Dr Brown's Administeria

Dr Chris Brown
The Doctor provides Linux training, authoring and consultancy. He finds his PhD in particle physics to be of no help in this work at all.

Esoteric system administration goodness from the impenetrable bowels of the server room.

Predicting hard disk failure

Smart Is it possible to predict hard drive failure before it happens? The facts about the technology.

Smart (Self-Monitoring Analysis and Reporting Technology) is a technology developed by a number of major drive manufacturers to provide detailed diagnostic information about the health of your hard drives. The idea is to improve storage reliability by detecting indicators of impending drive failure before you've suffered any downtime or data loss. The idea isn't new. HAL 9000, you'll remember, predicted the failure of the ship's AE35 parabolic antenna in *2001: A Space Odyssey*, though fortunately in the case of hard drives we don't have to struggle into our space suits to go out and fix them. Open the pod bay doors, HAL.

For Linux, the package **smartmontools** from Bruce Allen provides user-space monitoring of Smart data. The program *smartctl* is the main reporting tool. The screenshot shows an example of the detailed reporting available using this tool. As you'll see, Smart maintains various "raw" performance measures, and normalises each one to a value in the range of 1 to 200. For each normalised value, the drive manufacturer sets a threshold, and if the normalised value falls below the threshold, the drive is considered likely to fail. At least that's the theory behind it all.

The second program, *smartd*, is a daemon that polls the Smart reports from the drives (every 30 minutes by default) and logs any error or changes in attribute values. In addition to logging to a file, smartd can also be configured to send email warnings if problems are detected. For more information, see **http://smartmontools.sourceforge.net**. You'll find content of practical relevance to either attack or defend instead of getting information that is either confusing or just plain wrong.

```
  4 Start_Stop_Count        0x0032   100   100   000
  5 Reallocated_Sector_Ct   0x0033   200   200   140
  7 Seek_Error_Rate         0x002e   200   200   000
  9 Power_On_Hours          0x0032   099   099   000
 10 Spin_Retry_Count        0x0032   100   100   000
 11 Calibration_Retry_Count 0x0032   100   100   000
 12 Power_Cycle_Count       0x0032   100   100   000
192 Power-Off_Retract_Count 0x0032   200   200   000
193 Load_Cycle_Count        0x0032   185   185   000
194 Temperature_Celsius     0x0022   108   095   000
196 Reallocated_Event_Count 0x0032   200   200   000
197 Current_Pending_Sector  0x0032   200   200   000
198 Offline_Uncorrectable   0x0030   100   253   000
199 UDMA_CRC_Error_Count    0x0032   200   200   000
200 Multi_Zone_Error_Rate   0x0008   100   253   000
240 Head_Flying_Hours       0x0032   100   100   000
241 Unknown_Attribute       0x0032   173   173   000
242 Unknown_Attribute       0x0032   200   200   000
```

› Raw *S.M.A.R.T.* data from *smartctl*. I was especially intrigued by the 'head_flying_hours'.

How smart is Smart?

Are Smart parameters really a good predictor of drive failure? Three guys from Google presented a paper at the fifth Usenix Conference on File and Storage Technologies in which they investigated failures over a population of 100,000 disk drives. They found that there were some Smart parameters that have a large impact on failure probability, but that a large fraction of failed drives had no predictive Smart signals. The paper is available at **http://labs.google.com/papers/disk_failures.pdf**.

Geeks vs humanity?

Ubuntu has probably done more than any other distribution to broaden the appeal of Linux and raise its profile as an easy to use, intuitive distribution. But when Canonical reaches its goal to entirely banish the command line from the end user's experience of Linux, what will we see? Some kind of better-late-than-never version of Windows?

Linux has no need to apologise for the way it is. Every time someone covers up more of the command line with some new graphical tool, they cover up more of what attracted me to Linux in the first place. I like the power of the shell, the myriad of "tiny languages", the flexibility of using programs in combination, the ease of scripting around the command line tools and the simplicity of documenting how to perform a task at the command line compared with graphical explanations.

Don't get me wrong. I value the work Canonical is doing. But in looking at its slogan "Linux for human beings", I found myself thinking "Why shouldn't Linux be for geeks?" Even geeks are human beings aren't they?

On an unrelated note...

A report entitled "Linux Kernel Development" from the Linux Foundation highlights the breathtaking speed and scale of development. Since March 2005 there's been a new kernel release every 2.7 months on average, with between 5,000 and 10,000 changes per release. The current kernel (2.6.24) is approaching nine million lines of code. The report concludes that "The Linux kernel is one of the largest and most successful open source projects that has ever come about." Read the full text at **www.linuxfoundation.org/publications/linuxkernel.development.php**

LXF114.sysadmin 62 21/11/08 5:40:27 pm

fstab demystified

A handy illustrated guide to the syntax of the fstab file.

All Linux distributions use a file called **/etc/fstab** to control the mounting of filesystems. This is an important file, and mistakes here can even prevent the system booting. The format of **fstab** used to be fairly straightforward, but the explosion of different filesystem types, different mount options and different ways of identifying partitions has made it more challenging. In this back-to-basics tutorial, we'll try to make sense of some typical entries in **fstab**.

Each line in **fstab** is broken out into six fields as follows:

» **Field 1** The device containing the filesystem. Exhibit A shows the use of a simple Linux device name to identify a disk partition. More recent Linux distibutions use disk labels or UUIDs to identify the partition. If the mount refers to an NFS filesystem, this field contains an entry such as **docserver:/usr/share**, which means 'the directory **/usr/share** on the NFS server called docserver'. Or if you're using logical volumes, this field will contain a device mapper name such as **/dev/VolGroup00/LogVol00**

» **Field 2** This is the mount point. It's usually the name of an empty directory within the root partition.

» **Field 3** This is the filesystem type. Linux potentially supports many filesystem types; for a list of what's currently available in your kernel, look at **/proc/filesystems**.

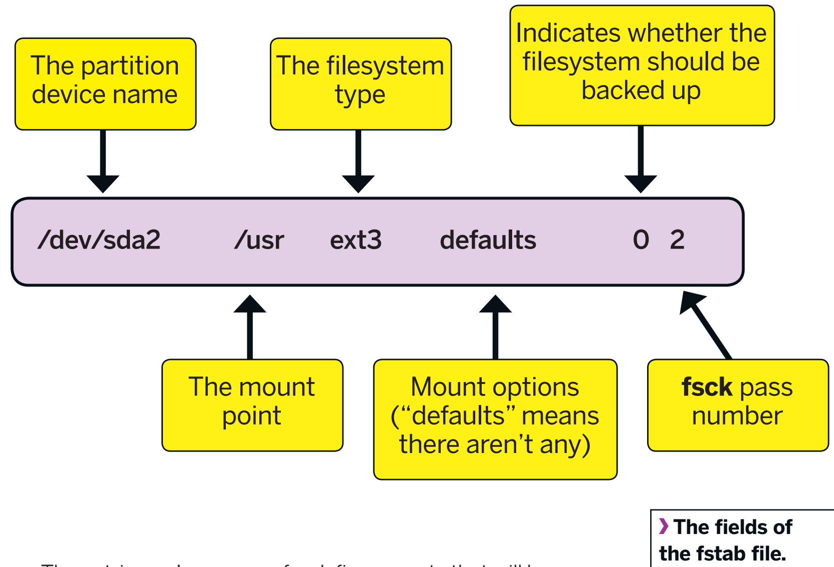
» **Field 4** This is where you specify the mount options. The set of options available depends to some extent on what the filesystem type is, and there are potentially lots and lots of them. See the table "Mount Options" for more detail. If there are no options to be specified, the word "defaults" appears here.

» **Field 5** This field is all but obsolete. It is used by *dump* (an incremental backup program) to indicate which partitions need to be backed up.

» **Field 6** This field gives some control over the order that filesystems get checked (by *fsck*) at boot time. Standard practice is to put '1' here for the root partition and '2' for all the others.

Fantasy filesystems

Not all the filesystems in **fstab** are real. Some of them are a figment of the kernel's imagination. In recent years, the concept of a "file" has been broadened to mean any source of data (not necessarily something stored on disk) that can be accessed using the standard file I/O system calls, and so can be viewed via standard commands like *ls* or *cat*. Perhaps the best-known of these pseudo filesystem types is proc, which gives a view into internal kernel data including subdirectories that provide "per process" information.



› The fields of the fstab file.

The entries we've seen so far define mounts that will be performed at boot time, in response to a mount – a command buried somewhere in a boot script. The diagram below shows an entry for a removable device such as a CD. This line is not there to define a boot-time mount, but to make mounting of removable media easier. For one thing, it means you can mount a CD just by typing a command such as

```
$ mount /dev/hdc
```

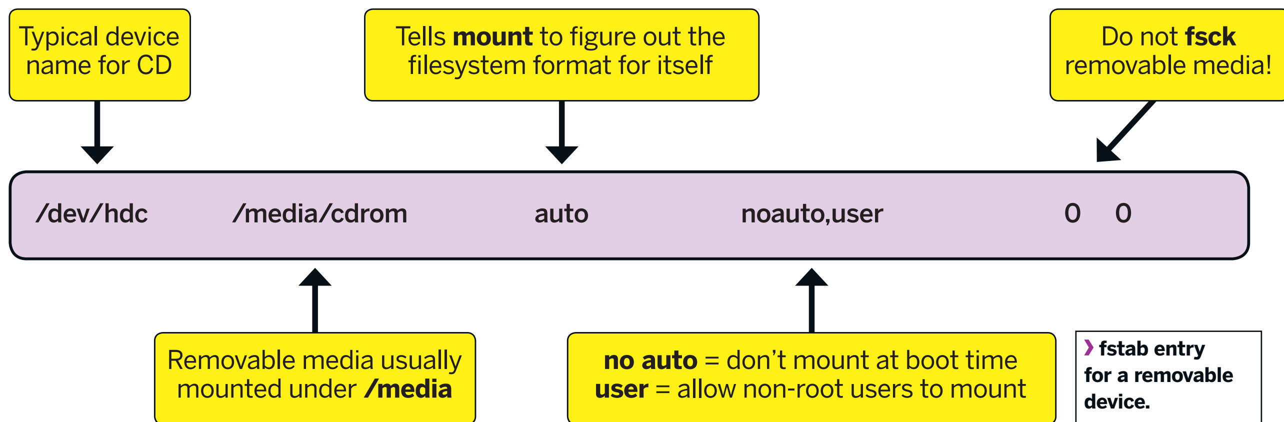
Let's wrap up by taking a look at a few more mount options. There are many, many more options than we've shown here. Take a look at the man pages for *fstab* and *mount*, which are very detailed indeed.

» **ro** Mount read-only. For example, if **/usr** is on a separate partition, mounting it read-only will improve security by preventing modification to binaries in **/usr/bin** and libraries in **/usr/lib**.

» **noatime** Prevents the updating of the "last access time" timestamp of files under this mount point. Can improve performance under certain conditions. Ubuntu uses the "relatime" option, which is similar.

» **sync** Forces all writes to the filesystem to be synchronous (to be flushed to disk immediately).

» **nosuid** This prevents the "setuid" and "setgid" attributes from taking effect. It's an option that's often used on removable media. It prevents Mr Bad Guy from getting root by turning up with a CD containing, for example, a shell owned by root with the setuid bit on it.



› fstab entry for a removable device.

Protect your data

Worried about the possibility of hard-drive theft? Keep data from prying eyes by using these encryption methods

Right, so you're up to date with the security updates on your servers. You've disabled direct root logins, set a *Grub* password, and screwed down the firewall so tight that even SSH packets aren't allowed in until they've wiped their feet. You feel pretty secure. And then a disgruntled employee wanders into your server room with a screwdriver, removes your hard drive and gets your entire corporate database. It's not only servers that are vulnerable; it's likely your employees carry copies of company confidential stuff on their laptops too. Laptops and memory sticks are especially easy to steal. There have been enough high-profile reports of data theft in the media over the last year or two to draw attention to the dangers.

Don't despair. There are ways to ensure confidentiality of data even if the Bad Guy has your hard drive in his evil clutches. And while the mathematics behind it all may be beyond most of us (including me), putting encryption technology into practice is easy enough. In this tutorial we'll take a look at two different ways of encrypting filesystems using Ubuntu.

Plan A: Encrypting the whole filesystem

In Linux-speak, a block device is a storage device that can be accessed randomly, on a block-by-block basis. Block devices are important because Linux uses them to build filesystems on, and the most obvious example of a block device is a disk drive. However, the Linux 2.6 kernel includes a device mapper layer that allows the creation of virtual layers of block devices on top of real block devices like disks. These virtual block devices can do different things such as striping, mirroring, taking snapshots, encryption and so on. Logical volume management (LVM) and software RAID are both built as device-mapper layers.

Of particular interest here is the *dm-crypt* device mapper that encrypts a virtual block device on to an underlying disk partition. The technology has been in the kernel for a while, but with the Alternate Ubuntu 8.10 CD, Canonical has made it much easier to deploy. This CD does not provide a Live distribution, but uses a more traditional, text-based installer. In particular, the partitioner offers the option of putting pieces of your filesystem on encrypted logical volumes.

The easiest approach is to select "Guided – use entire disk and set up encrypted LVM" at the main partitioner screen.

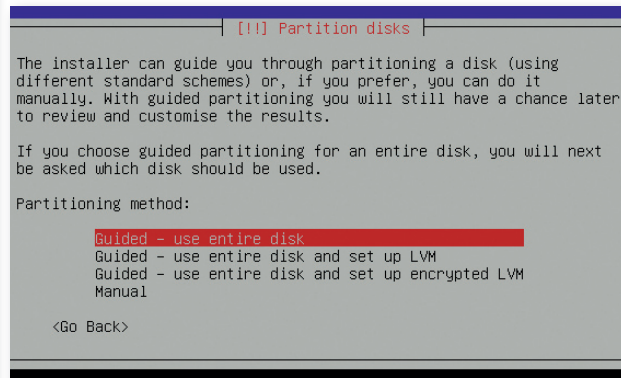
Plan B: The ecryptfs stacked filesystem

Whole-partition encryption at the block device level (as I have just described) certainly defeats the hard drive thief, but has a couple of disadvantages. First, you need to supply the passphrase at boot time, making unattended reboots impossible. Second, it's not obvious what to do about backups; for example, how do you make incremental backups to untrusted remote backup machines? And third, it does nothing to give individual users a nice warm fuzzy feeling that their data is safe from the prying eyes of other users, as all users' files are protected using a single system-wide passphrase. Once the system is booted, and the system administrator has "unlocked" the encrypted partition, the system behaves just as if it were a normal partition.

There is an alternative approach to encryption – the ecryptfs filesystem – that does address these issues. Ecryptfs is a stacked filesystem: it layers an 'upper' filesystem on top of an existing, mounted 'lower' filesystem. The upper level provides the unencrypted view of the files; the lower level is the encrypted view, the one that is actually stored on disk. Sorry about the missing 'n' in the name by the way. I guess it has gone to join the missing 'n' in 'umount'; I just hope they are having fun together!

Ecryptfs is registered as a virtual filesystem type within the Linux kernel, and filesystems can be mounted by specifying a type of 'ecryptfs' to the mount command. For example, if I do this:

```
cd /home/chris
mkdir lower
mkdir upper
```



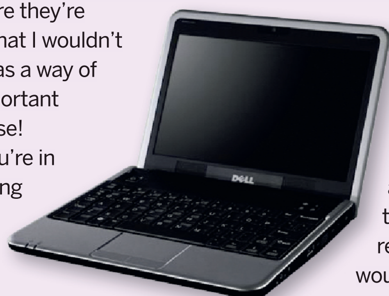
› **Installing from the Ubuntu 8.10 'Alternate' CD offers an easy way to set up an encrypted filesystem.**

A few warnings

Do be careful – for either of the encryption techniques we've discussed, if you forget the relevant pass phrase, your data is truly irrecoverable. Forgetting a regular login password, even root's, isn't really serious – it's easy enough to do some sort of rescue boot and reset it. Forgetting your encryption pass phrase is an entirely different and more severe situation. You are (if you'll allow a rather technical term) stuffed. I don't know enough cryptography or mathematics to work out the numbers behind a brute force attack, but I'm sure they're gigantic enough that I wouldn't want to rely on it as a way of recovering an important corporate database!

Secondly, if you're in the habit of carrying your laptop home in hibernate or sleep mode, so that Linux is still booted, neither of our techniques offer any protection at all. As long as the guy who steals the laptop doesn't reboot, or log you out, he has full access to your data. So, get into the habit of shutting the laptop down entirely before abandoning it on the luggage rack of the 18.06 from Southend-on-Sea to Liverpool Street.

Finally, to end on a slightly grim note, section 49 of the Regulation of Investigatory Powers Act 2000 Part 3 (investigation of electronic data protected by encryption etc) specifies conditions under which criminal investigators can legally require users to give over their encryption keys. But I assume you would have to have done something really bad before that would happen.



LXF114.sysadmin 64 21/11/08 5:40:31 pm

```
sudo mount -t ecryptfs lower upper
```

I'll end up with the unencrypted view (**/home/chris/upper**) stacked on top of the encrypted view (**/home/chris lower**). You can verify this using the *mount* command:

```
$ mount | grep ecrypt
/home/chris/lower on /home/chris/upper type ecryptfs
(rw,ecryptfs_sig=dbcc9a3da3399a69,ecryptfs_
cipher=aes,ecryptfs_key_bytes=16,)
```

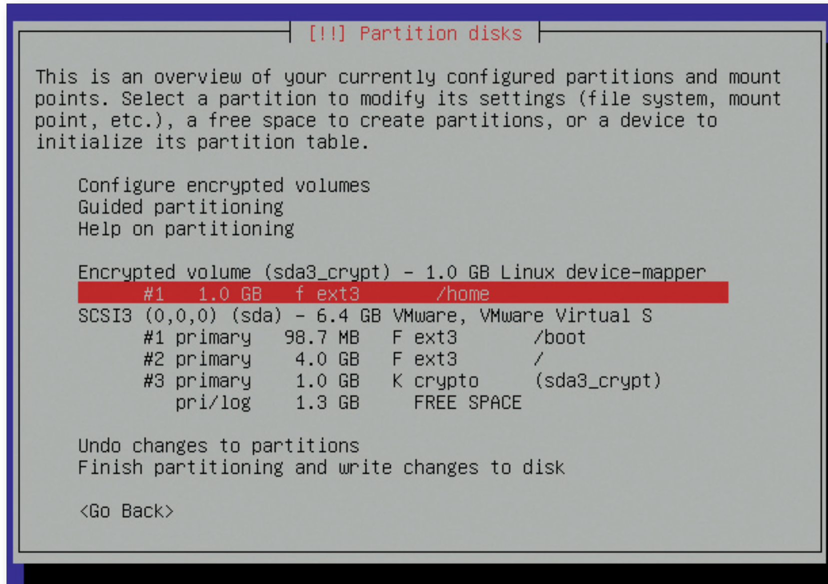
With this mount in place, for each (plaintext) file that I create in **/home/chris/upper**, a corresponding encrypted file will exist in **/home/chris/lower**. For example:

```
$ echo "Attack at dawn" > upper/battleplan
$ ls -l lower upper
lower:
total 12
-rw-r--r-- 1 chris chris 12288 2008-10-09 13:40 battleplan
upper:
total 0
-rw-r--r-- 1 chris chris    15 2008-10-09 13:40 battleplan
```

There are a couple of things worth noting here. First, the file names in the lower directory are the same as those in the upper directory. This is perhaps not ideal... the enemy who steals my hard drive will not find out what my battle plan is, but he will discover that I have one, which is more than I might wish. Second, notice that the file in the lower directory is much larger – there's an overhead of approximately 12 kilobytes on every file. The extra content at the front of each lower file contains cryptographic metadata about the file. Having this information in the file contents makes it convenient to transfer or back up the files while preserving all the information necessary to access them later, but it's an overhead you'll need to keep in mind if you plan to have a large number of very small private files.

> **"Canonical has tried to make ecryptfs as painless to use as possible."**

› At boot time, you're prompted to enter the passphrase to unlock the partition you have encrypted.



› Manual configuration of encrypted partitions needs care. Here, /home is mounted from the encrypted partition sda3_crypt.

Note that, when the ecryptfs mount is in place, ecryptfs will not prevent other logged-in users from seeing your private files if the access permissions on your "upper level" directory are too open. Your private directory should be mode 700.

The Ubuntu way

In keeping with its motto of "Linux for human beings", in Ubuntu 8.10 Canonical has tried to make ecryptfs as painless to use as possible. There is a script called *ecryptfs-setup-private* that sets everything up for a user, creating the upper and lower directories (called **~/Private** and **~/.Private** respectively) and setting their permissions. You're prompted for an encryption key, or you can let the system choose one at random. In either case, the key is itself encrypted ('wrapped') using a pass phrase of your choosing. Once set up, the ecryptfs filesystem can be mounted and unmounted using two simple scripts, *ecryptfs-mount-private* and *ecryptfs-umount-private*, which in effect wrap the **mount -t ecryptfs** command shown earlier.

Ubuntu things one step further. It integrates the mounting and unmounting operations into the login / logout process by hooking in to a new PAM module called **pam_ecryptfs.so**. This module is invoked at login time; it 'unwraps' the encryption key and uses it to perform the ecryptfs mount of **~/.Private** on to **~/Private**. For this to work, the pass phrase used to wrap the encryption key must be the same as your login password.

If you install from the 'Alternate' CD, the installer screen that creates the initial user account offers the option to set up ecryptfs. This option does not appear to be offered on the desktop CD, though I should say that while was experimenting with this I was working with the beta release of Ubuntu 8.10, and things might have changed by the time of the final release.

One rather whimsical piece of the implementation in Ubuntu is the presence of a file named **THIS DIRECTORY HAS BEEN UNMOUNTED TO PROTECT YOUR DATA – Run mount. ecryptfs_private to mount again** in the **Private** directory. Of course, this file only shows up when the ecryptfs mount is not in place. This file turns out to be a symbolic link to **mount. ecryptfs_private**. LXF

LXF114.sysadmin 65 21/11/08 5:40:35 pm