Seven Ways to Increase Security in a New FreeBSD Installation

BY MARIUSZ ZABORSKI

FreeBSD can be used as a desktop or as a server operating system. When setting up new boxes, it is crucial to choose the most secure configuration. We have a lot of data on our computers, and it all needs to be protected. This article describes seven configuration improvements that everyone should remember when configuring a new FreeBSD box.

1 Full Disk Encryption

With a fresh installation, it is important to decide about hard drive encryption—particularly since it may be challenging to add later. The purpose of disk encryption is to protect data stored on the system. We store a lot of personal and business data on our computers, and most of the time, we do not want that to be read by unauthorized people. Our laptop may be stolen, and if so, it should be assumed that a thief might gain access to almost every document, photo, and all pages logged in to through our web browser. Some people may also try to tamper with or read our data when we are far away from our computer.

We recommend encrypting all systems—especially servers or personal computers. Encryption of hard disks is relatively inexpensive and the advantages are significant.

In FreeBSD, there are three main ways to encrypt storage:

- GBDE,
- ZFS native encryption,
- GELI.

Geom Based Disk Encryption (GBDE) is the oldest disk encryption mechanism in FreeBSD. But the GBDE is currently neglected by developers. GDBE supports only AES-CBC with a 128-bit key length and uses a separate key for each write, which may introduce some CPU overhead. It also has to store a per-sector key, which adds some disk space overheads. Since

other more efficient and flexible encryption alternatives appeared, GBDE has become less interesting to developers and users.

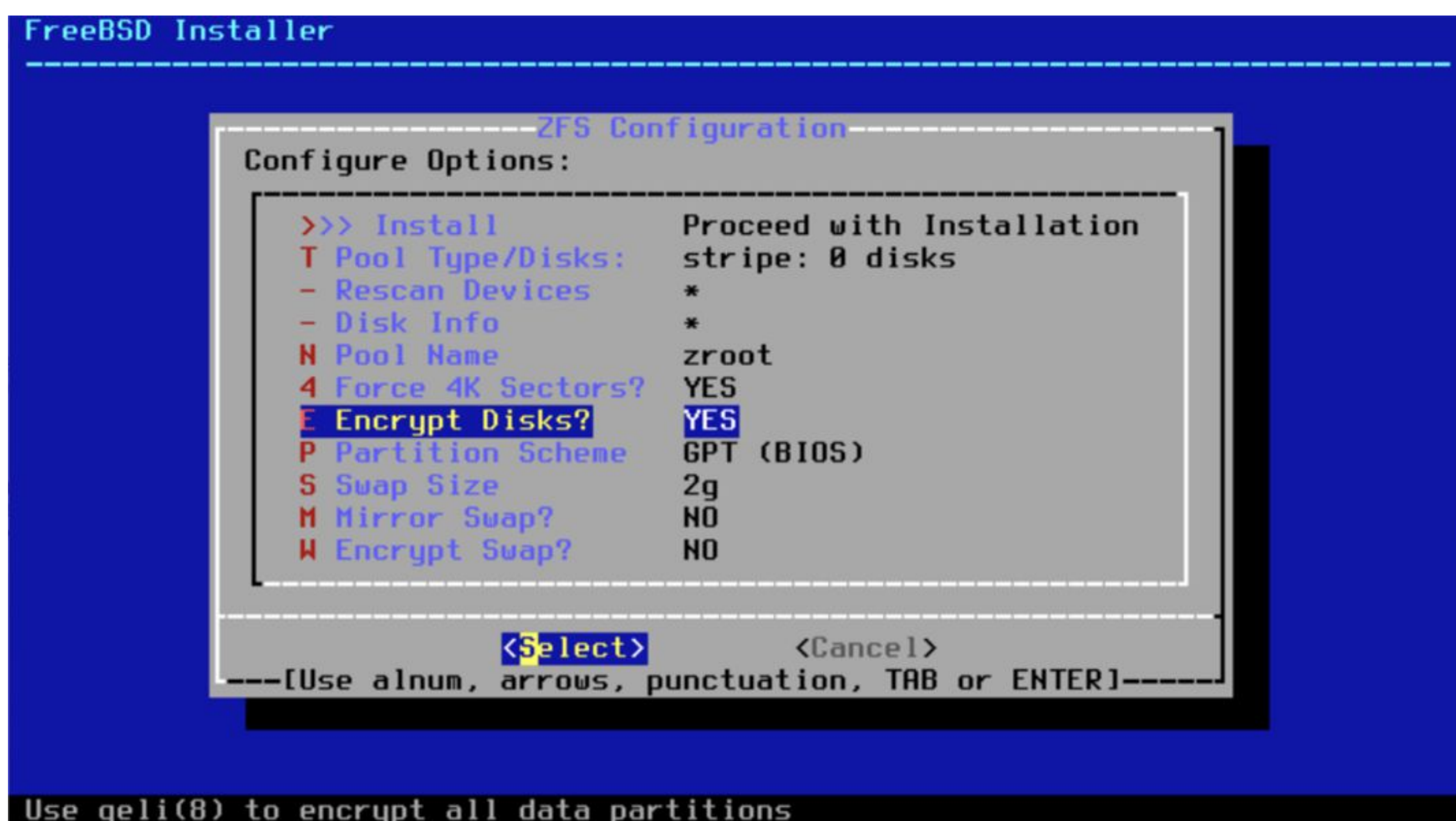
Introduced in version 13.0, OpenZFS native disk encryption is the newest disk encryption mechanism in FreeBSD. OpenZFS supports AES-CCM and AES-GCM with a 128-, 192-, or 256-bit key length. In OpenZFS encryption, not all data is encrypted. Datasets and snapshots, pool layouts or dataset properties are stored unencrypted. Often, those data may not be confidential, but it is something that users should be aware of. Another issue with OpenZFS disk encryption is that FreeBSD cannot boot from an encrypted dataset.

OpenZFS native disk encryption is a solid choice if users want to encrypt only a couple of datasets and protect only some parts of the system. The significant advantage of OpenZFS native disk encryption is that it can be enabled after installing the system. The only requirement is to use ZFS on the box and users can create new encrypted datasets.

GELI is our utility of choice. It supports full disk encryption, in which case only a FreeBSD-boot partition is not encrypted. And GELI supports many cryptographic algorithms. By default, it uses AES-XTS with a 256-bit key length.

If a user fears bootloader tampering on their critical boxes, they may want to reconfigure GELI. GELI can be set up to use a key file and a passphrase to encrypt a device. The bootloader can be moved and the key file from the hard disk to an external memory stick. No one can boot without this memory stick (because the key is only on it). Even if someone stole the memory stick and the box, they could not decrypt the device unless they knew the passphrase. Just remember to make a backup of the memory stick—if it is lost, no one will be able to recover the data.

The default GELI encryption may be easily configured from the FreeBSD installer:



GELI also supports one-time keys. When the box is rebooted, the encryption key will not be removed from the system. This means all data stored in the swap partition or temporary file systems will be removed.

Potentially, the operating system may swap out a critical application—for example, a browser—and store it on the hard drive, including user secrets stored in browser storage. If the swap is not encrypted, the attacker may read information from a disk.

The swap encryption may also be easily enabled from the FreeBSD installer, and so it is crucial to remember to address this. The option to do it is available on the same boot menu window as full disk encryption.

2 Use ZFS Because of Data Integrity

Another way of increasing the security of FreeBSD boxes is to use the best file system on the market—ZFS. This is also something that users have to think about during the installation process because changing a file system may be challenging. ZFS has dozens of exciting features, but in this section, we will focus on only one: ZFS data integrity from a security perspective.

ZFS uses Merkle trees. Merkle trees are data structures in which every leaf node has a cryptographic hash. This means ZFS is hierarchically checksumming data and all of its metadata.

If there is an underlying problem with hard drives, like corrupted sectors, misdirected write/read[1], phantom write[2] or something else that will corrupt our data, ZFS will discover this.

In file systems that do not support data integrity, the incorrect data may be returned to the application when such inconsistency occurs. With ZFS, the error will be returned instead of data. If we configure mirroring or RAIDZ, the file system will try to fix the issue.

ZFS supports many checksumming algorithms with SHA-256 as default. A user can decide whether they want to enable or disable integrity checks and it is highly recommended to keep it enabled. Thanks to data integrity, we can be sure that nothing was tampering with our data.

3 ZFS `setuid` and `exec` Properties

Besides data integrity, we can also add some additional protection with ZFS. We like to disable `setuid` on datasets that do not need them. The ZFS `setuid` property controls wherever the `setuid` bit is honored in file systems. The `setuid` binaries, when run, impersonate the owner of the binary. The `setuid` is very useful, and at the same time very dangerous, so we recommend keeping it limited and disabling this property on most datasets—except those containing systems `bin/sbin` directory.

Another attractive property is `exec`. This property allows us to disable whether programs in a dataset are allowed to be executed. We like to disable it for `/tmp` and `Download` directories. Most of the time, we move or download random files there, and we do not want to execute them by mistake.

4 Do Not Use Root

UNIX-like operating systems have a powerful account called `root`, which has absolute control over a system. It is obvious we should limit and audit access to this user. One thing that increases security is to set up a very complicated, unguessable password for the root account and not share it with anybody. Instead of logging or using `su(1)` for switching to a root account, we should use applications like `sudo(8)`.

`sudo(8)` is an application that allows you to control access to running commands as root or any other user in the system. System administrators can create a list of privileged commands that users can perform impersonating other users. In contrast to `su(1)`, `sudo(8)` does not require the `root` or other user password to operate. When a user wants to execute commands with elevated privileges, it asks them for their own password. This limits password sharing across system administrators. We can simply install `sudo(8)` using FreeBSD binary package management:

```
# pkg install sudo
```

Activities invoked by `sudo(8)` are logged. This adds some accountability in terms of who and what had been run. This is also why we should avoid spawning or logging as a different user.

Please remember not to overcomplicate the `sudo` configuration, keep it as simple as possible, and never give access to scripts/programs that users can modify without using `sudo(8)`. An overlooked sudo entry can be used as an easy-to-use vector attack.

The system should be configured with a complicated password for the root user and avoid allowing it to log in remotely (for example, through `ssh(1)`). When `sudo(8)` is up and running on some of the critical boxes, users may even consider locking the root user:

```
sudo pw lock root
```

Just be sure there is an option to boot in the single mode or an alternative recovery method in case root has to be unlocked.

Thanks to using `sudo(8)` on a daily basis—without a root shell—users can also omit issues when leaving an unlocked computer where someone could use the root shell to install malware. `sudo(8)` mitigates this issue by requesting a user password after some time of inactivity.

5 Backup

There are two kinds of people in the world—those who do backups and those who will do backups. Unfortunately, the second group will learn the importance of backups the hard way—when they lose the data. The stories with ransomware[3] and cloud providers[4] show us that many people still resist doing backups.[3] It is better to prevent than to cure—and the same goes with backup. Users understand the value of backups until it is too late. If you only learn one lesson from this article—please remember to perform backups.

Fortunately, we have used ZFS on our boxes to employ another interesting feature called snapshots. A snapshot is a copy of a dataset at a specific point in time. What is very useful about them is that we can extract those copies using the command zfs and import them with the command `zfs receive`. The easiest way to create a ZFS backup is by doing a snapshot and exporting it to the file.

```
# zfs snapshot -r name_of_the_pool@name_of_the_snapshot
# zfs send -R -c name_of_the_pool@name_of_the_snapshot > export_name
```

The commands above will create recursive snapshots of all datasets in the pool `name_of_the_pool`, and then export them to a file of `export_name`. The `-c` option means that the exported data will be compressed. When importing, just be sure that the system also supports all used compression algorithms. Now, the backup can be copied to an external disk, encrypted, sent to a cloud, any other server, or any other platform where users want to keep backups. There is also the possibility to do a simple remote backup using ssh:

```
# zfs send -R -c name_of_the_pool@name_of_the_snapshot | ssh example.com
cat > mybackupfile
```

Or, if the remote server supports ZFS, this can also be automatically imported on it:

```
# zfs send -R -c name_of_the_pool@name_of_the_snapshot | ssh example.com
zfs receive storage/mybackup
```

Thanks to this, the data will be available to browse on this server. Some additional configuration may be required to enable access to ZFS features. The user must be allowed to do the `zfs receive` command through `zfs allow` or by using `sudo(8)`. Sending whole backups may be a little too much overhead with respect to required storage and network bandwidth. Fortunately, ZFS also supports sending incremental snapshots. Using it, ZFS will create streams only with a difference between those two snapshots. The following command allows you to replicate a snapshot using less storage and network resources:

```
# zfs send -c -i name_of_previous_snapshot name_of_the_pool@name_of_the_
snapshot | ssh example.com zfs receive storage/mybackup
```

It is also possible to export these incremental portions to the files. However, if zfs send creates multiple small increment portions, they may be a little too complicated to recover—first the whole backup needs to be imported, and then each single file needs to be separately imported in the order of creation.

There are a couple of alternatives if the boxes are using different file systems. Users may choose tools like `dump(8)` for UFS, `rsync(1)`, or commercial products like Tarsnap.

6 Keep it Up to Date

Another lesson to remember is to upgrade our operating systems on a regular basis. This is one lesson that everybody talks about but somehow it is often forgotten.

FreeBSD security patches may be downloaded and installed using these commands:

```
# freebsd-update fetch
# freebsd-update install
```

For pkg updates and system updates:

```
# pkg update
# pkg upgrade
```

When using some production tools, it is also sensible to check if they do not have known vulnerabilities using:

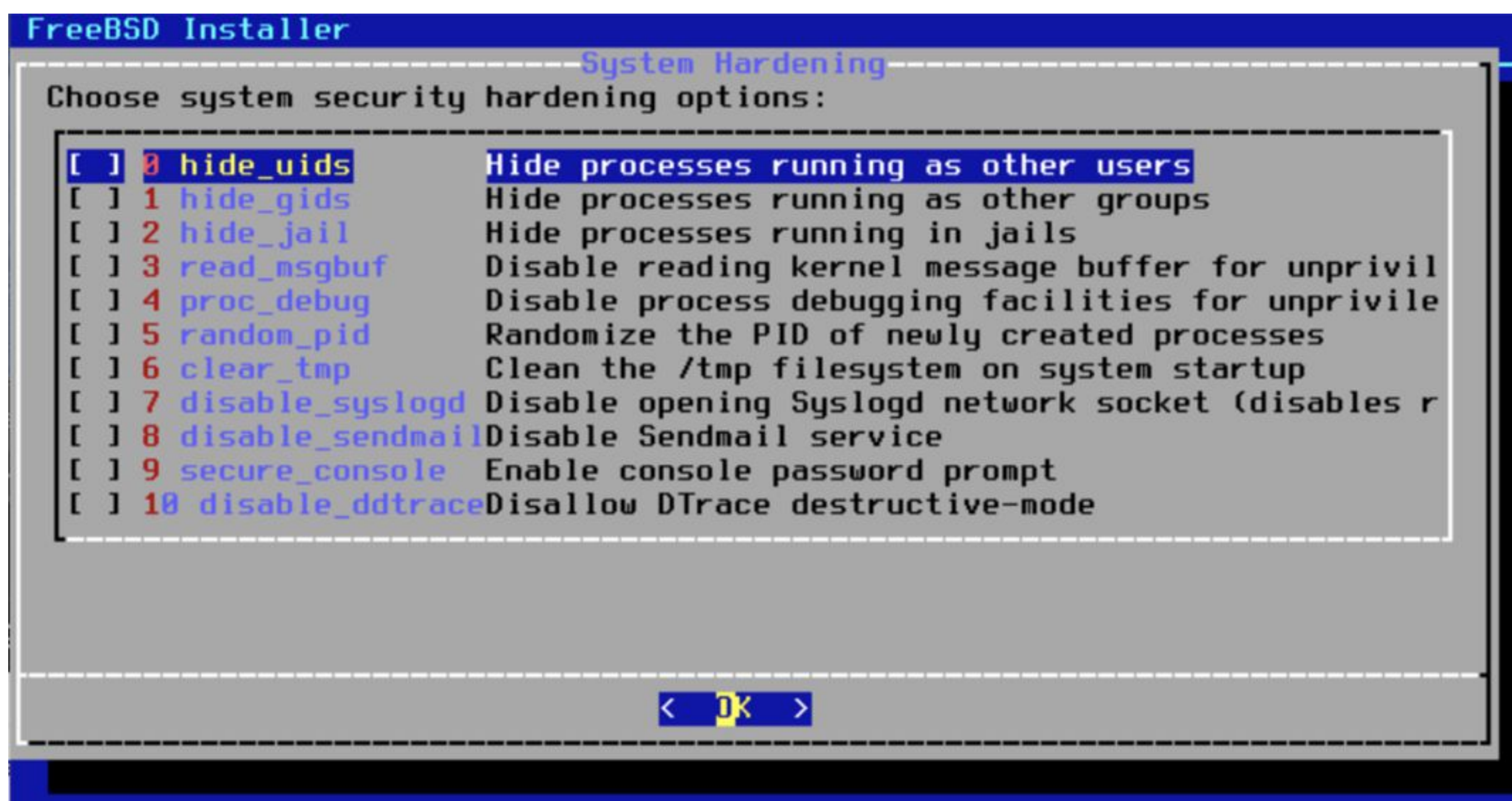
```
# pkg audit
```

As our file system of choice is ZFS, it is good to use a boot environment while doing upgrades. That allows us to easily roll back to a previous version of the system when something goes wrong.

7 Hardening

When installing a new FreeBSD instance, one of the last menus is about hardening the operating system. Some options make minor tweaks in FreeBSD and they slightly increase security. I'm thinking of things like hiding process UIDs and GIDs, clearing tmp on system startup, and randomizing the process identifiers.

By default, all the hardening options are disabled. We would recommend enabling them.



After the installation, we also recommend enabling ASLR (address space layout randomization), which changes the memory layout of the process each time they are run and makes applications harder to exploit. To enable it to run this on the FreeBSD box:

```
# sysctl kern.elf64.aslr.enable=1
# sysctl kern.elf32.aslr.enable=1
```

Users can also add those entries to /boot/loader.conf to enable them on the startup.

Summary

In this article, we described seven ways to improve the security of FreeBSD boxes. Using ZFS in the FreeBSD world is very popular but reducing the dataset's capabilities to run `setuid` and `exec` files is not common—and users can benefit from it. We hope full disk encryption, backups, and `sudo(8)` usage are already a standard, but it is always good to be reminded to use them. We also discussed minor tweaks in the FreeBSD configuration that randomize or hide some system information.

1. Misdirected write/read is a situation when the hard drive will write/read the data from a different place than the system requested. It may accrue because of the bit flip on CPI/cable or when the hard drive head will mass up.
2. Phantom write is a situation when the operating system thinks that some data was stored on the disk, but for some reason, the operation never made it to the disk.
3. https://www.zdnet.com/article/ransomware-this-is-the-first-thing-you-should-think-about-if-you-fall-victim-to-an-attack/
4. https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU

MARIUSZ ZABORSKI currently works as a security expert at 4Prime. He has been the proud owner of the FreeBSD commit bit since 2015. His main areas of interest are OS security and low-level programming. In the past, he worked at Fudo Security, where he led a team developing the most advanced PAM solution in IT infrastructure. In 2018, Mariusz organized the Polish BSD User Group. In his free time, he enjoys blogging at https://oshogbo.vexillium.org.