22nd Chaos Communication Congress

Complete Hard Disk Encryption with FreeBSD

Marc Schiesser
m.schiesser [at] quantentunnel.de

December 29th 2005
Berliner Congress Center, Germany

Part I:
Motivation

Why use in-storage encryption?

- „ ... Citigroup said computer tapes containing account data on 3.9 million customers, including Social Security numbers, were lost ...“ [Reuters, 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]
- „ ... a laptop computer containing the names and Social Security numbers of 16,500 current and former MCI Inc. employees was stolen ... “ [Noguchi, 2005]
- „Iron Mountain ... also lost tapes holding information on 600,000 current and former Time Warner workers.“ [Reuters, 2005]

Partial encryption

File-based encryption

- user can decide for each file individually whether and how it is to be encrypted
- examples of these tools include: PGP, GnuPG, Ncrypt as well as built-in operating system solutions

advantages:

- ✓ no “unnecessary“ encryption (saves CPU cycles)
- ✓ different keys for different files

drawbacks:

- ✗ meta data is not encrypted
- ✗ leakage risk

The “leakage risk“

- You encrypt a file or a whole partition, assuming that it protects your data from unauthorized access while the machine is turned *off*.
- Unfortunately, a lot of programs create temporary copies of the files you are working on.
- In the case of an application/OS crash, these copies usually remain unencrypted in the filesystem.
- Even *if* all temporary copies are deleted afterwards, the data still remains on the medium until physically overwritten.
- The swap partition represents an additional leakage risk.

The “leakage risk“ (cont'd)

Your encrypted data is *leaked* to unencrypted parts of the medium.

Implications

- file-based encryption becomes virtually useless
- partition-based encryption requires a lot of symlinking to mitigate the risk, but still does not eliminate it completely

Partial encryption

Partition-based encryption

advantages:

- ✔ encryption of meta data
- ✔ all data on the partition is encrypted *by default*
- ✔ dramatically reduces leakage risk

drawbacks:

- ✗ leakage risk still existent !!!

but...

Partial encryption

... an even bigger problem exists:

The encrypted data or the encryption key can be obtained by compromising the as yet *unencrypted* OS and applications – therefore rendering the encryption of the data itself irrelevant.

Partial encryption

How secure do you consider your data and program code to be from unauthorized access in:

- a hotel room
- your office
- checked-in baggage
- a locker at the airport, train station, etc.
- your parked car

?

Conclusion

Unless you keep your notebook in sight at all times, you cannot be sure it has not been compromised !!!

The solution ?

Encrypt the OS and applications too – along with everything else !

The next problem

Today's computers cannot boot encrypted code.

Conclusion

- The OS (at least the core parts) must be booted from an unencrypted medium.
- The hard disk is too risky to boot from.

The next solution

- USB memory sticks:
  - provide plenty of space
  - are affordable
  - are bootable
  - are light and small and thus:
  - can be carried around all the time

- alternative media (e.g. 8cm CD-R/W) work, but are less convenient
- write access is not a requirement for operation (only for the set-up)

Part II:
Implementation

Requirements

- a bootable, removable medium (preferably a USB memory stick)
  - minimum size: 5MB (in this case, kernel modules need to be loaded from the hard disk *after* init(8))
- FreeBSD 5.x or later installation disc
- basic FreeBSD 5.x or later system (fixed installation or LiveCD):
  - the FreeBSD installation disc (disc 1) also includes a 'fix-it' live filesystem
  - a graphical desktop environment, bootable from CD, is available at www.freesbie.org
- notebook/machine (capable of booting from a USB drive)

FreeBSD's device name space

- Devices nodes are located in /dev/ (by default) and created/removed automatically. Important devices:

ad*	ATA hard disks
da*	SCSI disks and mass storage devices attached via USB
acd*	ATAPI CD and DVD drives
afd*	ATAPI floppy drives
fd*	floppy disk drives
md*	memory disks
random	entropy device
null	null device
zero	zero device

- device indexing starts at 0, for more information see the appropriate man pages (e.g.: da(4), md(4))
- the kernel output during the booting process lists the recognized devices, see /var/run/dmesg.boot

FreeBSD's device name space (cont'd)

- What in PC terminology is called a partition, is called a slice in BSD derivatives !
- Slice numbering starts at 1 (not 0 !), an 's' is put between the device and the slice number:
  - 'ad0s1' is the first slice on the first ATA disk
  - 'da1s3' is the third slice on the second SCSI disk
- BSD derivatives further allow for partitioning of the slices themselves – these are called partitions and indicated by a letter after the slice number:
  - 'ad0s1a' is the boot partition of the first slice on the first ATA disk
  - 'da1s3b' is the swap partition of the third slice on the second SCSI disk

The usual disclaimer

Backup all of your data *before* proceeding, because the contents of your hard disk(s) will be erased !!

(another) important disclaimer

All following slides will assume that the hard disk to be encrypted is 'ad0' and the removable (USB) medium 'da0'.

You are responsible for adjusting these paths to your set-up !!

Cleaning the hard disk(s)

- Storage encryption is no good if previously stored, unencrypted data is still recoverable because it has not been physically overwritten !
- Therefore, all media that are to be encrypted, must be cleaned:

  # dd if=/dev/zero of=/dev/ad0 bs=1m (fastest)

  # dd if=/dev/random of=/dev/ad0 bs=1m (most secure)

- Adjust 'ad0' to your actual set-up and double-check before hitting 'Enter' !!!!

The tools provided by FreeBSD

	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

Solution 1:

unencrypted root filesystem

[GBDE, GELI]

Introducing GBDE

- GEOM Based Disk Encryption
- available in FreeBSD 5.x and later (5.x was declared -STABLE with 5.3-RELEASE)
- 128-bit AES encryption
- filesystem independent
- passphrase changeable without the need for reformatting

- for more information, see geom(4) and gbde(4)

GBDE: Initialization

- Before the encrypted disk can be used, it must be initialized:

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

- The lockfile is needed later, make sure you specify a path where it is absolutely safe !!!

The lockfile

- contains 16 bytes of data that is required to access the master-key and thus to correctly encrypt and decrypt the data
- use of the lockfile is not mandatory, GBDE can also store its contents in the first sector of the medium/partition
- in the previous case, however, “only“ the passphrase would be required to get access to the decrypted data
- since another medium (the removable boot medium) is needed anyway, we can increase security a lot more by separating the lockfile from the encrypted data !

passphrase + lockfile:

two-factor authentication

The lockfile (cont'd)

- in order to access the plain text data:
  - the encrypted data itself,
  - the passphrase
  - *and* the lockfile

  are required !
- if the lockfile is unavailable (lost or destroyed), even knowledge of the passphrase will not yield access to the decrypted data
- for a detailed description of GBDE's design see:
  - gbde(4)
  - Poul-Henning Kamp's paper “GEOM Based Disk Encryption“ [Kamp, 2003]

Accessing the decrypted data

- after the disk/partition has been initialized by GBDE, it is ready for use
- in order to gain access to the *decrypted* data, the corresponding device node must be *attached*:

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

- a new device node with the suffix '.bde' will be created:
  - /dev/ad0 provides access to the *cipher text*
  - /dev/ad0.bde provides access to the *plain text*
- after attaching, the plain text device node (.bde) can be partitioned, formated, mounted, etc. just like the original 'ad0' node

GBDE attaching and detaching

WARNING:

- after a particular storage area has been attached, it remains this way until it is explicitly *detached* or the system is shut down
- even unmounting an encrypted partition will not protect the data, as it will still be accessible through the corresponding '.bde' device node

as long as a storage area is attached, it is exposed to the same kind of threats *as any other data accessible through the system*

Partitioning the encrypted device

- usually, slicing and partitioning would be done using sysinstall
- since sysinstall does not support GBDE or GELI device nodes, the fdisk and bsdlabel tools must be used
- assuming that only one OS (FreeBSD) will be used on the machine, we can skip slicing (using fdisk) and start partitioning with bsdlabel(8)
- first write a standard label to the disk and then edit it:

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

Partitioning the encrypted device (cont'd)

- an example layout (100 MB disk):

```
# /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
```

- column 1: a=boot partition; b=swap; c=whole disk; d, e, f, g, h=at your disposal
  columns 2,3: partition size and its offset in sectors
  column 4: 4.2BSD, swap or unused
  columns 5,6,7: optional parameters, no changes required

- for details see bsdlabel(8)

Partitioning the encrypted device (cont'd)

- a modified layout:

```
# /dev/ad0.bde:
8 partitions:
#        size     offset    fstype   [fsize bsize bps/cpg]
a:   100000         16    4.2BSD           0     0
b:    48544     100016      swap           0     0
c:   198560          0    unused           0     0       # "raw" part, don't edit
d:    50000     148560    4.2BSD           0     0
```

- encrypted 50000 KB boot partition /dev/ad0.bdea
- encrypted ~24 MB swap partition /dev/ad0.bdeb
- encrypted 25000 KB additional partition /dev/ad0.bded (mountable anywhere, e.g. home directory)

FreeBSD's device name space revisited

- because the partitions a, b and d reside *inside* the encrypted disk, the partition name is put after the .bde suffix !
  - /dev/ad0.bdea
  - /dev/ad0.bdeb
  - etc.
- if the disk had first been partitioned, the partitions would have to be encrypted individually afterwards – resulting in different device node names:
  - /dev/ad0a.bde
  - /dev/ad0b.bde
  - etc.

Scenario: multiple operating systems

- if multiple operating systems are to be hosted on the same hard disk, slicing (PC terminology: partitioning) is required
- in that case, it is still possible to encrypt the whole *slice* on which FreeBSD resides – providing protection to both FreeBSD and the data on the same slice
- example of FreeBSD on slice /dev/ad0s1:
  - first use fdisk(8), in order to create the slices you need: ad0s1, ad0s2, etc.
  - then use gbde(8) to encrypt the FreeBSD slice: /dev/ad0s1.bde
  - use bsdlabel(8) to partition the encrypted FreeBSD slice: /dev/ad0s1.bdea, /dev/ad0s1.bdeb, etc.

Creating the filesystem

- now that device nodes for the encrypted partitions exist, we can create filesystems on them:

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

- the swap partition ('b') does not need a filesystem
- do not format the 'c' partition – better yet, ignore it altogether
- for details consult newfs(8)

Installing FreeBSD

- after the encrypted filesystems are ready, we should be able to install FreeBSD using its standard installation tool sysinstall
- unfortunately, sysinstall cannot deal with GBDE or GELI encrypted partitions

- what this means:
  - manual extraction of the various distribution archives
  - post-installation adjustments

Installing FreeBSD (sort of)

- mount the FreeBSD installation medium (if you are using the 'fix-it' live filesystem, find out where it is already mounted): assumption here /dist
- mount the encrypted boot partition ('a'): assumption here /fixed
- /dist/5.4-RELEASE (adjust to your version) contains one directory for each distribution – installing base is mandatory
- in order to install the base system, type:

```
# export DESTDIR=/fixed/
# cd /dist/5.4-RELEASE/base && ./install.sh
```

- repeat step two for each distribution you want to install (replace base with the appropriate name)

Now what?

- we now have a complete FreeBSD installation on the encrypted boot partition
- the encrypted swap partition is also set up
- since everything is encrypted, the machine cannot boot from the hard disk

- we now have to set up the removable boot medium

Slicing the boot medium

- since the data on the removable boot medium must not be encrypted, we can use the standard installation tool sysinstall for slicing and partitioning
- start sysinstall and then choose 'Configure' – 'Fdisk'
- the boot medium used here will be /dev/da0 – make sure you use the correct disk for slicing
- create at least one slice > 8 MB (recommended: > 25 MB), do not forget to make it bootable

Partitioning the boot medium

- since the swap partition on the encrypted hard disk will be used, the removable medium just needs a boot partition
- specify /removable as the mount point (the mount point actually does not matter, but it keeps sysinstall quiet)
- sysinstall creates the partition and the filesystem on it and also mounts it on the just specified location: /removable

Copying the required files

- the boot medium should now be mounted on /removable
- the encrypted boot partition should be mounted on /fixed
- required for booting is the /boot directory – copy it onto the boot medium:

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

The kernel modules

- kernel modules are loaded by loader(8) according to the entries in /boot/loader.conf – then control is passed over to the kernel
- manual (un)loading of modules is possible through the userland tools kldload(8) and kldunload(8)
- the kernel module geom_bde.ko must be loaded in advance:

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

- if any additional/third party modules are needed at boot time, they must be copied to /boot/kernel/ and appropriate entries added to /boot/loader.conf (defaults can be found in /boot/defaults/loader.conf)

The kernel modules (cont'd)

- the modules which are not needed at boot time, can be deleted from the removable medium
- in order to save space on the boot medium (/removable) and even more importantly – to speed up loading time, it is beneficial to gzip the kernel and the modules:

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

- since there is also a /boot directory on the encrypted (fixed) disk, we can load modules even though the original boot medium is not attached anymore !!
- make sure you keep the code on the removable medium in sync with the fixed medium (do NOT mix different versions/ releases/branches)

The problem with GBDE

- before an encrypted partition can be mounted it must first be attached through the *userland* utility gbde(8)
- a userland process such as gbde(8) or mount(8) must be forked from a parent process
- the first process created by the kernel is init(8), which in turn calls the rc(8) script
- the kernel, however, mounts the root filesystem *before* it calls init(8)
- with GBDE, the kernel must mount an unencrypted root filesystem, which contains the tools necessary for mounting the GBDE encrypted disk

The problem with GBDE

Solution

- create an *unencrypted* memory disk which can be mounted as the root filesystem – thus providing r/w access even from a read-only boot medium !
- the memory disk contains the executables which are required to get access to the plain text data on the encrypted partition (init(8), rc(8), gbde(8), mount(8))
- everything else is accessed through symlinked directories pointing from the memory disk to the encrypted hard disk

Creating the memory image

- first we create a regular file holding the memory image data, 10 MB should be enough for the root filesystem as most directories will only be symlinks:

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

- now we must create a device node for the image file:

```
# mdconfig -a -t vnode -f /removable/boot/mfsroot
md1
```

- create a filesystem on it and mount it (/memdisk):

```
# newfs /dev/md1
# mount /dev/md1 /memdisk
```

Populating the memory disk

- choose a directory which serves as the mount point for the encrypted boot partition (e.g. /safe):

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

- some directories act as mount points and do not need to be symlinked; etc is needed to hold rc(8) and the lockfile:

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

- copy rc(8) and the lockfile onto the memory disk:

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

- important: the GBDE lockfile is modified each time the passphrase is changed!

Populating the memory disk (cont'd)

- after the kernel has booted up, it invokes init(8), which in turn starts rc(8) – the startup script
- since we also need gbde(8) and mount(8) to get access to the plain text data, we can use the statically linked binaries in /rescue and save space on dependencies (libraries):

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

Creating the symlinks

- create the missing directory entries in the root (/) by sym-linking them to their actual location on the encrypted hard disk (/safe):

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

Gluing it all together

- insert the following commands after the line 'export HOME PATH' (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
```

- when rc(8) is called by init(8), the added commands take care of attaching the encrypted boot partition and then mount it
- after that, we wipe out rc(8) and the lockfile (a copy!) in /etc and symlink it so that it points to the complete directory on the encrypted disk – thus providing everything else for a successful boot
- after these added commands, the normal rc(8) operation will resume – completing the whole booting process

Integrating the memory disk

- as the memory disk image now contains all the necessary data, we can unmount and detach it:

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

- in order to save space and to speed up the booting process, we can compress the memory image:

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

- add the following lines to boot/loader.conf on the boot medium (/removable):

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

The memory disk: how it works

- the file containing the memory disk image must be loaded by loader(8) similar to a kernel module; it must be specified in /boot/loader.conf
- if the kernel has been compiled with the MD_ROOT option (already included in GENERIC), it will mount the image that was loaded first as the root filesystem
- this way, it is not even necessary to create /etc/fstab in order to mount the root filesystem
- more information can be found in md(4)

The swap partition

- although the swap partition itself does already exist, the system does not yet know which device to use

- in order to specify the encrypted swap partition to the system, an entry must be added to /etc/fstab on *the encrypted hard disk*:

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

Post-installation issues

- the encrypted disk, the boot medium and the memory disk are now ready for use and we can boot the system:

  ```
  # reboot
  ```

- since we did not use sysinstall in order to install the system, we have to adjust the important settings (timezone, keyboard map, root password, etc) afterwards:

  ```
  # sysinstall
  ```

- the system is now fully functional and you can start customizing, installing packages (using pkg_add(1)), etc.

Solution 2:

encrypted root filesystem

[GELI only]

Introducing GELI

- first released in FreeBSD 6.0
- several ciphers: AES, Blowfish, 3DES
- variable key length
- filesystem independent
- passphrase changeable without the need for reformatting
- supports dedicated hardware for accelerating cryptographic operations

- *allows kernel to mount encrypted root filesystem*

Getting started with GELI

- erase previously stored data as already discussed
- initialize the disk with GELI:

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

- the -b parameter instructs the GELI kernel module to ask for the passphrase if an encrypted partition has been found
- asking for the passphrase is done in *kernel space*, so the plain text device node will be available *before* the kernel mounts the root filesystem

New problems with GELI

- although GELI supports the use of a keyfile analogous to GBDE's lockfile, it *cannot* be used for partitions which have their passphrase prompted for in kernel space (-b parameter)
- the user must therefore rely on only a passphrase (one-factor authentication)
- unfortunately, the passphrase can be hard to keep secret when used on a mobile device
- *trade-off between better usability/maintainability and increased security*

Attaching & partitioning

- attaching works the same way it does with GBDE, except that we cannot use a keyfile here:

```
# geli attach /dev/ad0
Enter passphrase:
GEOM_ELI: Device ad0.eli created.
GEOM_ELI:      Cipher: AES
GEOM_ELI: Key length: 128
GEOM_ELI:      Crypto: software
```

- after the encrypted hard disk has been attached, it can be partitioned, again using bsdlabel instead of sysinstall:

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

Filesystem creation & system installation

- create filesystems on the partitions, where required:

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

- for the actual system installation, the GBDE principles apply:

```
# mount /dev/ad0.elia /fixed
# export DESTDIR=/fixed/
# cd /dist/6.0-RELEASE/base && ./install.sh
```

The removable medium

- slice and partition the removable medium according to the GBDE defaults
- in order to be able to boot from the removable medium, we need the /boot directory again:

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

- the modules geom_eli.ko and its dependency crypto.ko are required for operation and need to be loaded by loader(8), so that the code for attaching the encrypted hard disk is available *before* the root filesystem is mounted

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

The memory disk...

...is not needed anymore !!!

- GELI can ask for the passphrase to an encrypted partition in kernel space while the kernel is still booting
- GELI subsequently attaches the disk/partition and creates a device node for accessing the *decrypted* data (plain text)
- since the kernel does not know which device will be mounted as the root filesystem, the /etc/fstab file must be created on the *removable* medium:

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

- the swap partition as well as all other partitions must be specified in /etc/fstab *on the encrypted hard disk*

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

Post-installation issues

- since the system could not be installed with sysinstall, some settings need to be adjusted afterwards (e.g. timezone, keyboard map, root password)

```
# sysinstall
```

Part III:
Implications

The big picture: how it works

- you attach/insert the removable medium and boot the kernel and some modules from it – although this medium is unencrypted, we trust its contents because we carry it along all the time
- the removable medium is also set up to prompt for the decryption key and then mount the partition(s) on the fixed hard disk – which is encrypted, because we do not want to look after it all the time
- after the hard disk has been mounted, the rest of the OS is loaded from the encrypted disk – providing access to all encrypted applications and user data in a transparent way
- now we can (and should!) detach/remove the removable medium

Recommended way of deployment

- carry the removable medium on you at all times – otherwise you cannot guarantee its integrity and we are back where we started
- know the passphrase by heart and train yourself at entering it quickly and correctly – in case someone *is* watching you
- change the passphrase from time to time – especially if you feel like being watched
  - with GBDE, changing the passphrase will also change the contents of the lockfile !
- remove/detach the boot medium *as soon* as the system is up (if you boot from a CD, do not leave it in the drive)

For your own safety

- reminder: access to the plain text data requires:
  - passphrase,
  - lockfile/keyfile
  - and the encrypted data itself

- if you are afraid of forgetting your passphrase, write it down – but keep it in a secure place and apart from data and lockfile; use this only as a last resort
- the lockfile/keyfile can easily be deleted by accident or on purpose – either way, your data is lost unless you have a copy of the lockfile/keyfile (keep the copy in a secure place too)
- if you backup the data itself in plain text, be aware that this represents a potential risk

What complete disk encryption does *not* protect (from)

- data that is leaked onto *unencrypted* media, for example removable media or network shares (NFS) or even additional hard disks that have not been set up for encryption
- all data on *attached* partitions is vulnerable to *any* kind of remote attack or even local attack
- as the OS partition must always be mounted, its data does not receive *any additional* protection as long as the system is up
- *no* encrypted data is protected from destruction
- both GBDE and GELI cannot protect against attacks relying on compromising the hardware

What complete disk encryption *does* protect (from)

- detached or not yet attached storage areas do not have a corresponding *plain text* device node; the data therefore *is* protected against compromise – even if the system is up
- since the whole hard disk is encrypted, there is no risk that plain text data will leak onto the main permanent storage device
- swapped out data can no longer be trivially recovered from the hard disk
- data on the hard disk can no longer be accessed by circumventing OS or host machine defenses

Trade-offs

- since all security measures require some sort of trade-off, a lot of security features are turned off by default
- complete hard disk encryption makes no exception here; before deploying it, any user should therefore be aware of at least the following trade-offs:
  - performance: since all I/O from and to the disk require additional computation, the overall performance can decrease dramatically – especially on slower CPUs
  - administrative work: each installation must be very carefully set up and the removable medium must also be kept up-to-date (security patches, version upgrades, etc.)
  - convenience: each time the system is booted, the removable medium must be attached/inserted and the passphrase be typed in; booting from the removable medium usually causes an additional delay

Complete hard disk encryption

- complete hard disk encryption protects against these two specific threats:
  - reading out the contents of the disk on a different system in order to circumvent OS or host machine defenses
  - compromising program code in order to leak the encryption key or the encrypted data itself

Summing up

- determine whether data leakage and the compromise of program code represents an unacceptable risk according to your own situation and environment (determine the weakest link)
- be aware of the capabilities and limits of the hardware in question (bootable media, performance)
- understand the trade-offs inherent with using complete hard disk encryption, especially when deciding on the GBDE vs. GELI issue
- perform the implementation with great care and understand both what the individual commands actually do and how they work together to form the whole
- keep the data on both the hard disk *and* the removable medium up-to-date and in sync
- understand what complete hard disk encryption *does* protect and it does *not*

References

- [Kamp, 2003]
  P.-H. Kamp, *GBDE – GEOM Based Disk Encryption*
  http://phk.freebsd.dk/pubs/bsdcon-03.gbde.paper.pdf
  July 7, 2003
- [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
- [Reuters, 2005]
  Reuters, *Stolen PCs contained Motorola staff records*
  http://news.zdnet.co.uk/internet/security/0,39020375,39203514,00.htm
  June 13, 2005