Complete Hard Disk Encryption Using FreeBSD's GEOM Framework

Marc Schiesser
m.schiesser [at] quantentunnel.de

October 20th 2005

Abstract

Most technologies and techniques intended for securing digital data focus on protection while the machine is turned *on* – mostly by defending against remote attacks. An attacker with *physical* access to the machine, however, can easily circumvent these defenses by reading out the contents of the storage medium on a different, fully accessible system or even compromise program code in order to leak encrypted information.

Especially for mobile users, that threat is *real*. And for those carrying around sensitive data, the risk is most likely *high*.

This paper describes a method of mitigating that particular risk by protecting not only the data through encryption, but also the applications and the operating system from being compromised while the machine is turned *off*.

The platform of choice will be FreeBSD, as its GEOM framework provides the flexibility to accomplish this task. The solution does not involve programming, but merely relies on the tools already provided by FreeBSD.

Table of Contents

1 Background & motivation....2
2 Partial disk encryption....3
  2.1 File-based encryption....4
  2.2 Partition-based encryption....5
  2.3 The leakage risk....5
  2.4 New attack vectors....6
3 Complete disk encryption....6
  3.1 Tools provided by FreeBSD....6
  3.2 The problem with complete disk encryption....7
  3.3 Requirements....8
  3.4 Complete hard disk encryption using GBDE....8
    3.4.1 Erasing previously stored data....8
    3.4.2 Initialization & the lockfile....9
    3.4.3 Attaching the encrypted medium....9
    3.4.4 Partitioning....10
    3.4.5 Creating the filesystem....11
    3.4.6 Installing FreeBSD....11
    3.4.7 Preparing the removable medium....12
    3.4.8 The kernel modules....12
    3.4.9 The problem with GBDE....13
    3.4.10 The memory disk....13
    3.4.11 Populating the memory disk filesystem....14
    3.4.12 The booting process....14
    3.4.13 Creating the symlinks....15
    3.4.14 Integrating the memory disk image....15
    3.4.15 The swap partition....16
    3.4.16 Post-installation issues....16
  3.5 Complete hard disk encryption using GELI....16
    3.5.1 Readying the hard disk....17
    3.5.2 Improvements and new problems with GELI....17
    3.5.3 Initializing, attaching and partitioning....18
    3.5.4 Filesystem creation and system installation....19
    3.5.5 The removable medium....19
    3.5.6 Mounting the encrypted partition....19
4 Complete hard disk encryption in context....20
  4.1 New defenses & new attack vectors – again....20
  4.2 Trade-offs....22
  4.3 GBDE vs. GELI....23
5 Conclusion....23
References & further reading....24

1 Background & motivation

As more and more data enters the digital world, appropriate measures must be taken in order to protect it.

Considering the ever-increasing number of networked devices and the inherent exponential growth of the Internet, it is imperative that a large amount of effort go into securing devices against remote attacks. Common technologies and techniques include firewalls, intrusion detection systems (IDS), encryption of all kinds of network transmissions as well as hardening network stacks and fixing buffer overflows.

At the same time, we are witnessing increasingly sophisticated and complex *mobile* devices such as PDAs, smartphones and cell phones becoming pervasive and assuming all kinds of important tasks. Between the general-purpose laptop and the (once) special-purpose cell phone, pretty much anything in between is available.

As people use these devices, they also generate data – either explicitly or implicitly. Explicitly stored data might for example include: entering a meeting into the electronic schedule, storing a telephone number and associating a name with it, or saving an email message draft in order to finish it later.

But then there is also the data which is stored implicitly. Examples include the history of the telephone numbers called or received, browser caches, recently accessed files, silently by the software archived or backed-up data such as email messages, log files and so on.

Even if the user remembers to delete the explicitly stored files after they are no longer needed, it is possible to trace a lot of his or her activity on the device by looking at the aforementioned, implicitly stored data. The more sophisticated the device is, the more such data will usually be generated, mostly without the user's knowledge.

In terms of performance, laptop computers hardly lag behind their desktop counterparts – enabling them to run the same powerful and complex software. It also means that the users tend to generate far more data – both explicitly and implicitly – than on simpler devices.

In addition to being exposed to remote attacks, laptop users are also faced with an increased exposure of the machine itself. While stationary computers are physically accessible by usually only a limited number of people, a laptop computer is *intended* to be used anywhere and anytime.

This paper does not try to provide any solutions to mitigating the risks of remote attacks. Instead, it concentrates on the risks posed by attackers with *physical* access to the device. An attacker with physical access to a machine can either:

- boot his own operating system, thus circumventing restrictions such as login procedures, filesystem and network access control and sandboxes
- or remove the hard disk from the targeted machine and install it in a system which is under the control of the attacker – in case the target's booting sequence is protected (e.g. by a BIOS password)

Unfortunately, most people and companies take quite lax an approach when it comes to protecting their data *in-storage,* while the machine is turned off. The following quotes illustrate just how serious a problem the lack of in-storage encryption can become:

- „Thieves stole computer equipment from Fort Carson containing soldiers' Social Security numbers and other personal records, the Army said ...” [Sarche, 2005]
- „Personal devices "are carrying incredibly sensitive information," said Joel

Yarmon, who, as technology director for the staff of Sen. Ted Stevens (R-Alaska), had to scramble over a weekend last month after a colleague lost one of the office's wireless messaging devices. In this case, the data included "personal phone numbers of leaders of Congress. . . . If that were to leak, that would be very embarrassing," Yarmon said.” [Noguchi, 2005]

- „A customer database and the current access codes to the supposedly secure Intranet of one of Europe's largest financial services group was left on a hard disk offered for sale on eBay.” [Leyden, 2004]
- „ ... Citigroup said computer tapes containing account data on 3.9 million customers, including Social Security numbers, were lost by United Parcel Service.” [Reuters, 2005]
- „Earlier this year, a laptop computer containing the names and Social Security numbers of 16,500 current and former MCI Inc. employees was stolen from the car of an MCI financial analyst in Colorado. In another case, a former Morgan Stanley employee sold a used BlackBerry on the online auction site eBay with confidential information still stored on the device. And in yet another incident, personal information for 665 families in Japan was recently stolen along with a handheld device belonging to a Japanese power-company employee.” [Noguchi, 2005]
- „ ... trading firm Ameritrade acknowledged that the company that handles its backup data had lost a tape containing information on about 200,000 customers. ” [Lemos, 2005]
- „MCI last month lost a laptop that stores Social Security numbers of 16,500 current and former employees. Iron Mountain, an outside data manager for Time Warner, also lost tapes holding information on 600,000 current and former Time Warner workers.” [Reuters, 2005]

Even though the number of press articles reporting damage due to stolen mobile computers – or more specifically: storage media – does not reach the amount of publicity that remotely attacked and compromised machines provoke, it must also be taken into account that data on a laptop does not face as much exposure as it does on an Internet server.

A laptop computer can be insured and data regularly be backed up in order to limit the damage in case of *loss or theft*; but protecting *the data from unauthorized access* requires a different approach.

2 Partial disk encryption

Encryption of in-storage data (as opposed to in-transmission) is not a completely new idea, though. Several tools for encrypting *individual files* have been around for quite some time. Examples include the famous PGP (Pretty Good Privacy) as well as its free counterpart GnuPG and the somewhat less known tools AESCrypt[1] and NCrypt[2].

More sophisticated approaches aim towards encrypting *entire partitions*. The

1 http://aescrypt.sourceforge.net/
2 http://ncrypt.sourceforge.net/

vncrypt[3] project is an example that takes this approach.

2.1 File-based encryption

The idea is that the user can decide for each file individually, whether and how it is to be encrypted. This has the following implications:

- CPU cycles can be saved on data that the user decides is not worth the effort. This is an advantage, since encryption requires a lot of processing power. It also allows the user to choose different keys for different files (although reality usually reflects the opposite phenomenon).
- Meta data is not encrypted. Even if the file's contents are sufficiently protected, information such as file name, ownership, creation and modification date, permissions and size are still stored in the clear. This represents a risk which is not to be underestimated.

The usability of in-storage encryption largely depends on how transparent the encryption and decryption process is performed to the user. In order to minimize user interaction, the relevant system calls must be modified to perform the cryptographic functions accordingly. That way, neither the user nor the applications must make any additional effort to process encrypted file content, since the kernel will take care of this task.

If system call modification is not going to take place, any program required to process encrypted data must either be modified to perform the necessary cryptographic functions itself or it must rely on an external program for that task. This conversion between *cipher text* and *plain text* – and vice versa – is hardly possible without requiring any user interaction.

Scenario: file-based encryption of huge files

The file might be a database or multimedia container: if the cryptographic functions are not performed on-the-fly (usually through modified system calls), the entire file content must be temporarily stored in plain text – therefore consuming twice the space.

Then the *unencrypted* copy must be opened by the application. After the file has been closed, it obviously must be encrypted again – unless no modification has taken place. The application will therefore save the data in plain text, which must then be encrypted and written out in its cipher text form again – but by a program capable of doing the appropriate encryption.

The unencrypted (temporary) copy could of course just be unlinked (removed from the name space), but in that case the unencrypted data would still remain on the medium until physically completely overwritten. So, if one wants to really destroy the temporary copy, several overwrites are required – which can consume a lot of time with large files. Therefore, a lot of unnecessary I/O must be performed.

Scenario: file-based encryption of many files

If one wants to encrypt more than just a small bunch of files, it actually does not matter how small or large they are – the procedure described above still must be adhered to unless encryption and decryption is performed on-the-fly.

Aside from its lack of scalability, file-based encryption also suffers from the leakage

3 http://sourceforge.net/projects/vncrypt/

risk “phenomenon” – which will be discussed in chapter 2.3. In most cases, encryption is therefore either abandoned or the following, more effective and efficient scheme is chosen.

2.2 Partition-based encryption

Obviously, creating a temporary plain text copy of an entire partition each time the data is accessed, is hardly a sane solution. The system must therefore be able to perform the encryption and decryption on-the-fly, as it has been implemented in the FreeBSD kernel for GBDE [Kamp, 2003a] and GELI [Dawidek, 2005a] and `cgd(4)` in NetBSD [Dowdeswell & Ioannidis, 2003]. A few third party add-ons also exist. One example of this is the aforementioned `vncrypt`, which was developed at Sourceforge.

`vncrypt` is, however, in a further sense still file-based, because the encrypted partition is only a mounted *pseudo*-device created via the `vn(4)` facility from a *regular* file. This file holds all the partition's data in encrypted form – including meta data. OpenBSD's `vnconfig(8)` provides a similar feature [OpenBSD, 1993].

One aspect associated with partition-based encryption is that its set-up process is usually more extensive than it is for file-based encryption. But once it has been done, partition-based encryption is far superior to the file-based encryption scheme. All data going to the particular partition is – by default – stored encrypted. As both encryption and decryption is performed *transparently* to the user and *on-the-fly*, it is also feasible to encrypt both large amounts of files and large amounts of data.

But unfortunately, this scheme it not perfect either.

2.3 The leakage risk

As obvious as it may sound, partition-based encryption protects only what goes onto the encrypted partition. The following scenario highlights the particular problem.

<u>Scenario: editing a sensitive document stored on an encrypted partition</u>

A mobile user needs to have a lot of data at his immediate disposal. Since some information is sensitive, he decides to put it on an encrypted partition.

Unfortunately, the encryption becomes basically useless as soon as encrypted files are opened with applications that create temporary copies of the files currently being worked on, often in the **/tmp** directory. So unless the user happens to have **/tmp** encrypted, his sensitive data is *leaked* to an *unencrypted* part of the medium. Even if the application deletes the temporary copy afterwards, the data still remains on the medium until it is *physically* overwritten. Meta data such as file name, size and ownership may also have leaked and may therefore remain accessible for some time.

This phenomenon happens equally implicitly with printing. Even if the application itself does not leak any data, the spooler will usually create a Postscript document in a subdirectory of **/var/spool/lpd/**, which is not encrypted unless specifically done so.

Even though it is possible to symlink the “hot” directories such as **/tmp**, **/var/tmp,** as well as the complete **/home** or **/var/spool/lpd/** to directories on the encrypted partition, the leakage risk can never be avoided completely. It is something that users of partition-based encryption just have to be aware of and learn to live with by minimizing the amount of leaked data as much as possible.

The leakage risk is also another reason why file-based encryption is virtually useless. While this issue is certainly a problem for sensitive data, there is a *far* bigger problem, which so far has been quietly ignored.

2.4 New attack vectors

The point of storing data is to be able to retrieve it at some later date. So far, everything that was discussed, was based on the assumption that both the OS and the applications were stored *unencrypted* – there is also no point in doing otherwise as long as the data itself is not encrypted:

- if data cannot[4] be destroyed, stolen or modified *remotely*, a dedicated attacker will find a way to gain local (physical) access to the system
- if login procedures, filesystem access control and other restrictions imposed by the OS and applications cannot be defeated or circumvented, the attacker will boot his/her own OS
- if the booting sequence on the machine is protected, the attacker will remove the hard disk and access it from a system under his control
- if the data on the hard disk is encrypted and a brute-force attack is not feasible, then the attacker will most likely[5] target the OS and/or the applications

The key motivation behind complete disk encryption is illustrated in the last point: the OS and the applications are now the target. Instead of breaking the encryption, an attacker can try and subvert the kernel or the applications, so they leak the desired data or the encryption key.

The goal is therefore to encrypt the OS and all the applications as well. Just as any security measure that is taken, this scheme involves trade-offs, such as less convenience and decreased performance. These issues will be discussed later. Every user considering this scheme must therefore decide for him- or herself, whether the increase in security is worth the trade-offs.

3 Complete disk encryption

3.1 Tools provided by FreeBSD

The platform of choice here is FreeBSD, because it comes with a modular, very powerful I/O framework called GEOM [Kamp, 2003b] since the release of the 5.x branch. The 5.x branch underwent several major changes compared to the 4.x branch and was not declared -STABLE until the 5.3-RELEASE in November 2004. The 5.x branch did, however, feature a GEOM class and a corresponding userland utility called GBDE (GEOM Based Disk Encryption) as early as January 2003 when 5.0-RELEASE came out. GBDE was specifically designed to operate on the sector level and is therefore able to

4 perfect security is not possible; therefore 'cannot' should rather be read as 'cannot easily enough'

5 tampering with the hardware is of course also possible, for example with a hardware keylogger; defending against this kind of attack is not discussed in this paper

encrypt entire partitions and even hard disks or other media.

When the 5.x branch was finally declared -STABLE and therefore ready for production use, 6.x became the new developer branch, carrying all the new, more disruptive features. Into this branch added was also a new module and userland utility called GELI [Dawidek, 2005b]. In addition to containing most of the GBDE features, GELI was designed to enable the kernel to mount the root filesystem (/) from an encrypted partition. GBDE does not allow to do this and therefore requires a “detour“ in order to make complete hard disk encryption work.

This paper will discuss the realization of complete hard disk encryption with both tools without having to rely on programming. GELI is a more elegant solution, because it was designed with this application in mind. GBDE, on the other hand, has seen more exposure because it has been available for much longer then GELI and therefore is more likely to have received more testing. Using GBDE for complete hard disk encryption also illustrates some interesting problems inherent with the booting process and how these can be solved.

Which approach is in the end chosen, is left to the user. The following table lists the most important features of GBDE and GELI [Dawidek, 2005b].

	GBDE	***GELI***
First released in FreeBSD	5.0	6.0
Cryptographic algorithms	AES	AES, Blowfish, 3DES
Variable key length	No	Yes
Allows kernel to mount encrypted root partition	No	Yes
Dedicated hardware encryption acceleration	No	Yes, crypto(9)
Passphrase easily changeable	Yes	Yes
Filesystem independent	Yes	Yes
Automatic detach on last close	No	Yes

Table 1: the most important GBDE and GELI features

3.2 The problem with complete disk encryption

There are cases in which it is desirable to encrypt the whole hard disk – especially with mobile devices. This also includes the encryption of the kernel and the boot loader.

Today's computers, however, cannot boot encrypted code. But if the boot code is not encrypted, it can easily be compromised. The solution is therefore to store all code necessary for booting and then mounting the encrypted hard disk partition on a medium that can be carried around *at all times.*

While virtually any *removable* medium is easier to carry around than a *fixed* one, USB memory sticks are currently the best solution. They provide plenty of space at affordable prices, are easily rewritable many times and easy to use since operating systems treat them like a hard disk. But most importantly, they are small and light.

Obviously, putting the boot code on a removable medium instead of the fixed hard disk does not solve the problem of compromise – the risk is simply shifted toward the removable medium. But since that medium can be looked after a lot more easily, there *is* a considerable benefit to the user.

3.3 Requirements

Independent of whether GBDE or GELI is used, the following things are required:

- A bootable, removable medium. It will carry the boot code as well as the kernel. This medium is preferably a USB memory stick, because it is small, light and offers a lot of easily rewritable space.
- The device intended for complete disk encryption. Is is very important that this device is capable of booting from the removable medium mentioned above. Especially older BIOSes may not be able to boot from USB mass storage. Bootable CDs will probably work on most machines. Although they work equally well (r/w access is *not* a requirement for operation), they are harder to set up and maintain.
- In order to set up and install everything, a basic FreeBSD system is required. The FreeBSD installation discs carry a “live filesystem” – a FreeBSD system which can be booted directly from the CD. It can be accessed via the `sysinstall` menu entry 'Fixit'.

All following instructions are assumed to be executed from the aforementioned “live filesystem” provided by the FreeBSD installation discs.

Before proceeding any further, the user is strongly urged to back up all data on the media and the devices in question.

*Furthermore, it will be assumed that the hard disk to be encrypted is accessible through the device node **/dev/ad0** and the removable (USB) medium through **/dev/da0**. These paths **must** be adjusted to the actual set-up!*

3.4 Complete hard disk encryption using GBDE

3.4.1 Erasing previously stored data

Before a medium is set up to store encrypted data, it is important to completely erase all data previously stored on it. *All* data on it has to be *physically* overwritten – ideally multiple times. Otherwise the data that has previously been stored unencrypted would still be accessible at the sector level of the hard disk until overwritten by new data. There are two ways to wipe a hard disk clean:

```
# dd if=/dev/zero of=/dev/ad0 bs=1m
```

overwrites the entire hard disk space with zero values. The parameter `bs` sets the block size to 1 MB – the default (512 B) would take a very long time with large disks.

```
# dd if=/dev/random of=/dev/ad0 bs=1m
```

does the same thing, but uses entropy instead of zero values to overwrite data. The problem with the first approach is that it is quite obvious which parts of the medium carry data and which ones are unused. Attackers looking for potential clues about the encryption key can often exploit this information.

In most cases, however, this should not be a major risk. The downside of using entropy is that it requires *far* more processing power than simply filling the hard disk space with zero values. The required amount of time may therefore be too great a trade-off for the additional increase in security – especially on older, slower hardware.

3.4.2 Initialization & the lockfile

After the hard disk to be encrypted has been wiped clean, it can be initialized for encryption. This is done using the `gbde(8)` command:

```
# gbde init /dev/ad0 -L /very/safe/place/lockfile
Enter new passphrase:
Reenter new passphrase:
```

The lockfile is very important, as it is needed later to access the master key which is used to encrypt all data. The 16 bytes of data stored in this lockfile could also be saved in the first sector of the medium or the partition, respectively. In that case, however, only the passphrase would be required to get access to the data. The passphrase – however strong it is – will face intensive exposure with mobile devices as it must be typed in each time the system is booted up. It therefore cannot be realistically guaranteed that the passphrase remains only known to those authorized to access the protected system and data.

But since an additional medium is needed anyway in order to boot the core OS parts, it might as well be used as a storage area for the lockfile – effectively functioning as a kind of access token.

With this scheme, two things are required to get access to the data: the passphrase *and* the lockfile. *If the lockfile is unavailable (lost or destroyed), even knowledge of the passphrase will not yield access to the data!*

3.4.3 Attaching the encrypted medium

After the initialization is complete, the encrypted hard disk must now be *attached* – meaning that the user has to provide both the passphrase and the lockfile to `gbde`, which in turn provides (or denies) access to the decrypted data.

```
# gbde attach /dev/ad0 -l /very/safe/place/lockfile
Enter passphrase:
```

If the passphrase and the lockfile are valid, `gbde` creates an additional device node in the **/dev** directory. This newly created node carries the name of the just attached device ('ad0') plus the suffix '.bde'.

- **/dev/ad0** can be used to access the *actual* contents of the hard disk, in this case the *cipher text*
- **/dev/ad0.bde** is an abstraction created by GBDE and allows *plain text* access to the data

All reads from and writes to the .bde-node are automatically de-/encrypted by GBDE and therefore no user interaction is required once the correct passphrase and lockfile has been provided.

The **ad0.bde** node acts just like the original **ad0** node: it can be partitioned using `bsdlabel(8)` or sliced with `fdisk(8)`, it can be formated as well as mounted.

It is important to keep in mind that once a storage area has been attached and the corresponding .bde device node for it has been created, it remains that way until it is explicitly *detached* via the gbde command or the system is shut down. *In the period between attaching and detaching, there is no additional protection by GBDE.*

3.4.4 Partitioning

The next step is to partition the hard disk. This is usually done using sysinstall(8) – which, unfortunately, does not support GBDE partitions and fails to list device nodes with a .bde suffix. Therefore, this work has to be done using the tool bsdlabel.

```
# bsdlabel -w /dev/ad0.bde
# bsdlabel -e /dev/ad0.bde
```

First, a standard label is written to the encrypted disk, so that it can be edited afterwards. bsdlabel will display the current disk label in a text editor, so it can be modified. In order to make the numbers in the following example easier to read, the disk size is assumed to be 100 MB. The contents of the temporary file generated by bsdlabel might look like this:

```
# /dev/ad0.bde:
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
  a:  198544      16    unused        0     0
  c:  198560       0    unused        0     0        # "raw" part, don't edit
```

Each partition occupies one line. The values have the following meaning:

column	***description***
1	a=boot partition; b=swap partition ; c=whole disk; d, e, f, g, h=freely available
2 and 3	partition size and its offset in sectors
4	filesystem type: 4.2BSD, swap or unused
5, 6 and 7	optional parameters, no changes required

Table 2: bsdlabel(8) file format

After the temporary file has been edited and the editor closed, bsdlabel will write the label to the encrypted hard disk – provided no errors have been found (e.g. overlapping partitions).

It is important to understand the device node names of the newly created partitions. The encrypted boot partition (usually assigned the letter 'a'), is now accessible via device node **/dev/ad0.bdea**. The swap partition is **ad0.bdeb** and so on. Just as adding a boot partition to an unencrypted disk would result in a **ad0a** device node, adding an encrypted *slice* holding several partitions *inside* would result in **ad0s1.bdea**, **ad0s1.bdeb** and so on.

An easy way to keep the naming concept in mind is to remember that everything written *after* the .bde suffix is encrypted and therefore hidden even to the kernel until the device is attached.

For example: **ad0s1.bdea** means that the data on the first slice is encrypted – *including* the information that there *is* a boot partition inside that slice. If the slice is not attached, it is only possible to tell that there *is* a slice on the disk – neither the contents of the slice, nor the fact that there *is* at least one partition inside the slice can be unveiled.

In fact, the node **ad0s1.bdea** does not even exist until the slice has been successfully attached, because without having the key (and the lockfile), the kernel cannot know that there is a partition inside the encrypted slide.

Scenario: multiple operating systems on the same disk

It is also possible to have multiple operating systems on the same disk – each on its own slice. The slice containing FreeBSD can be encrypted *completely*, hiding even the fact that the FreeBSD slice contains multiple partitions *inside* (boot, swap, etc). This way, all data on the FreeBSD slice remains protected, while the other operating systems on the machine can function normally on their unencrypted slices. In fact, they cannot even compromise the data on the FreeBSD slice – even if an attacker manages to get root access to a system residing on an unencrypted slice.

3.4.5 Creating the filesystem

Now that device nodes for the encrypted partitions exist, filesystems can be created on them:

```
# newfs /dev/ad0.bdea
# newfs /dev/ad0.bded
```

etc.

Note that the swap partition does not need a filesystem; the 'c' partition represents the entire (encrypted) disk. This partition must *not* be formated or otherwise be modified!

3.4.6 Installing FreeBSD

Now that the filesystems have been created, FreeBSD can be installed on the encrypted hard disk. Usually, this would be done using `sysinstall` again. But just as `sysinstall` cannot partition and format encrypted media, it cannot install the system on them. The distributions that comprise the FreeBSD operating system, therefore have to be installed *manually*.

The FreeBSD installation disc contains a directory that is named after the release version of the system, for example: 5.4-RELEASE, 6.0-BETA etc. In this directory, each distribution – such as `base`, `manpages` or `src` – has its own subdirectory with an `install.sh` script. The `base` distribution is required, all others are optional.

In order to install the distributions, the encrypted boot partition (and others, if used for example for **/usr**) has to be mounted and the environment variable DESTDIR set to the path where the encrypted boot partition has been mounted. Then all distributions can be installed using their respective `install.sh` script.

The following example assumes that the encrypted boot partition **/dev/ad0.bdea** has been mounted on **/fixed** and the FreeBSD installation disc on **/dist** (the “live-filesystem” default). If the live-filesystem is used, the **/fixed** directory is easy to create because the root (**/**) is a memory disk.

```
# mount /dev/ad0.bdea /fixed
# export DESTDIR=/fixed/
# cd /dist/5.4-RELEASE/base && ./install.sh
You are about to extract the base distribution into /fixed - are you SURE
you want to do this over your installed system (y/n)?
```

After all desired distributions have been installed, there is a complete FreeBSD installation on the encrypted disk and the swap partition is also ready. But since this system cannot be booted from the hard disk, it is necessary to set up the removable medium.

3.4.7 Preparing the removable medium

As it has already been discussed, this medium will not be encrypted. This means that the standard tool `sysinstall` can be used. The removable medium needs one partition of at least 7 MB. This provides only space for the kernel, some modules and the utilities required for mounting the encrypted partition(s). All other modules such as third party drivers need to be loaded *after* `init(8)` has been invoked.

If it is desired that all FreeBSD kernel modules be available on the removable medium and thus are loadable *before* `init` is called, the slice should be at least 25 MB in size.

The removable medium can be sliced using `fdisk` or via 'Configure' - 'Fdisk' in the `sysinstall` menu. The changes made to the medium can be applied immediately by hitting 'W'. After that, the slice has to be labeled (`sysinstall` menu 'Label'). All the space on the slice can be used for the boot partition, since the swap partition on the encrypted hard disk will be used. The mount point for the boot partition does not matter; this text, however, will assume that it has been mounted on **/removable**.

`sysinstall` then creates the partition, the filesystem on it and also mounts it on the specified location (**/removable**). After that, `sysinstall` can be quit in order to copy the files required for booting from the removable medium. All that is required is the **/boot** directory – it can be copied from the installation on the encrypted hard disk:

```
# cp -Rpv /fixed/boot /removable
```

3.4.8 The kernel modules

User interaction with GBDE is done through the userland tool `gbde(8)`, but most of the work is carried out by the kernel module **geom_bde.ko**. This module must be loaded before the userland utility is called.

Usually, kernel modules are loaded by `loader(8)` based on the contents of the file **/boot/loader.conf** – then control is passed over to the kernel. In order to have the GBDE module loaded before `init` is executed, it must be loaded in advance by `loader`. The following instruction adds the GBDE kernel module to the `loader` configuration file on the removable medium:

```
# echo geom_bde_load=\“YES\“>> /removable/boot/loader.conf
```

In case additional kernel modules are needed at boot time, they must be copied to **/boot/kernel/** and appropriate entries must be added to **/boot/loader.conf** (this file overrides the defaults in **/boot/defaults/loader.conf**).

In order to save space on the removable medium and also to speed up loading, all kernel modules and even the kernel itself can be gzipped.

```
# cd /removable/boot/kernel
# gzip kernel geom_bde.ko acpi.ko
```

Binary code compresses to about half of the original size and thus brings a noticeable decrease in loading time. The modules which will not be used or later will be loaded from the hard disk can be deleted from the removable medium.

It is important, however, that the code on the removable medium (kernel, modules, etc) is kept in sync with the system on the hard disk.

3.4.9 The problem with GBDE

As discussed earlier, GBDE has been designed with the encryption of partitions and even entire media in mind. Unfortunately, however, the **geom_bde.ko** module does *not* allow the kernel to mount an encrypted partition as the root filesystem.

This is because the passphrase must be provided through the utility *in user space* – even though the module obviously operates *in kernel space*. So, by the time the kernel must mount the root filesystem, the user has not even had the possibility of providing the passphrase and attaching the encrypted device.

There are two solutions to this problem:

- The kernel must be modified to allow mounting of an encrypted root filesystem *by asking for the passphrase in kernel space*. This way, the device node which gives access to the decrypted data (the .bde device node) would be available *before* `init` is started and could be specified in **/etc/fstab** as the root file-system. The new facility – GELI – has implemented this scheme and therefore makes it a lot easier than the second solution.
- The second solution is not really a solution, but more a “hack”, as the shortcomings of GBDE are not solved but avoided. The only conclusion is therefore that the root filesystem *cannot* be encrypted and that the filesystem(s) on the hard disk – although encrypted – must be mounted on directories residing in the *unencrypted* root filesystem. Attaching and mounting the encrypted hard disk must be done *after* the kernel has mounted an *unencrypted* root filesystem and started `init` and subsequently `gbde` from it.

3.4.10 The memory disk

Since the contents of the root filesystem will not be encrypted, it is best to store on it only what is needed to mount the encrypted partitions. Mounting the filesystem on the removable medium as the root filesystem means that the removable medium would have to be attached to the computer while the system is in use and therefore face a lot of unnecessary exposure.

The better solution is to store an image of a *memory disk* on the removable medium, which contains just the utilities necessary to mount the encrypted hard disk. The kernel can mount the memory disk as the root filesystem and invoke `init` on it, so that `gbde` can be executed. After the user has provided the passphrase to the encrypted partitions on the hard disk, the utilities on the memory disk can mount the encrypted partitions and then load the rest of the operating system from the encrypted hard disk – including all applications and user data.

First, an image for the memory disk must be created on the removable medium.

```
# dd if=/dev/zero of=/removable/boot/mfsroot bs=1m count=10
```

Then a device node for the image is needed, so that a filesystem can be created on it and then mounted.

```
# mdconfig -a -t vnode -f /removable/boot/mfsroot
md1
# newfs /dev/md1
# mount /dev/md1 /memdisk
```

If the output of `mdconfig(8)` differs from 'md1', the path in the following instructions must be adjusted. The assumed mounting point for the memory disk will be **/memdisk**.

3.4.11 Populating the memory disk filesystem

Since this filesystem is going to be mounted as the root filesystem, a directory must be created to serve as the mount point for the encrypted boot partition (**/memdisk/safe**).

```
# cd /memdisk
# mkdir safe
```

Some other directories also act as mount points and do not need to be symlinked to the encrypted hard disk. The directory **/etc**, however, is required, because the `rc(8)` script in it will be modified to mount the encrypted partitions.

```
# mkdir cdrom dev dist mnt etc
```

Now, the lockfile, which is needed to access the encrypted data, must be copied onto the removable medium – turning it into a kind of access token, without which the encrypted data cannot be accessed even with the passphrase available.

```
# cp /very/safe/place/lockfile /memdisk/etc/
```

It is important to remember that the lockfile is updated each time the passphrase is changed.

3.4.12 The booting process

After the kernel has been loaded from the removable medium it mounts the memory disk as the root filesystem and then executes `init`, the first process. `init` in turn calls `rc`, a script that controls the automatic boot process. Since `rc` is a text file rather than a binary executable, it can be easily modified to mount the encrypted boot partition *before* the majority of the system startup – which requires a lot of files – takes place. The `rc` script can therefore be copied from the installation on the hard disk and then be edited.

```
# cp /fixed/etc/rc /memdisk/etc/
```

The following commands have to be inserted after the line “export HOME PATH” (in 5.4-RELEASE: line 51) into **/memdisk/etc/rc**:

```
/rescue/gbde attach /dev/ad0 -l /etc/lockfile && \
/rescue/mount /dev/ad0.bdea /safe && \
/rescue/mount -w -f /dev/md0 / && \
/rescue/rm -R /etc && \
/rescue/ln -s safe/etc /etc
```

The commands first attach the encrypted boot partition, mount it on **/safe** and then erase the **/etc** directory from the memory disk, so that it can be symlinked to the directory on the encrypted disk.

Obviously, the utilities in the **/rescue** directory need to be on the memory disk. The **/rescue** directory is already part of a FreeBSD default installation and provides *statically linked* executables of the most important tools. Although the size of the **/rescue** directory seems at first glance to be huge (~470 MB!), there is in fact *one* binary which has been *hardlinked* to the various names of the utilities. The **/rescue** directory therefore contains about 130 tools which can be executed *without any dependencies on libraries*. The total size is less than 4 MB. Although this fits easily on the created memory disk, the directory cannot be just copied. The following example uses `tar(1)` in order to preserve the hardlinks.

```
# cd /fixed
# tar -cvf tmp.tar rescue
# cd /memdisk
# tar -xvf /fixed/tmp.tar
# rm /fixed/tmp.tar
```

3.4.13 Creating the symlinks

The files required for mounting the encrypted boot partition are now in place and the `rc` script has also been appropriately modified. But since the encrypted boot partition will not be mounted as the root (**/**), but in a subdirectory of the memory disk (**/safe**), all of the relevant directories must have entries in the root pointing to the actual directories in **/safe**.

```
# umount /fixed
# mount /dev/ad0.bdea /memdisk/safe
# cd /memdisk
# ln -s safe/* .
```

3.4.14 Integrating the memory disk image

The memory disk image now contains all the necessary data, so it can be unmounted and detached (if the memory disk image was not previously accessible through **/dev/md1**, the third line must be adjusted).

```
# umount /memdisk/safe
# umount /memdisk
# mdconfig -d -u1
```

In order to save space and to speed up the booting process, the memory disk image can also be gzipped, just like the kernel modules and the kernel itself:

```
# gzip /removable/boot/mfsroot
```

If the kernel was compiled with the MD_ROOT option – which is the case with the GENERIC kernel – it is able to mount the root from a memory disk. The file that holds the image of the memory disk must be loaded by the FreeBSD `loader`. This works almost the same way as with kernel modules, as the image must be listed in the configuration file **/boot/loader.conf**. Compared to executable code however, the memory disk image must be explicitly specified as such in the configuration file, so the kernel knows how to handle the file's contents. The following three lines are required in **/boot/loader.conf** on the removable medium:

```
        mfsroot_load="YES"
        mfsroot_type="mfs_root"
        mfsroot_name="/boot/mfsroot"
```

It is also important to note that there is no need to maintain an extra copy of the **/etc/fstab** file on the removable medium as the kernel automatically mounts the first memory disk that has been preloaded. Although this **/etc/fstab** issue is not a major problem, it is a necessary measure in order to make this scheme work with GELI – which is able to mount an encrypted partition as the root filesystem.

3.4.15 The swap partition

Although the swap partition has already been set up and is ready for use, the operating system does not yet know which device to use. It is therefore necessary to create an entry for it in the file **/etc/fstab**. This file must be stored on the hard disk, *not* the removable medium.

```
# mount /dev/ad0.bdea /fixed/
# echo “/dev/ad0.bdeb   none   swap   sw   0   0“ > /fixed/etc/fstab
```

Now, the system is finally ready and can be used by booting from the removable medium. The modified `rc` script will ask for the passphrase and then mount the encrypted partition, so that the rest of the system can be loaded.

3.4.16 Post-installation issues

Since the system on the encrypted disk was not installed using `sysinstall`, a few things such as setting the timezone, the keyboard map and the *root password* have not yet been taken care of. These settings can easily be changed by calling `sysinstall` now. Packages such as the X server, which is not part of the system, can be added using `pkg_add(8)`. The system is now fully functional and ready for use.

3.5 Complete hard disk encryption using GELI

This chapter describes the process of setting up complete hard disk encryption using FreeBSD's new GELI facility. GELI is so far only available on the 6.x branch. It is important to note that the memory disk approach as discussed previously with GBDE is also possible with GELI. But since GELI makes it possible to mount an encrypted partition as the root filesystem, the memory disk is not a requirement anymore. This advantage, however, is somewhat weakened by a drawback that the memory disk scheme does not suffer from. This particular issue will be discussed in more detail later and ultimately it is up to the user to decide which scheme is more appropriate.

As the concept of having a memory disk with GELI is very similar to having one with GBDE, this chapter discusses only how to use GELI to boot *directly* with an encrypted root filesystem – *without* the need for a memory disk.

Many of the steps required to make complete hard disk encryption work with GBDE are also necessary with GELI – regardless of whether a memory disk is used or not. Therefore the description and explanation of some steps will be shortened or omitted completely here. The necessary commands will of course be given, but for a more detailed explanation the respective chapters in the GBDE part are recommended for reference.

3.5.1 Readying the hard disk

As it has already been mentioned in the GBDE chapter, erasure of previously stored data on the medium intended for encryption is strongly recommended. The data can be overwritten by either using the zero or the entropy device as a source.

```
# dd if=/dev/zero of=/dev/ad0 bs=1m
```

-- or --

```
# dd if=/dev/random of=/dev/ad0 bs=1m
```

Their respective advantages and drawbacks were discussed in chapter 3.4.1.

3.5.2 Improvements and new problems with GELI

Just as GBDE, GELI must first initialize the medium intended for encryption. GELI's big advantage over GBDE for the purpose of complete hard disk encryption is that it enables the kernel to mount an encrypted partition as the root filesystem. This works by passing the -b parameter to the `geli(8)` userland tool when the hard disk is initialized. This parameter causes GELI to flag the partition as “ask for passphrase upon discovery”.

When the kernel initializes the various storage media in the system at boot time, it searches the partitions on them for any that have been flagged by the user and then asks for the passphrase of the respective partition. The most important fact is, that this is done in kernel space – the new device node providing access to the plain text (with the suffix .eli, analogous to GBDE's .bde suffix) therefore already exists *before* the kernel mounts the root filesystem.

Furthermore – as it is possible with GBDE – GELI also allows the key material to be retrieved from additional sources besides the passphrase. While GBDE uses the 16-byte *lockfile* for this purpose, GELI supports the specification of a *keyfile* with the -K parameter. The size of this keyfile is not hardcoded into GELI and can be chosen freely by the user; if '–' instead of a file name is given, GELI will read the contents of the keyfile from the standard input.

This way it is even possible to concatenate several files and feed them to GELI's standard input through a pipeline. The individual files would then each hold a part of the key and the key would therefore be distributed across several (physical, if chosen) places.

Unfortunately, however, the keyfile *cannot* be used with partitions which have been flagged for “ask for passphrase upon discovery”. Using a passphrase and a keyfile to grant access to the encrypted data would require that a parameter be passed to the kernel – specifying the path to the keyfile. This path could of course also be hardcoded into the kernel, for example that the keyfile must be located at **/boot/geli.keys/<device>**.

Unfortunately, this functionality does not yet exist in GELI. The ability to mount an encrypted partition as the root filesystem comes therefore at the price of having to rely only on the passphrase to protect the data. The memory disk approach that was discussed in order to make complete hard disk encryption work with GBDE also works with GELI. Although it is harder to set up and maintain, it combines the advantages of “something you know” and “something you have”, namely a passphrase *and* a lockfile/keyfile. Especially on mobile devices it is risky to rely only on a passphrase, since it will face intensive exposure as it must be typed in each time the system is booted up.

The choice between better usability and increased security is therefore left to user.

3.5.3 Initializing, attaching and partitioning

Initializing the hard disk with GELI works similarly as it does with GBDE – except that the partition must be flagged as “ask for passphrase upon discovery” and therefore cannot (yet) use a keyfile.

```
# geli init -b /dev/ad0
Enter new passphrase:
Reenter new passphrase:
```

Very important here is to specify the -b parameter, which causes the **geom_eli.ko** kernel module to ask for the passphrase if a GELI encrypted partition has been found. The -a parameter can (optionally) be used to specify the encryption algorithm: AES, Blowfish or 3DES.

If this set-up is performed directly from the 'fix-it' live filesystem, then the **/lib** directory must be created by symlinking it to the existing **/dist/lib** directory. This is necessary because GELI needs to find its libraries in **/lib**. The GELI executable will actually run without **/lib**, but will then *hide* its features from the user – therefore making the problem much less obvious.

Attaching the hard disk is also largely the same as with GBDE, again except that the keyfile parameter must be omitted from the command.

```
# geli attach /dev/ad0
Enter passphrase:
```

Upon successful attachment, a new device node will be created in the **/dev** directory which carries the name of the specified device plus a '.eli' suffix. Just like the '.bde' device node created by GBDE, this node provides access to the plain text. The output of `geli` after successful attachment looks something like this (details depend on the parameters used and the available hardware):

```
GEOM_ELI: Device ad0.eli created.
GEOM_ELI:     Cipher: AES
GEOM_ELI: Key length: 128
GEOM_ELI:     Crypto: software
```

Since `sysinstall` cannot read GELI encrypted partitions either, the partitioning must be done using the `bsdlabel` tool.

```
# bsdlabel -w /dev/ad0.eli
# bsdlabel -e /dev/ad0.eli
```

Partition management was discussed in more detail in chapter 3.4.4.

3.5.4 Filesystem creation and system installation

Now that the partition layout has been set, the filesystem(s) can be created, so FreeBSD can be installed.

```
# newfs /dev/ad0.elia
# newfs /dev/ad0.elid
```

etc.

The actual installation of the system on the encrypted hard disk must also be done manually, since `sysinstall` does not support GELI encrypted partitions.

```
# mount /dev/ad0.elia /fixed
# export DESTDIR=/fixed/
# cd /dist/6.0-RELEASE/base && ./install.sh
You are about to extract the base distribution into /fixed - are you SURE
you want to do this over your installed system (y/n)?
```

3.5.5 The removable medium

Since this medium is not going to be encrypted, it can be sliced and partitioned with `sysinstall`. The size requirements are largely the same as for GBDE – the minimum is even a bit lower because there is no need to store the image of the memory disk. With a customized kernel, this minimum may be as low as 4 MB.

In order to boot the kernel from the removable medium (**/removable**), it is necessary to copy the **/boot** directory from the encrypted hard disk (mounted on **/fixed**).

```
# cp -Rpv /fixed/boot /removable
```

All kernel modules except **geom_eli.ko** and its dependency **crypto.ko** (and **acpi.ko**, if used) can be deleted if space is a problem. Further, all modules and even the kernel can be gzipped. This saves not only space, but also reduces loading time.

```
# cd /removable/boot/kernel
# gzip kernel geom_eli.ko acpi.ko
```

Just as it is the case with GBDE, GELI also needs its kernel module **geom_eli.ko** loaded by `loader(8)` in order to ask for the passphrase before the root filesystem is mounted. The following command adds the appropriate entry to **/boot/loader.conf**.

```
# echo geom_eli_load=\“YES\“>> /removable/boot/loader.conf
```

3.5.6 Mounting the encrypted partition

Because of GELI's ability to mount encrypted partitions as the root filesystem the *entire* workaround with the memory disk can be avoided. So far, however, the kernel does not know which partition it must mount as the root filesystem – even if the device node to the plain text of the encrypted hard disk has been created by GELI. The memory disk approach, which is necessary to make complete hard disk encryption work with GBDE, has the advantage that the kernel will automatically mount the memory disk as the root filesystem if an image has been preloaded.

In this case, however, it is necessary to create an entry in **/etc/fstab**, so the kernel knows which partition to mount as the root filesystem.

```
# mkdir /removable/etc
# echo "/dev/ad0.elia   /     ufs    rw   1     1" >> /removable/etc/fstab
```

It is important to note that this file must be stored on the *removable* medium and serves only the purpose of specifying the device for the root filesystem. As soon as the kernel has read out the contents of the file, it will mount the specified device as the root filesystem and the files on the removable medium (including **fstab**) will be *outside* of the filesystem name space. This means that the removable medium must first be mounted before the files on it can be accessed through the filesystem name space. It also means, however, that the removable medium can actually be *removed* after the root filesystem has been mounted from the encrypted hard disk – thus reducing unnecessary exposure. It is crucial that the removable medium be always in the possession of the user, because the whole concept of complete hard disk encryption relies on the assumption that the boot medium – therefore the *removable* medium, not the hard disk – is uncompromised and its contents are trusted.

If any other partitions need to be mounted in order to boot up the system – for example **/dev/ad0.elid** for **/usr** – they must be specified in **/etc/fstab** as well. Since most installations use at least one swap partition, the command for adding the appropriate entry to **/etc/fstab** is given below.

```
# echo "/dev/ad0.elib   none   swap   sw   0   0" > /fixed/etc/fstab
```

The system is now ready for use and can be booted from the removable medium. As the different storage devices in the system are found, GELI searches them for any partitions that were initialized with the `geli init -b` parameter and asks for the passphrase. If the correct one has been provided, GELI will create new device nodes for plain text access to the hard disk and the partitions on it (e.g. **/dev/ad0.elia**), which then can then be mounted as specified in **/etc/fstab**.

After that, the rest of the system is loaded. `sysinstall` can then be used in order to adjust the various settings that could not be set during the installation procedure – such as timezone, keyboard map and especially the root password!

4 Complete hard disk encryption in context

4.1 New defenses & new attack vectors – again

Any user seriously thinking about using complete hard disk encryption should be aware of what it actually protects and what it does not.

Since encryption requires a lot of processing power and can therefore have a noticeable impact on performance, it is usually not enabled by default. FreeBSD marks no exception here. Although it provides strong encryption algorithms and two powerful tools for encrypting storage media, it is up to the user to discover and apply this functionality.

This paper gave instructions on how to encrypt an entire hard disk while most of the operating system is still stored and loaded from it. It is important to remember, however, that FreeBSD – or any other software component for that matter – will not warn the user if the encrypted data on the hard disk is leaked (see chapter 2.3) or intentionally copied to another, unencrypted medium, such as an external drive or a smart media card. It is the responsibility of the user to encrypt these media as well.

This responsibility applies equally well to data *in transit*. Network transmissions are in most cases not encrypted by default either. Since all encryption and decryption of the data on the hard disk is done transparently to the user once the passphrase has been provided, it is easy to forget that some directories might contain data which is stored on a different machine and made available through NFS, for example – in which case the data is transferred *in the clear* over the network, unless *explicitly* set up otherwise.

The mounting facility in UNIX is very powerful; but it also makes it difficult to keep track of which medium actually holds what data.

The network poses of course an additional threat, because of an attacker's ability to target the machine remotely. The problem has already been discussed in chapter 1. If a particular machine is easier to attack remotely than locally, any reasonable attacker will not even bother with getting physical access to the machine. In that case it would make no sense to use complete hard disk encryption, because it does not eliminate the weakest link (the network connectivity).

If, on the other hand, not the network, but the unencrypted or not fully encrypted hard disk is the weakest link and the attacker is also capable of getting physical access to the machine (for reasons discussed in chapter 2.4), then complete hard disk encryption makes sense.

A key point to remember is that as long as a particular storage area is attached, the data residing on it is not protected any more than any other data accessible to the system. This applies to both GBDE and GELI; even unmounting an encrypted storage area will not protect the data from compromise since the corresponding device node providing access to the plain text still exists. In order to remove this plain text device node, the storage area in question must be *detached*. With GBDE this must be done manually, GELI has a feature that allows for automatic detachment on the last close – but this option must be *explicitly* specified.

Since the partition holding the operating system must always be attached and mounted, its contents are also vulnerable during the entire time the system is up. This means that remotely or even locally introduced viruses, worms and trojans can compromise the system in the same way they can do it on a system *without* complete hard disk encryption.

Another way to attack the system would be by compromising the hardware itself, for example by installing a hardware keylogger. This kind of attack is very hard to defend against and this paper makes no attempt to solve this issue.

What complete hard disk encryption *does* protect against, is attacks which aim at either accessing data by reading out the contents of the hard disk on a different system in order to defeat the defenses on the original system or by compromising the system stored on the hard disk, so the encryption key or the data itself can be leaked. Encryption does *not*, however, prevent the data from being destroyed, both accidentally and intentionally.

If it is chosen that the encrypted partition is mounted directly as the root filesystem – without the need for a memory disk, then it is crucial that a strong passphrase be chosen, because that will be the only thing required to access the encrypted data. Choosing the memory disk approach makes for a more resilient security model, since it enables the user to use a lockfile (GBDE) or a keyfile (GELI) – in order to get access to the data.

While all these previously mentioned conditions and precautions matter, it is absolutely crucial to understand that the concept of complete hard disk encryption depends upon the assumption that the data on the removable medium is *trusted*.

The removable medium must be used because the majority of the hardware is not capable of booting encrypted code. Since the kernel and all the other code necessary for mounting the encrypted partition(s) must be stored in the clear on the removable medium, the problem of critical code getting compromised has, in fact, not really been solved. The most efficient way to attack a system like this would most likely be by compromising the code on the removable medium.

It is therefore crucial that the user keep the removable medium with him or her at all times. If there is the slightest reason to believe that the data on it may have been compromised, its contents must be erased and reconstructed according to the instructions in the respective GBDE or GELI chapters.

If the removable medium has been lost or stolen *and* there was a keyfile or lockfile stored on it, then two issue must be taken into account:

- The user will not be able to access to encrypted data even with the passphrase. It is therefore strongly recommended that a backup of the keyfile/lockfile be made and kept in a secure place – preferably without network connectivity.
- The second possibility can be equally devastating, since the keyfile/lockfile could fall into the hands of someone who is determined to break into the system. In that case, all the attacker needs is the passphrase – which can be very hard to keep secret for a mobile device. It is therefore recommended that both the passphrase *and* the keyfile/lockfile are changed in the event of a removable medium loss or theft.

4.2 Trade-offs

Complete hard disk encryption offers protection against specific attacks as discussed in chapter 4.1. This additional protection, however, comes at a cost – which is usually why security measures are not enabled by default. In the case of complete hard disk encryption, the trade-offs worth mentioning the most are the following:

- Performance. Encryption and decryption consume a lot of processing power. Since each I/O operation on the encrypted hard disk requires additional computation, the throughput is often limited by the power of the CPU(s) and not the bandwidth of the storage medium. Especially write operations, which must be encrypted, are noticeably slower than read operations, where decryption is performed. Systems which must frequently swap out data to secondary storage and therefore usually to the encrypted hard disk can suffer from an enormous performance penalty. In cases where performance becomes too big a problem it is suggested that dedicated hardware be used for cryptographic operations. GELI supports this by using the crypto(9) framework, GBDE unfortunately does so far not allow for dedicated hardware to be used and must therefore rely on the CPU(s) instead.
- Convenience. Each time the system is booted, the user is required to attach or insert the removable medium and enter the passphrase. Booting off a removable medium is usually slower than booting from a hard disk and the passphrase introduces an additional delay.
- Administrative work. Obviously the whole scheme must first be set up before it can be used. The majority of this quite lengthy process must also be repeated with each system upgrade as the code on the removable medium must not get out of sync with the code on the hard disk. As this set-up or upgrade process is also prone to errors such as typos, it may be considered an additional risk to the data stored on the device.

This list is by no means exhaustive and every user thinking about using complete hard disk encryption is strongly encouraged to carefully evaluate its benefits and drawbacks.

4.3 GBDE vs. GELI

FreeBSD provides two tools for encrypting partitions, GBDE and GELI. Both can be used to make complete hard disk encryption work. If GBDE is chosen, the memory disk approach *must* be used, as GBDE does not allow the kernel to mount an encrypted partition as the root filesystem. The advantage is that it is possible to use a lockfile in addition to a passphrase. This makes for a more robust security model and should compensate for the administrative “overhead” caused by the memory disk.

GELI not only makes it possible to use a memory disk too, it also allows the user to choose from different cryptographic algorithms and key lengths. In addition to that it also offers support for dedicated cryptographic hardware devices and of course eliminates the need for a memory disk by being able to directly mount the encrypted boot partition. The drawback of mounting the root directly from an encrypted partition is that GELI so far does not allow for a keyfile to be used and therefore the security of the encrypted data depends solely on the passphrase chosen.

Looking at the features of the two tools, it may seem as though GELI would be the better choice in *any* situation. It should be noted, however, that GBDE has been around for much longer than GELI and therefore is more likely to have received more testing and review.

5 Conclusion

Mobile devices are intended to be used anywhere and anytime. As these devices get increasingly sophisticated, they allow the users to store massive amounts of data – a lot of which may often be sensitive. Encrypting individual files simply does not scale and on top of that does nothing to prevent the data from leaking to other places. Partition-based encryption scales much better but still, a lot of information can be compiled from unencrypted sources such as system log files, temporary working copies of opened files or the swap partition. In addition to that, both schemes do nothing to protect the operating system or the applications from being compromised.

In order to defend against this kind of attack, it is necessary to encrypt the operating system and the applications as well and boot the core parts such as the kernel from a removable medium. Since the boot code must be stored unencrypted in order to be loaded, it must be kept on a medium that can easily be looked after.

FreeBSD provides two tools capable of encrypting disks: GBDE and GELI. Complete hard disk encryption can be accomplished by using either a memory disk as the root filesystem and then mount the encrypted hard disk in a subdirectory or by directly mounting the encrypted hard disk as the root filesystem.

The first approach can be done with both GBDE and GELI and has the advantage that a lockfile or keyfile can be used in addition to the passphrase, therefore providing more robust security. The second approach omits the memory disk and therefore saves some administrative work. It works only with GELI, however, and does not allow for a keyfile to be used – therefore requiring a trade-off between better usability/maintain-ability and security.

Under no circumstances does complete hard disk encryption solve all problems related to security or protect against any kind of attack. What it *does* protect against, is attacks which are aimed at accessing data by reading out the contents of the particular hard disk on a different system in order to defeat the original defenses or to compromise the operating system or applications in order to leak the encryption key or the encrypted data itself.

As with any security measure, complete hard disk encryption requires the users to make trade-offs. The increase in security comes at the cost of decreased performance, less convenience and more administrative work.

Complete hard disk encryption makes sense if an unencrypted or partially encrypted hard disk is the weakest link to a particular kind of attack.

References & further reading

Dawidek, 2005a
P. J. Dawidek, *geli – control utility for cryptographic GEOM class*
FreeBSD manual page
April 11, 2005

Dawidek, 2005b
P. J. Dawidek, *GELI - disk encryption GEOM class committed*
http://lists.freebsd.org/pipermail/freebsd-current/2005-July/053449.html
posted on the 'freebsd-current' mailing list
July 28, 2005

Dowdeswell & Ioannidis, 2003
R. C. Dowdeswell & J. Ioannidis, *The CryptoGraphic Disk Driver*
http://www.usenix.org/events/usenix03/tech/freenix03/full_papers/dowdeswell/dowdeswell.pdf
June 2003

Kamp, 2003a
P.-H. Kamp, *GBDE – GEOM Based Disk Encryption*
http://phk.freebsd.dk/pubs/bsdcon-03.gbde.paper.pdf
July 7, 2003

Kamp, 2003b
P.-H. Kamp, *GEOM Tutorial*
http://phk.freebsd.dk/pubs/bsdcon-03.slides.geom-tutorial.pdf
August 19, 2003

Lemos, 2005
R. Lemos, *Backups tapes a backdoor for identity thieves*
http://www.securityfocus.com/news/11048
April 28, 2005

Leyden, 2004
J. Leyden, *Oops! Firm accidentally eBays customer database*
http://www.theregister.co.uk/2004/06/07/hdd_wipe_shortcomings/
June 7, 2004

Noguchi, 2005

Y. Noguchi, *Lost a BlackBerry? Data Could Open A Security Breach*
http://www.washingtonpost.com/wp-dyn/content/article/2005/07/24/AR2005072401135.html
July 25, 2005

OpenBSD, 1993

vnconfig - configure vnode disks for file swapping or pseudo file systems
OpenBSD manual page
http://www.openbsd.org/cgi-bin/man.cgi?query=vnconfig&sektion=8&arch=i386&apropos=0&manpath=OpenBSD+Current
July 8, 1993

Reuters, 2005

Reuters, *Stolen PCs contained Motorola staff records*
http://news.zdnet.co.uk/internet/security/0,39020375,39203514,00.htm
June 13, 2005

Sarche, 2005

J. Sarche, *Hackers hit U.S. Army computers,*
http://www.globetechnology.com/servlet/story/RTGAM.20050913.gtarmysep13/BNStory/Technology/
September 13, 2005