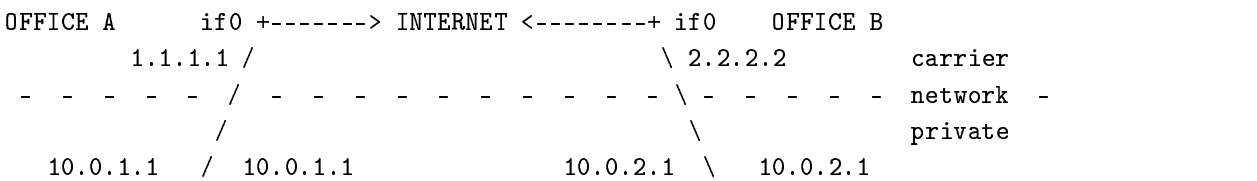
```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
200.0.0.1       0.0.0.0         255.255.255.255 UH    0      0        0 tunl0
200.0.0.2       0.0.0.0         255.255.255.255 UH    0      0        0 tunl1
200.0.0.3       0.0.0.0         255.255.255.255 UH    0      0        0 tunl2
  ...             ...                 ...       ...  ...    ...      ...  ...
0.0.0.0         200.0.0.1       0.0.0.0         UG    0      0        0 tunl0
```

Note that the point-to-point routes are usually set up automatically by the controlling daemon, so in the last routing table, you only need to set the default and net routes by yourself.

5.1.5 Example: Connect Two Private Ethernets

The classical class of VPNs is the situation where two or more physically separated offices are to be connected in a transparent and secure way via the internet.

We first look at the routing required to connect two offices:



We have two LANs A and B with private IP address spaces 10.0.1.0/24 and 10.0.2.0/24, respectively and we assume that one host in LAN A (called gw.a) is connected to the internet via the network interface `if0` (this is a generic name, think of it as `ppp0` for a PPP link or `eth1` for a DSL or leased line, etc.) with IP 1.1.1.1. The same is true for a host gw.b in LAN B, connected to the internet via `if0` with IP address 2.2.2.2. The `eth0` IP address of the gateways is, as usual, 10.0.1.1 and 10.0.2.1, respectively.

We want to connect these two LANs with a tunnel, so that traffic can flow between them, although they use IP addresses that are not allowed to travel the internet.

The advantage of using private IP addresses is that these need not be reserved (and possibly paid for) and that they make it hard for external intruders to attack hosts, simply due to the fact that IP packets destined for these IP addresses are immediately dropped by every internet router.

We identify the internet, together with the `if0` interfaces of each gateway, to be the *carrier network*, while the two LANs and the tunnel between the two gateways constitute the *private network*.

If we first consider the routing needed for the carrier network, we simply get the following tables:

```
gw.a's Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
2.2.2.2         0.0.0.0         255.255.255.255 UH    0      0        0 if0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo

gw.b's Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
1.1.1.1         0.0.0.0         255.255.255.255 UH    0      0        0 if0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
```

On top of this, we construct the routing tables for the private network:

```
gw.a's Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.2.1        0.0.0.0         255.255.255.255 UH    0      0        0 tunl0
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
10.0.2.0        10.0.2.1        255.255.255.0   UG    0      0        0 tunl0

gw.b's Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.1.1        0.0.0.0         255.255.255.255 UH    0      0        0 tunl0
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
10.0.1.0        10.0.1.1        255.255.255.0   UG    0      0        0 tunl0
```

5.1.6 Example: Connect a Mobile Host to the Internal LAN

This class of VPNs is a special case of Example 5.1.5 (Connect Two Private Ethernets), and the routing involved is in fact easier when considering the tables only.

What makes this example special and interesting from a configuration point of view, is the fact that the tunnels are dynamically created and destroyed possibly quite often (as opposed to created at boot time and destroyed at system shutdown) and that the IP address of the mobile host may change frequently, depending on what LAN or ISP the laptop is currently plugged into.

Often one part of the carrier network consists of a dynamically established PPP link, which adds even more configuration overhead and variables to consider.

This setup depends heavily on the VPN software used, so further discussion of this is postponed to the sections on specific VPN software below.

5.2 CIPE - Cryptographic IP Encapsulation

CIPE by Olaf Titz (*Olaf.Titz@inka.de*) ships with the International Kernel Patch (see Section 2 (Obtaining and Installing the International Kernel Patch)). The userspace tools needed are in the `net/cipe` subdirectoy of the patched kernel source. The package is also available from *http: //sites.inka.de/~bigred/devel/cipe.html*. I only document version 1.3.0 (Apr 1999) here, although there is a newer version out (1.4.2). It will not inter-operate with earlier versions due to a bug in the key handling or so in those earlier versions, but it finally supports Windows! See *http://CIPE-Win32.sourceforge.net/*. There is also a mailing list dedicated to CIPE. You can subscribe to it by sending a message with the single line *subscribe cipe-l* in its body to *majordomo@inka.de*.

5.2.1 Concepts and Features

CIPE is a very simple package in that it requires you to make some decisions at compile-time, which leads to specialized and simpler code, which in turn is supposedly less bug-loaden than a complicated protocol like IPSec. CIPE is only compatible to itself, but often enough that is no limitation.

These disadvantages turn out to be big wins when it comes to maintaining the package, as configuration is much easier done than e.g. the disk encryption in Section 4 (Disk Encryption).

CIPE intercepts the network stack just above the network layer by adding a new network interface (e.g. `cipcb0`). (IP-)Packets routed through it will be encrypted and put through a tunnel to a peer gateway, where they will be decrypted and delivered.

It employs symmetric ciphers (Blowfish and IDEA) and a shared secret (key) with dynamic re-keying, i.e. the shared secret is only used for authentication and exchanging the first dynamically generated, actual encryption key. Data is encrypted using these dynamic keys.

The package consists of a kernel module (`cipcb` for Blowfish and `cipci` for IDEA) and a user space deamon (`ciped-cb` resp. `ciped-ci`).

5.2.2 Compiling and Installing

The only thing you need to do in your kernel is to enable loadable module support (`CONFIG_MODULES`). Of course you need to have some kind of (carrier) network running, but I assume that this is in a working state. See the appropriate HOWTOs for how to configure that.

CIPE currently only works as modules and I do not see anything indicating a change to that. Also, the building of the CIPE modules is currently not included in the kernel build process, not even with the international kernel, although the entries in `make config` suggest otherwise. At least on my 2.2.17.6 CIPE modules were not build resp. installed by `make modules modules_install`.

So you have to do that yourself: Change into the CIPE directory. It is the `net/cipe` subfolder of the kernel source tree if you use CIPE from the international kernel patch or `cipe-1.3.0` if you extracted it from the tar archive on *inka.de*.

If you have the source of the kernel under `/usr/src/linux`, then you can just enter `./configure` to let CIPE figure out the details of your kernel. Otherwise you have to tell the configuration script where to find the kernel source with the `-with-linux` option, e.g.

```
root# ./configure --with-linux=/usr/src/linux-2.2.17
```

if your kernel source resides in `/usr/src/linux-2.2.17`. If you want to use the IDEA cipher instead of the default Blowfish, then call `./configure` with the option `-enable-idea`. These are the compile-time options mentioned in the 5.2.1 (Concepts and Features) Section. You can compile different versions of the CIPE modules and userspace tools one after the other to allow the use of both ciphers.

After the configure script has successfully finished, you will find a new folder named something like `2.2.17-i386-cb` in the CIPE directory. This helps you with the compilation of multiple versions of the CIPE package (e.g. for using both ciphers). The first part of the folder name consists of the kernel version (like `uname -r`) for which the modules will be compiled, followed by the machine type (like `uname -m`) and by the CIPE features: The first character stands for the protocol version (only c=3 allowed) and the cipher used (b=Blowfish and i=IDEA).

To compile and install the modules and userspace tools, enter the newly created directory and type

```
root# make && make install
```

That was it! Have a look at the file `cipe.info` which accompanies the CIPE sources if you want to compile CIPE for special cases (e.g. if you do not have the kernel source, but only the includes installed).

5.2.3 Testing the Installation

First, you have to do a minimal configuration: Copy the `samples` directory to `/etc/cipe` and edit the file `/etc/cipe/options` to include only the parameter `key`. For a first test you can safely go with the value provided by the `options` file of the `samples` directory. Later, you have to change this key, of course.

It is important that the key is the same on both machines you want to connect with a CIPE tunnel. For first tests, I would recommend that you set up a CIPE connection between hosts on the local Ethernet segment, if possible. Things like CIPE over PPP are covered in more depth later on.

To be specific, assume that we are living on an ethernet that consists of two boxes and uses IP addresses from the private range 10.0.0.0/24. We want to use Blowfish encryption. The two boxes, called A and B, have IP addresses 10.0.0.1 and 10.0.0.2 respectively.

The first step after compiling and installing CIPE on both machines is to load the modules on both hosts:

```
root@A# modprobe cipcb
root@B# modprobe cipcb
```

Next, you start the CIPE-daemon on each machine:

```
root@A# ciped-cb me=10.0.0.1:6789 peer=10.0.0.2:6543 ipaddr=10.0.1.1 ptpaddr=10.0.1.2
root@B# ciped-cb peer=10.0.0.1:6789 me=10.0.0.2:6543 ptpaddr=10.0.1.1 ipaddr=10.0.1.2
```

As you can see, the configuration parameters have to be swapped, so to speak, as me's peer is peer's me. Also, we need to use additional IP addresses for the CIPE interface (called `cibcb` here), see next section.

If everything worked well, you can try to ping the peer to see if the tunnel has been successfully established:

```
user@A$ ping 10.0.1.2
user@B$ ping 10.0.1.1
```

A common mistake is to have a firewalling policy of DENY or REJECT and no ACCEPT rule corresponding to the UDP port numbers used. `ping` will report "sendto: write: operation not permitted" if this is the case. See the ipchains-HOWTO for how to add such rules. If you are using masquerading together with CIPE, you may want to look at the CIPE+Masquerading-miniHOWTO, too.

If you were able to ping the other host, you can now try to telnet or ssh the other host. If all works well, you have successfully completed the first tests.

5.2.4 Configuration Overview

In this subsection you will learn of the various parameters that influence CIPE's behaviour. You have seen the most important ones in the last section, here we discuss their meaning in detail:

key, nokey

For security reasons, the `key` parameter must be set via an options file, owned by root and unreadable for anyone else. It is followed by the 128-bit shared secret, i.e. the key that is used for authentication and generation of session keys. To generate a truly random key and add it directly to your options file, so no-one can sneak it, issue the following command:

```
root# echo key $(head -c 16 /dev/random | md5sum) >> /etc/cipe/options
```

It is your responsibility that you transfer the key from one machine to the other in a secure fashion. *Do not use ftp or other unencrypted protocols to send the key!* The recommended way is to learn the key by heart and type it into the other box by hand or to send it via PGP/GnuPG-encrypted E-Mail. But also transferring it via removable media or through a ssh session is acceptable. Make sure you are *absolutely certain* that the other end of the ssh connection is indeed the box you think it is. `nokey` leads to simple (and unauthenticated) IP-in-IP tunnelling, useful only for testing. `key` and `nokey` are mutually exclusive options.

me, peer

The IP address of my host, followed by the colon-separated port number the CIPE daemon should listen to (`me`) and the IP address of the other end of the CIPE tunnel, followed by the colon-separated port number to which we wish to connect to (`peer`). These are the IP addresses that hosts of the *carrier network* "see". Make sure that all firewalls between the two hosts let pass UDP packets destined for those ports. `peer` can be 0.0.0.0:0, in which case the IP address is taken as the one of the host connecting to us, provided it successfully authenticates itself. `me` needs not be specified. In this case CIPE will pick appropriate values and report them to the `ip-up` script.

ipaddr, ptpaddr

The IP address of the `cipcb` (resp. `cipci`) interface on my host (`ipaddr`) and the IP address of the `cipcb` (resp. `cipci`) interface of the peer (`ptpaddr`). The CIPE daemon uses this information to automatically set a point-to-point route between the two interfaces after a connection has been established (just like the PPP daemon does after successful dial-in). These are the IP addresses that hosts in the *private network* "see". `ptpaddr` can be 0.0.0.0, in which case the peer's IP address is determined when the other end tries to connect and successfully authenticates itself.

mtu, cttl, metric

These optional parameters specify the `cipcb/i` device MTU (maximum transfer unit), the UDP packets' TTL (time to live) value and the `cipcb/i` device metric. See the appropriate (networking) HOWTOs if you don't know what I am talking about.

tokxc, tokey, tokxts, ping, toping, maxerr

These optional parameters specify the key exchange timeout (10), the dynamic key lifetime (600), the key exchange timestamp window (0), the time between two successive keep-alive pings (0), the timeout for those pings (0) and the maximum number of network errors between two key exchanges (8). All these parameters take their values to be specified in seconds and a value of 0 (zero) means "disable this feature/no timeout". Default values are listed in parenthesis. `maxerr` is different, though: The value is just a number (no seconds) and a value of 0 (zero) means "accept no errors", while "accept any number of errors" corresponds to a value of -1 (minus one). You normally want to go with the default values, but sometimes adjusting the `maxerr` parameter is needed if your connection is not so reliable (e.g. PPP).

ipup, ipdown, arg

These optional parameters specify the name of the `ip-ip` and `ip-down` scripts to run resp. (default: `/etc/cipe/ip-up` resp. `/etc/cipe/ip-down`) and an optional argument to pass to the `ip-up` script. Useful if you intend to run more than one `ciped`.

There is also the `socks` parameter, which allows you to connect to a SOCKS5 server with CIPE. If you need it, you'll want to read the `cipe.info` file that comes with CIPE.

Two other parameters, `device` and `debug`, are only useful in testing, since CIPE 1.3.0 can automatically grab free `cipcb/i` interfaces as needed, and `debug` is useful for just that: debugging.

All these options can be specified in either of three ways:

1. via the default options file `/etc/cipe/options`,
2. via an options file specified as `-o file` on the command line,
3. via the command line as `parameter=value`,

in order of processing (i.e. command line options override `-o file` options override `/etc/cipe/options` options).

5.2.5 Configuration Examples

The examples are roughly ordered by complexity.

Configuration for Example 5.1.5 (Connect Two Private Ethernets)

Configuration for Example 5.1.4 (Encrypt the Local Ethernet)

Configuration for CIPE over PPP

Configuration for Example 5.1.6 (Connect a Mobile Host)

Configuration for Connecting Two Mobile Hosts With Dynamic IP Addresses

5.3 Other Network Traffic Encryption Approaches

I will just give you the necessary links to find other documentation on the following packages:

- FreeS/WAN (IPSec implementation) is available at *www.freeswan.org*. Current version is 1.5 (Sep 2000). FreeS/WAN is a Linux based implementation of the IETF IPSec standard. If you want to know all there is to know about IPSec, see the RFC's concerned with it, esp. the famous RFC's *2401: Security Architecture for the Internet Protocol* thru *2412: The OAKLEY Key Determination Protocol*. Note, however, that these documents add up to 20,000 lines of pure text or 800K.

  This package is very complex indeed, but as it is based on an IETF standard, it inter-operates with many other (commercial) VPN products, including PGPnet. This makes it the single obvious choice for admins that need extensive cross-platform capabilities.

  There is a very active mailing list you can subscribe to, if you send a message with the single line *subscribe linux-ipsec* in the body of the message to *majordomo@clinet.fi*.

- ENSkip is an implementation of Sun's SKIP protocol. Its advantages over other approaches include a long history, which implies a relative stableness, and good cross-platform capabilities (at least within the Unices department). You can find its homepage at *http://www.tik.ee.ethz.ch/~skip/*

- VPND is another approach out there for Linux (and FreeBSD). You can fetch additional information from its homepage at *http://sunsite.auc.dk/vpnd/*. There is also a mailing list, which you can subscribe to if you send an empty message to *vpnd-subscribe@sunsite.auc.dk* and follow the directions given in the reply.

- SSH (Secure Shell) can also be employed to establish an encrypted IP-tunnel between two hosts. There is a VPN mini-HOWTO already covering the setup, but it is rather dated (Aug 1997). get OpenSSH, the free implementation of SSH, from *www.OpenSSH.org*

- VTUN, a (TUN/TAP) driver for Linux, *BSD, and Solaris is a tunneling suite with encryption capability. See *vtu.sourceforge.net*. It's encryption libraries do not support RC4/5/6 (YET) but it works well with lower level algs.

- PoPToP server for Linux, a PPTP (for Windows clients) *does* support encryption with a series of kernel patches and patches to pppd to enable PPTP encrypted tunnels for mobile Windows boxen VPNs. See *http://www.moretonbay.com/vpn/pptp.html*. However, I strongly recommend to not use PPTP, regardless of the implementation. It's a flawed protocol, Bruce Schneier broke it twice and Microsoft is still unable or unwilling to employ well-known protocols and algorithms, see*http://www.comterpane.com/pptp.html* for a detailed discussion.

6 Frequently Asked Questions

6.1 Disk Encryption

6.1.1 General Questions

1. **How can I encrypt swap?** With the loop device approach, you cannot. PPDD claims you can. I don't know about (T)CFS. See the sections on individual approaches below.

6.1.2 Loop Device Encryption

1. **Can I use a journalling filesystem onto of `/dev/loop`?** I don't know.

2. **Can I use all this as modules?** Sure! In `make menuconfig` (or whatever), under "Crypto options", say M to "Crypto ciphers (`CONFIG_CIPHERS`)" and to the ciphers you want. Under "Block Device", say M to "loopback device (`CONFIG_BLK_DEV_LOOP`)" and to "General Encryption Support (`CONFIG_BLK_DEV_LOOP_GEN`)". Don't select any other encryption modules unless you can't live without them and they are no longer supported by the Crypto API.

   Build your kernel and modules, `make modules_install`, reboot, `depmod -a`.

   In `/etc/conf.modules`, add:

```
alias loop-xfer-gen-0 loop_gen
alias loop-xfer-gen-10 loop_gen
alias cipher-4 blowfish            # Blowfish
alias cipher-6 idea                # IDEA
alias cipher-7 serp6f              # Serpent
alias cipher-8 mars6               # MARS
alias cipher-9 twofish             # Twofish
alias cipher-11 rc62               # RC6
alias cipher-15 dfc2               # DFC
```

3. **Why all those funny numbers?** In short, the kernel knows ciphers only by number. If you really want to know how it works, you can `grep request_module` in `linux/crypto/api.c` and `linux/drivers/block/loop.c`.

   If the cipher you wish to use is not in the above list, then see the file `include/linux/crypto.h` in the kernel source tree. There you'll find the defines for all cipher numbers. See the directory `crypto` in the kernel tree for the (file)name of the module that implements the cipher.

4. **How do I change the password / the cipher?** You can't. At least not easily. Well, you can, but you won't be able to access your data anymore. Seriously: The passphrase is hashed and then used as the encryption key. You cannot change the password and expect the hash value (ie. the encryption key) to stay the same.

   What you *can* do, however, is the following. Make sure, you have not currently set up or even mounted the filesystem you want to change. In the notation of section 4.3 (Making an Encrypted Folder) you then issue:

```
root# losetup -e oldcipher /dev/loop0 ~user/.crypto
Password : (old one)
root# losetup -e newcipher /dev/loop1 ~user/.crypto
Password : (new one)
root# dd if=/dev/loop0 of=/dev/loop1 bs=1k conv=notrunc
root# losetup -d /dev/loop0
root# losetup -d /dev/loop1
```

   If you changed the cipher, you will need to change the entry in `/etc/fstab` accordingly.

   If you plan to do this often (and maybe you should do that), it pays to write a script for it that prompts you twice for the new password. If you are really lucky, then I will have written this for you already :-).

5. **My encrypted filesystem is (nearly) full. How do I enlarge it?** If you have enough free space left so that you can create the new-size file without having to remove the old first, then the solution is obvious:

   - Mount the old filesystem, e.g. on `crypto`.

- Setup and mount a new filesystem as described in section 4.3 (Making an Encrypted Folder), e.g. on `newcrypto`.
- Copy the files thus:

  ```
  user$ tar cf - -C crypto . | tar xf - -C newcrypto
  ```

6. **But I do not have that much free space left! Is there still hope?** Yes: *ext2resize* <http://www.dsv.nl/~buytenh/ext2resize/>. It is, as the name suggests, a tool that allows you to grow or shrink ext2 filesystems. And that seems to be exactly what we'll need.

   Basically, what you have to do is:

   - Enlarge the file wherein the filesystem is contained by e.g. 10M:

     ```
     user$ dd if=/dev/urandom bs=1024k count=10 >> ~/.crypto
     ```

   - Set up the loop device and check the filesystem:

     ```
     root# losetup -e cipher /dev/loop0 ~user/.crypto
     Password :
     root# e2fsck /dev/loop0
     e2fsck 1.12, 9-Jul-98 for EXT2 FS 0.5b, 95/08/09
     /dev/loop0: clean, 11/1920 files, 506/10240 blocks
     ```

   - Resize the filesystem. The second parameter to `ext2resize` is the new size in blocks. In our example: *10240 + 10M/1k = 20480*

     ```
     root# ext2resize /dev/loop0 20480
     ```

   - Now detach the loop device again:

     ```
     root# losetup -d /dev/loop0
     ```

   You have now an enlarged filesystem. Note, however, that power failure or the like can easily destroy your data while resizing is running! There is a patch floating around that will let you do this with a mounted filesystem. With this solution, however, you have to unmount the filesystem before resizing.

7. **Can I encrypt my swap partition? Can I page to a swapfile that's on an encrypted filesystem?** No, not yet. First, some issues with memory allocations in the driver have to be resolved.

8. **Cipher xyz does not work. It says "unsupported cipher xyz" / "ioctl: LOOP_SET_STATUS: Invalid argument" although I have compiled it into the kernel.** Not for all ciphers does a user-space setup tool currently exist. Please see section 4.2 (Patching the util-linux source) for a list of working ciphers or try them all in turn.

9. **I copied the file containing the crypto filesystem to another directory / I defragged my partitions and now I cannot access my data any more.** Most probably you have compiled the kernel without `CONFIG_BLK_LOOP_DEV_USE_REL_BLOCK`. As the copy now occupies different blocks on the filesystem and you are not using relative block numbers, the ciphers get it wrong. The only thing that will help you now is the *ext2recover* <http://www.iki.fi/markus.stenberg/ext2recover.html> tool by Markus Stenberg (*markus.stenberg@abacus-solutions.com*). It will recover your data except the first four bytes of every block. If your lost data was only text files, that should be enough. If it was binary data it will most probably not be enough.

10. **I use an older 2.2 kernel. How do I avoid the above mess?** You can use the "double loop trick":

    ```
    root# losetup /dev/loop0 crypto
    root# losetup -e ciphername /dev/loop1 /dev/loop0
    ```

This will create a new block device `/dev/loop0`, where it is guaranteed that blocks are sequentially numbered starting with zero. That's the same as using relative block numbers on `crypto` directly. If you create your crypto loop device this way, you can be sure it will still work after defrag etc.

Note that `mount` does currently not support this trick, so you will have to set up your loop devices by hand. All in all, you should really update to a newer kernel :-)

11. **How do I make backups?** You can't (at least not easily) when you equal "backup" with "incremental backup". If you can sleep well with image backups instead, and if you used relative block numbers for your loop device (see questions above), you can copy your file containing the crypto filesystem to wherever you want. This includes DAT tapes and removable media, as well as CDR(W).

    You may want to use tricks like the ones presented above to change the cipher and/or passphrase to use another password for the backup file, dedicated to backups. That is because you have a better chance to remember one of the passwords if the other one gets lost. Also, if you change your password regularly, then how do you remember the password you used two months ago?

    However, if you do *not* use relative block numbers, then you cannot easily copy your file around for backups. You can mount the filesystem and have it backed up as plaintext or on another encrypted device. But that will undermine your encryption thoroughly. You'll have to do something along the following lines instead:

```
root# losetup -e cipher /dev/loop0 ~user/.crypto
root# losetup /dev/loop1 ~user/crypto-backup
root# losetup -e cipher /dev/loop2 /dev/loop1
root# dd if=/dev/loop0 of=/dev/loop2 bs=1k
root# for i in 2 1 0; do losetup -d /dev/loop$i; done
root# cp ~user/crypto-backup /backup-media
```

    This will make `crypto-backup` immune to copies (see questions further above for why). `crypto-backup` needs to be at least the same size as `.crypto`, of course.

7 Glossary

This is only a very minimal glossary, which contains only two acronyms used, but not detailed in the text. The descriptions are based on the corresponding entries of the glossary that comes with the FreeS/WAN 1.3 distribution.

CBC

Cipher **B**lock **C**haining mode, a method of using a block cipher in which for each block except the first, the result of the previous encryption is XORed into the new block before it is encrypted. CBC is the mode used in IPSEC and the current Crypto-API ciphers.

An initialisation vector (IV) must be provided. It is XORed into the first block before encryption. The IV need not be secret but should be different for each message and unpredictable.

For loop device encryption, the IV is taken to be the physical block numbers of the file which contains the crypted filesystem.

ECB

Electronic **C**ode**B**ook mode, the simplest way to use a block cipher. Each block is encrypted independently.

The loop device crypto drivers that do not use the Crypto-API use their respective ciphers in ECB mode.

ECB mode is less secure than *CBC mode*.

8 Credits

Thanks go to the following people for suggesting improvements of this document or even contributing part of it:

Dale Amon, Randolph Bentson, Andries Brouwer, Lennert Buytenhek, dath@sequent.com, Michael Gende, Davide Giunchi, Josef Höök, Alexander S. A. Kjeldaas, Allan Latham, TheSpectra (AKA Pablo Lorenzzoni), Gary E. Miller, Richard Polton, Rik van Riel, Mark Ryan, Todd Sabin, Kurt Seifried, Gisle Sælensminde, Pim Snel, A. Steinmetz, Lion Templin, trb@eastpac.com.au, Doobee R. Tzeck, Erik Walthinsen, Frank v Waveren