FreeBSD JOURNAL

May/June 2021

SECURITY

Seven Ways to Increase Security in a New FreeBSD Installation

The copyinout Framework

Using TLS to Improve NFS Security

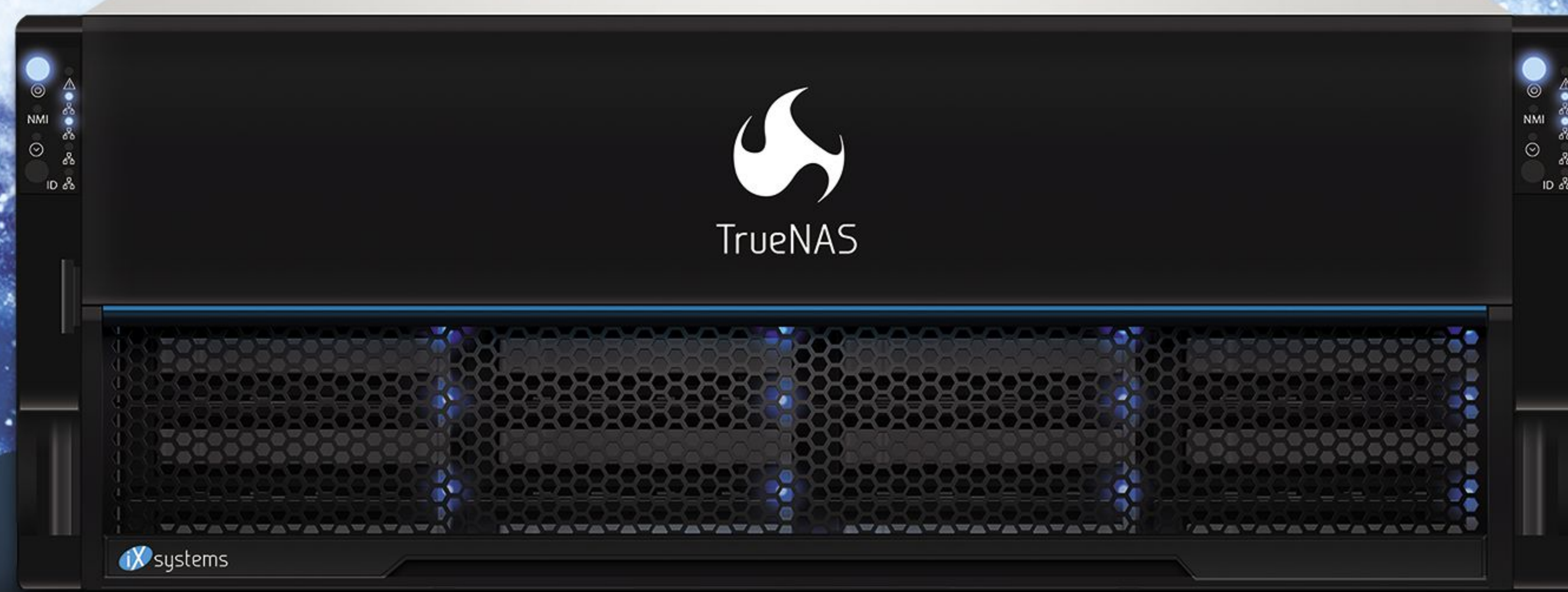
Plus:

Capsicum Case Study: Got

Practical Ports

TrueNAS® M-SERIES
Powerfully Scalable Enterprise Storage

OPEN STORAGE FOR ENTERPRISE WORKLOADS



UTILIZES FLASH-OPTIMIZED ZFS TECHNOLOGY

IDEAL FOR LATENCY-SENSITIVE AND BUSINESS-CRITICAL VIRTUAL MACHINES AND PHYSICAL WORKLOADS.

- PERFORMANCE AND SCALE WITHOUT COMPROMISE
- INTELLIGENT STORAGE OPTIMIZATION
- SELF-HEALING DATA PROTECTION
- UNLIMITED SNAPSHOTS AND REPLICATION

Contact iXsystems to Learn More about what TrueNAS® can do for your business!

iXsystems.com/TrueNAS | (855) GREP-4-iX

© 2020 iXsystems. TrueNAS is a registered trademark of iXsystems, Inc. All rights reserved. Intel, the Intel logo, Xeon, and Xeon Inside are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. All product names, logos, and brands are property of their respective owners.

FreeBSD® JOURNAL
Editorial Board

John Baldwin • FreeBSD Developer and Chair of FreeBSD Journal Editorial Board.

Justin Gibbs • Founder of the FreeBSD Foundation, President of the FreeBSD Foundation, and a Software Engineer at Facebook.

Daichi Goto • Director at BSD Consulting Inc. (Tokyo).

Tom Jones • FreeBSD Developer, Internet Engineer and Researcher at the University of Aberdeen.

Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics*.

Michael W Lucas • Author of more than 40 books including *Absolute FreeBSD*, the *FreeBSD Mastery* series, and *git commit murder*.

Ed Maste • Senior Director of Technology, FreeBSD Foundation and Member of the FreeBSD Core Team.

Kirk McKusick • Treasurer of the FreeBSD Foundation Board, and lead author of *The Design and Implementation* book series.

Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of Asia BSDCon, Member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology.

Benedict Reuschling • Vice President of the FreeBSD Foundation Board and a FreeBSD Documentation Committer.

Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge.

Mariusz Zaborski • FreeBSD Developer, Manager at Fudo Security.

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org
Copyright © 2021 by FreeBSD Foundation. All rights reserved. This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Foundation

Security

Information Security has a concept of "Defence in Depth," a strategy of applying multiple layers of security measures which work in concert to limit the impact of an attacker gaining unauthorized access to a system, for example, by exploiting a vulnerability. These layers include techniques applied to the design of a system or program, those applied during development, mitigations applied at run-time, and operational procedures and techniques.

The "Principle of Least Privilege" is a technique that can apply during design and operation. The principle states that a process should operate with only those privileges that are required. Tools like static analysis (automated source code analysis) are applied during development. Some tools operate on running software as part of the development process, such as code-coverage-guided automated fuzzing. American Fuzzy Lop (AFL) and Syzkaller are two such examples which have been used to good effect in FreeBSD. Runtime mitigations apply when software is used in production. Examples include address randomization and limitations on memory protection. Finally, operational procedures and techniques include those employed by the software author or provider, such as the FreeBSD security team's process for issuing security advisories, and by the user or sysadmin, such as setting and documenting configuration options.

Welcome to this security-focused *FreeBSD Journal* issue. We'll touch on a few different areas related to security, including ports, base system userland and kernel topics, and system configuration.

On behalf of the FreeBSD Foundation,
Ed Maste
Senior Director of Technology

FreeBSD Journal

May/June 2021

8 **Seven Ways to Increase Security in a New FreeBSD Installation**
By Mariusz Zaborski

14 **The copyinout Framework**
By Konrad Witaszczyk

25 **Using TLS to Improve NFS Security**
By Rick Macklem

33 **Capsicum Case Study: Got**
By Yang Zhong

3 **Foundation Letter**
By Ed Maste

5 **We Get Letters**
Dear They'll Blame Me for Ransomware
by Michael W Lucas

39 **Practical Ports**
Security Scanning a Jail
By Benedict Reuschling

43 **Events Calendar**
By Anne Dickison

WeGet letters

by Michael W Lucas

Dear Letters Columnist,

The boss says "everything must be secure." I started making a list of things to check for security, and there's no way I can do all this. What do I do?

Thank you,

—They'll Blame Me for Ransomware

Oh, TBMfR, my sweet summer child.

You hit the "rant" button. Buckle in.

I've previously written in this very column about how the word "firewall" means nothing. The term is void, without clarity or purpose. The F-word should be removed from your vocabulary immediately, by armed force if necessary, and replaced by a more specific term that means… something. Anything.

"Security?" It's like that, but even more appalling.

I will readily concede that out in meatspace, these eight distasteful letters have a role. How would you know which people to avoid without the phrase *security guard*? Yes, yes, *authoritarian goon* could serve in its place, but it doesn't precisely roll off the tongue. Social Security? That's a thing. But how do these relate to computing?

As always in these cases, I reach for the Single Source of Linguistic Truth: my Oxford English Dictionary, from that delightful Edwardian era best known as World War Intermission. Computers were people then and understood instructions like "lock up the cipher's secret keys at the end of your shift." We didn't have to define locks, or secrets, or ciphers in sufficient detail that a machine designed to the highest standards of malicious obedience couldn't misunderstand them. Security meant "do it right or the authoritarian goons will smack you until you do." So, let's go to the official definition of this word.

Wait. The official definition fills most of page 370 and spills over onto 371. There's no way I'll quote all that. I'll skim and cherry-pick some definitions that conveniently support my argument.

1. "The condition of being secure."

Defining a word with its own root? That's nearly as helpful as the documentation helpdesk staff give users. Moving on.

2. "The condition of being protected from or not exposed to danger; safety."

Here's my question: does the boss want the staff computers protected from danger, or the staff protected from the dangers of computers? Don't you dare try to tell me that computers don't threaten people; I've seen YouTube, and don't get me started on Myspace or Facestagram or whatever they call it these days.

3. **"Freedom from doubt; confidence, assurance. Now chiefly, well-founded confidence, certainty."**

Computers are not only doubt incarnate, they are unapologetic doubt factories, spewing digital uncertainty every millisecond they're running. We call ourselves engineers, but civil engineers get really upset when a suspension bridge unexpectedly dumps core. It's the sort of thing that makes the news and gets unpopular employees exiled to Farawayistan to maintain that oh-so-vital sham of accountability. In computing, when a server crashes, we check the logs and see there's nothing, so we wait to see if it happens again, all the while desperately hoping it doesn't. Maybe we turn on extra monitoring if we have it. This isn't a matter of laziness. The tools to identify many problems do not exist, and the ones that do exist are beyond the comprehension of the average sysadmin. You can learn the tools, yes, but when you master them, you have to figure out how to fix them and then it's too late, you've become a developer and your life is essentially concluded. Civil engineers at least have the benefit of being able to go look at their bridges and say helpful things like "This critical bolt is starting to bend, maybe we should stop sending trains full of lead across it while I check it out." They'll be told no, of course, but the engineers know to save the memos so that when the bridge dumps core the blame flows uphill.

If you don't want doubt, get a different job. Try something with feral hamsters. They're more rational than computers.

4. **"Freedom from care, anxiety, or apprehension; a feeling of safety or freedom from or absence of danger."**

Oh *heck*, I know—I KNOW—that you did not just try to apply "freedom from care" to anything in systems administration. Anxiety and apprehension are the soul of technology management. Confidence is for the organization's Chief Scapegoat Officer, someone so far removed from the day-to-day operation that they have no idea what's really going on down in the cubicle sewer. People who do the real work understand that computers are untrustworthy. Your test environment is exactly like your production environment, except that the database server has a slightly older CPU lacking two instructions present in production? Guess what's going to bite you? Hint: it's not that. You know about that. Rule 44 of systems administration clearly declares that "a perfect deployment means only that you haven't yet noticed the catastrophe." If you think this rule isn't true, you haven't been paying attention.

Your job description should read "go surfing in a blender."

And that's the real problem. You don't want to get chopped into Sysadmin Smoothie. Especially not for something as daft as the Chief Scapegoat Officer being unwilling to perform his one duty.

I recommend ignoring your boss' instruction in favor of building your own professional reputation.

The word "security" is thrown like a blanket over a bunch of other stuff. Experts who get fancy certifications like the CISSP will tell you that the Security Blanket covers a combination of confidentiality, integrity, and availability. I'm not an expert, because I let my CISSP lapse rather than risk ambush from unscrupulous recruiters armed with trank guns and nets. (If I ever awaken in a cubicle and discover I have a dart in my back, a sedative-induced hangover, and my feet forcibly jammed into—ugh—shoes, the world. Will. Pay.)

The good news is anyone can chant those three words. Confidentiality—the stuff that should be secret, remains secret. Integrity—data isn't mucked with except by authorized muckers. Availability—the computer more-or-less keeps running. The better news is you can take this mantra back to the boss and request clarification. Every organization has its own threats. Nobody knows what they are. Leverage this ocean of ignorance to accomplish three things.

First, address whatever your boss thinks the biggest threat is. That's probably whatever's been in the news most recently. At this exact moment, that's ransomware. This gives you every excuse to deploy a mammoth ZFS-backed fileserver and a snapshot regimen and declare that anything on the server is safe from ransomware.

Second, take advantage of the mandate to choose an interesting project that can be reasonably stuffed beneath the Security Blanket. Learn about proxy servers, or netflow, or DTrace, or tcpdump. Use an article about security flaws in old processors to get everything older than ten years old replaced.

Third, do the best you can with everything else.

Be sure to save all the emails where the boss refuses to let you do things. They might force the Chief Scapegoat Officer to go out and fall on his sword.

Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS's has written over 40 books and resists all efforts to make him stop. One title, *Only Footnotes*, was a collection of all the footnotes from all his books, but he immediately invalidated it by adding more footnotes, so that doesn't count. His most recent titles include *TLS Mastery* and *$ git sync murder*. He's currently writing a second edition of *DNSSEC Mastery*, doubling down on the harm he previously inflicted on civilization.

The FreeBSD Project is looking for

- Programmers
- Testers
- Researchers
- Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website
freebsd.org/projects/newbies.html

Downloading the Software
freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at **freebsdfoundation.org**

Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by

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.

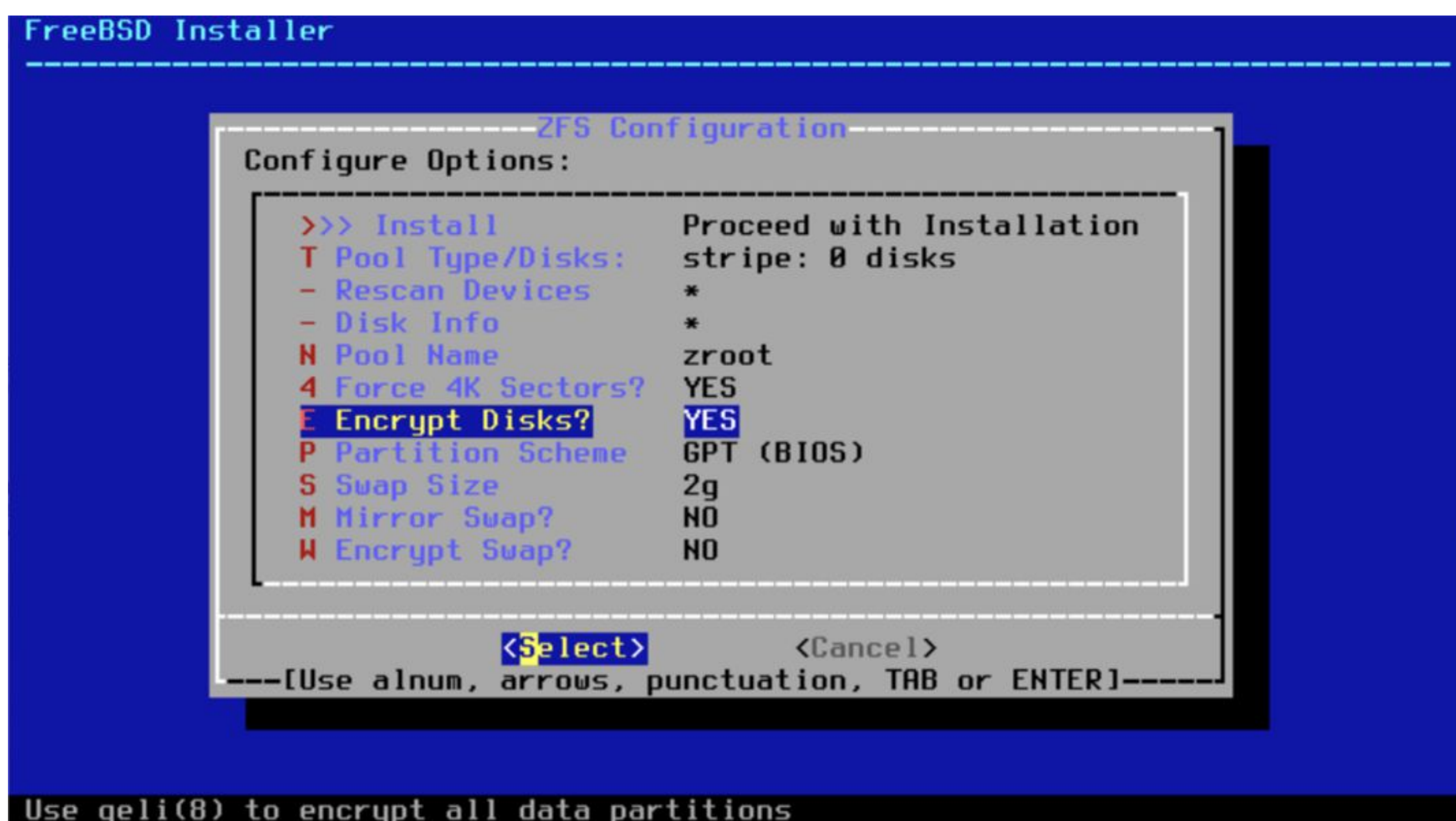
Introduced in version 13.0, OpenZFS native disk encryption is the newest disk encryption mechanism in FreeBSD. OpenZFS supports AES-CCM and AES-GCM with a 128-, 192-, or 256-bit key length. In OpenZFS encryption, not all data is encrypted. Datasets and snapshots, pool layouts or dataset properties are 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:

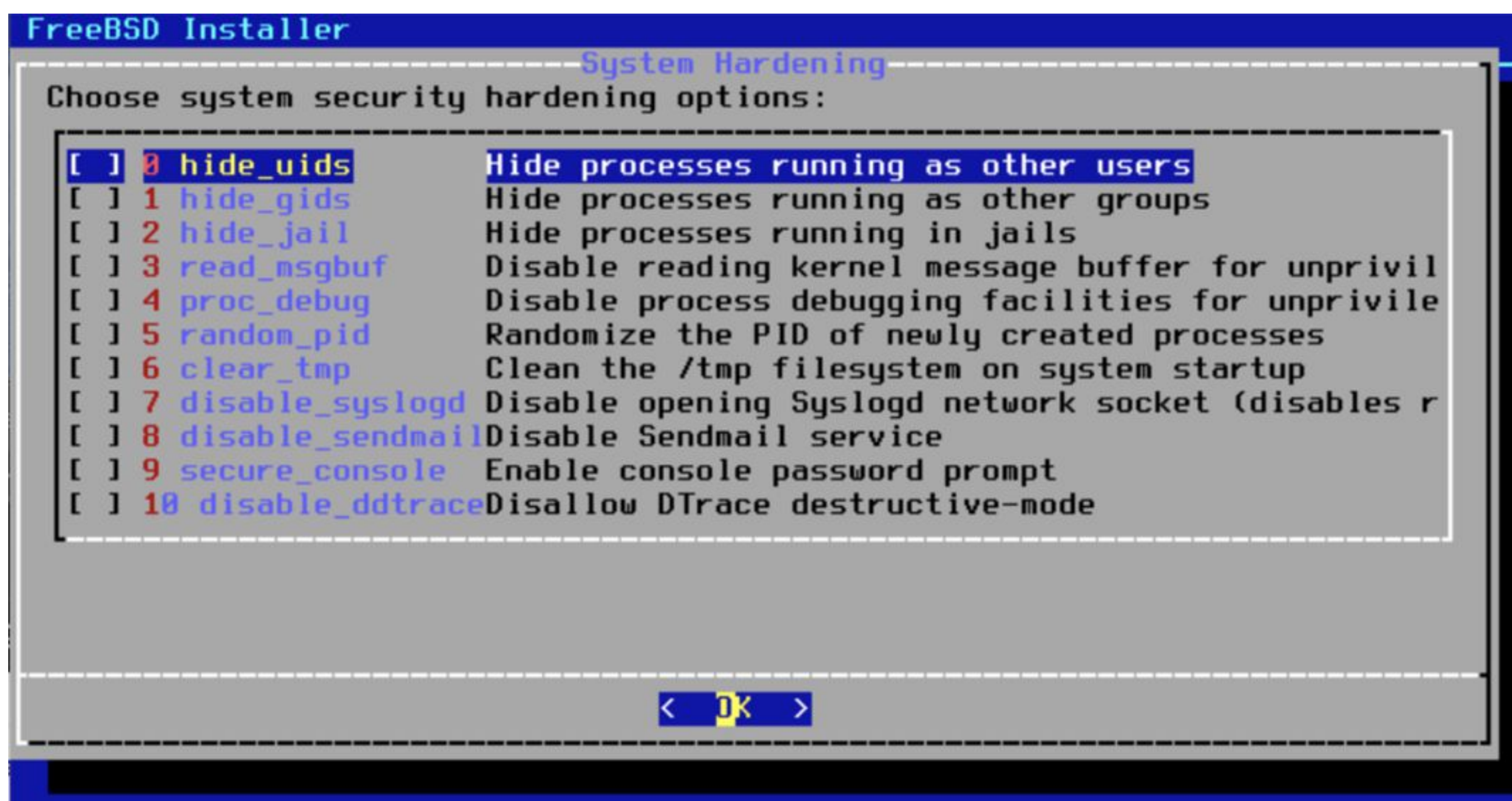
```
# pkg audit
```

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.

The copyinout Framework

ABI-independent, type-aware, capability-aware, copyin and copyout API in FreeBSD and CheriBSD

BY KONRAD WITASZCZYK

Memory copying between user and kernel address spaces is a crucial operation performed as part of system calls. It is used to copy system call arguments as well as a system call result. The current copy function prototypes are type-agnostic and copy a number of bytes from an arbitrary buffer. When a kernel copies its memory to the user space, it must make sure it does not leak any kernel data that might include secrets. This article describes the limitations of the current copy functions in FreeBSD and CheriBSD [1] and proposes a framework that could improve the security and code quality of system call handlers.

The described copyinout framework was implemented as part of the MSc thesis entitled "Capability-aware memory copying between address spaces" [2] under supervision from Ken Friis Larsen, University of Copenhagen, and David Chisnall, Microsoft Research Ltd. The original idea of the type-aware copyin and copyout API was proposed by David Chisnall.

Memory Copy Functions in FreeBSD

FreeBSD includes two main functions that copy memory between address spaces: `copyin()` and `copyout()` (see Listing 1). Both functions take three arguments: a source address, a destination address, and a number of bytes to be copied. `copyin()` copies `len` bytes from the user-space address `udaddr` to the kernel-space address `kaddr`. `copyout()` works in the opposite direction and copies `len` bytes from the kernel-space address `kaddr` to the user-space address `udaddr`. The functions return `0` on success and `EFAULT` if an invalid address was passed.

```
int copyin(const void * __restrict udaddr, void * _Nonnull __restrict kaddr, size_t len);
int copyout(const void * _Nonnull __restrict kaddr, void * __restrict udaddr, size_t len);
```

Listing 1. `copyin()` and `copyout()` function prototypes in FreeBSD 13.0-RELEASE.

Directly from the function prototypes, we can identify one potential security issue. The copy functions operate on arbitrary buffers. In case a buffer contains a structure object with padding between structure fields, the padding is also copied. A padding leak with sensitive information is commonly known as a kernel memory disclosure or a kernel memory leak. Such bugs can result in escalated privileges. They are not specific to FreeBSD and can be found in numerous operating systems [3] [4] [5] as well as they have been a subject of extensive research of detection [6] and mitigation [7] [8] [9] techniques.

The copy functions are used by system call handlers to copy system call arguments from the user space to the kernel space and copy system call results from the kernel space to the user space. Since the system calls are very frequently executed, kernel developers must provide copy function implementations with the lowest possible overhead. This can be achieved by providing machine-dependent implementations in an assembly language. Depending on a CPU architecture, the copy functions can also make use of security features if a CPU model provides them. For example, implementations for amd64 (see `amd64/amd64/support.S`) support Supervisor Mode Access Prevention (SMAP).

ABI Support

FreeBSD includes support for multiple ABIs. In particular:

- Native ABI for programs compiled for the same target as the kernel;
- 32-bit ABI for a 32-bit version of an architecture for which the kernel was compiled;
- Linux ABI for Linux user-space programs.

The ABIs are implemented as compatibility layers. Each compatibility layer provides its system call handlers that implement additional logic required to be executed before or after the kernel enters or returns from kernel routines that operate on native ABI objects. This includes copying and translating objects between address spaces, e.g. for the 32-bit ABI, a pointer in a system call argument or a system call result must be translated from or to a 32-bit pointer.

System Call Handlers

The kernel calls a specific system call handler with copied in system call arguments each time a user-space program makes a system call. For each supported ABI, the kernel keeps a `sysentvec` structure object (see Listing 2) describing ABI-specific properties and functions to be used by the kernel when a program is being executed. The structure includes the `sv_table` array with system call handler function pointers at stored positions specified by their corresponding system call numbers, and the `sv_fetch_syscall_args` function pointer to an architecture-specific function that copies in system call arguments.

As an example, let's consider the `jail(2)` system call. This system call has one argument: a pointer to a `jail` structure object (see Listing 3). The `jail` structure includes parameters describing the prison (see Listing 4). Once a user-space program performs the `jail` system call and enters the privileged mode, the kernel calls the `jail` system call handler — the `sys_jail()` function (see Listing 5) with already copied-in jail system call arguments — a `jail_args` structure object. The system call handler copies in a `jail` structure and calls the `kern_jail` kernel routine that implements `jail` system call logic.

```
struct sysentvec {
   (...)
   struct sysent *sv_table;
   (...)
   int            (*sv_fetch_syscall_args)(struct thread *);
   (...)
}
```

Listing 2. `sysentvec` structure describing ABI-specific properties and functions.

```
struct jail_args {
    char jail_l_[PADL_(struct jail *)];
    struct jail * jail;
    char jail_r_[PADR_(struct jail *)];
};
```

Listing 3. `jail` system call arguments structure for the native ABI.

```
struct jail {
    uint32_t                version;
    char                   *path;
    char                   *hostname;
    char                   *jailname;
    uint32_t                ip4s;
    uint32_t                ip6s;
    struct in_addr         *ip4;
    struct in6_addr        *ip6;
};
```

Listing 4. `jail` structure for the native ABI.

```
int
sys_jail(struct thread *td, struct jail_args *uap)
{
    (...)
    int error;
    struct jail j;

    (...)
           error = copyin(uap->jail, &j, sizeof(struct jail));
           if (error)
                return (error);
    (...)
    return (kern_jail(td, &j));
}
```

Listing 5. `jail` system call handler for the native ABI.

Compatibility Layers

Compatibility layers provide system call implementations for non-native ABIs. In particular, the 32-bit ABI is implemented as the freebsd32 compatibility layer. Let's consider the same `jail` system call for the 32-bit ABI. The 32-bit version of the `jail` arguments structure is called `freebsd32_jail_args` (see Listing 6) and includes a pointer to an object of the 32-bit version of the `jail` structure called `jail32` (see Listing 7). The only differences between the `jail` and `jail32` structures are architecture-independent field types. Each pointer is replaced with a 32-bit unsigned integer. Since pointers are 32-bit unsigned integers in 32-bit architectures, this change guarantees that the `jail32` structure compiled for a 64-bit kernel has the same layout as the jail structure compiled for a 32-bit kernel.

The `jail` system call handler for the 32-bit ABI is implemented as the `freebsd32_jail` function (see Listing 8). The function copies in a 32-bit jail object, translates each field for its

native ABI version using macros (see Listing 9) and calls the same **kern_jail** kernel routine as in the native ABI case. This means that the only difference between the native ABI and the 32-bit ABI in the **jail** system call handler is translating a user-space **jail** object to its native ABI version that can be used by the kernel.

```
struct freebsd32_jail_args {
    char jail_l_[PADL_(struct jail32 *)];
    struct jail32 * jail;
    char jail_r_[PADR_(struct jail32 *)];
};
```

Listing 6. `jail` system call arguments structure for the 32-bit ABI.

```
struct jail32 {
    uint32_t        version;
    uint32_t        path;
    uint32_t        hostname;
    uint32_t        jailname;
    uint32_t        ip4s;
    uint32_t        ip6s;
    uint32_t        ip4;
    uint32_t        ip6;
};
```

Listing 7. `jail` structure for the 32-bit ABI.

```
int
freebsd32_jail(struct thread *td, struct freebsd32_jail_args *uap)
{
     (...)
     int error;
     struct jail j;

     (...)
         struct jail32 j32;

         error = copyin(uap->jail, &j32, sizeof(struct jail32));
         if (error)
               return (error);
         CP(j32, j, version);
         PTRIN_CP(j32, j, path);
         PTRIN_CP(j32, j, hostname);
         PTRIN_CP(j32, j, jailname);
         CP(j32, j, ip4s);
         CP(j32, j, ip6s);
         PTRIN_CP(j32, j, ip4);
         PTRIN_CP(j32, j, ip6);
     (...)
     return (kern_jail(td, &j));
}
```

Listing 8. `jail` system call handler for the 32-bit ABI.

```
#define CP(src, dst, fld) do {              \
     (dst).fld = (src).fld;                 \
} while (0)

#define PTRIN(v) (void *)(uintptr_t)(v)
#define PTRIN_CP(src, dst, fld) do {        \
     (dst).fld = PTRIN((src).fld);          \
} while (0)
```

Listing 9. Helper macros used by the `jail` system call handler for the 32-bit ABI.

The copyinout API

Kernel memory disclosure and code duplication issues described in the previous sections are implications of type-unawareness of the copy functions. If `copyin()` and `copyout()` functions were aware of a structure of an underlying object stored in a copied buffer, they could copy fields of the objects and translate them if a user process ABI differs from the kernel ABI.

To eliminate these problems, we introduce the `copyinout API`. For each type `foo` that is copied between the user space and the kernel space, we introduce type-aware copy function variants (see Listing 10) that copy each field of the type `foo` and translate them from a source ABI to a destination ABI, e.g. a 32-bit pointer is translated to a 64-bit pointer for a 32-bit process on a 64-bit architecture. Additionally, the copy functions can perform additional operations depending on a CPU model, e.g. a kernel compiled for a CHERI CPU [11] can create CHERI capabilities [12] to set bounds or permissions for a field. In contrast to the original copy functions, the type-aware copy functions take only two arguments: a source address and a destination address. `copyin_foo()` copies an object stored at the uaddr user-space address to an object stored at the kaddr kernel-space address. `copyout_foo()` works in the opposite and copies an object stored at the `kaddr` kernel-space address to an object stored at the `uaddr` user-space address. For example, the `jail` structure described in the previous sections could be copied in using the following function call:

```
copyin_jail(uap->jail, &j);
```

```
int copyin_foo(const void *uaddr, struct foo *kaddr);
int copyout_foo(const struct foo *kaddr, const void *uaddr);
```

Listing 10. `copyin()` and `copyout()` function variants for the type `foo`.

The copyinout Framework

Implementations of the type-aware copy functions are independent of the `copyinout API` itself. In order to provide the implementations, we introduce the `copyinout framework` that consists of:

- Type annotations describing what copy functions should be generated for a type;
- A table of copy function pointers for each ABI;
- A kernel interface to dynamically register copy functions and call them;
- A code-generating tool that generates copy functions based on the type annotations.

The kernel defines type annotations that indicate what copy function should be generated:

- `__copyin` to generate `copyin()`;
- `__copyout` to generate `copyout()`;
- `__copyinout` to generate both `copyin()` and `copyout()`.

Additionally, the kernel defines field annotations that describe what value is stored in a field:

- `__uaddr_array(bar)` for a field that stores a pointer to an array with a number of elements stored in the field `bar`;
- `__uaddr_bounded(bar)` for a field that stores a pointer to a buffer with a number of bytes stored in the field `bar`;
- `__uaddr_code` for a field that stores a code pointer;
- `__uaddr_object` for a field that stores a pointer to an object;
- `__uaddr_unbounded` for a field that stores a pointer to a buffer with an unknown number of bytes;
- `__uaddr_str` for a field that stores a pointer to a string and hence its bounds can be computed with `strlen()`.

The field annotations can be used to generate copy functions that make use of security-related mechanisms, e.g., construct CHERI capabilities. Having the `copyinout framework` integrated, a kernel developer who wants to generate new copy functions for the type `foo` defined in Listing 11 must only add appropriate annotations and run the code-generating tool. In this case, the generated copy functions copy the field `len` and translate the pointer stored in the pointer `array`. Additionally, in CheriBSD, they construct the bounded capability array with bounds set to a value stored in the field `len` as part of the field translation.

```
struct foo {
    size_t len;
    int *array;
};
```

```
struct foo {
    size_t len;
    __uaddr_array(len) int * __capability array;
} __copyinout;
```

Listing 11. Type `foo` before and after `copyinout` changes.

In order to provide separate copy functions for different ABIs, the copyinout framework implements the `copyinout table` with type-specific `copyin()` and `copyout()` function pointers (see Listing 12) as part of the `sysentvec` structure (see Listing 13 as compared to Listing 2). Each ABI allocates its own `copyinout` table that is dynamically filled with entries using the SYSINIT framework through the Linker Set technique [10]. The `SYSINIT()` and `SYSUNINIT()` macro calls for the ABIs are generated by the code generating tool alongside generated copy functions. This allows generating copy functions that are used within a kernel module and should be registered and unregistered when the module is loaded and unloaded. With the `copyinout table`, the `copyinout API` can be defined as in-kernel macros that cast pointers and call functions from a `copyinout table` corresponding to an ABI of a currently running thread (see Listing 14). A `copyinout` table entry index for a specific function is a global variable initialized as part of the `SYSINIT()` call. For example, the type `foo` with its generated functions has an associated variable `copyinout_foo_idx` that can be used indirectly by the `COPYIN_CALL()` and `COPYOUT_CALL()` macros to determine function addresses and make function calls (see Listing 15).

```
typedef int copyin_t(const void * __capability uaddr, void * __capability kaddr);
typedef int copyout_t(const void * __capability kaddr, void * __capability uaddr);

struct copyinout {
     copyin_t       *ce_copyin;
     copyout_t      *ce_copyout;
};
```

Listing 12. `copyinout structure` with type-specific copy function pointers.

```
struct sysentvec {
    (...)
    struct sysent *sv_table;
    (...)
    int           (*sv_fetch_syscall_args)(struct thread *);
    (...)
    const struct copyinout *sv_copyinout;
}
```

Listing 13. `sysentvec structure` with the `copyinout` table.

```
#define THREAD_COPYINOUT(thread, type)                              \
     (thread)->td_proc->p_sysent->sv_copyinout[copyinout_##type##_idx]

#define COPYIN_FUN(type)                                            \
     THREAD_COPYINOUT(curthread, type).ce_copyin
#define COPYIN_CALL(type, uaddr, kaddr)                             \
     ((int (*)(const void * __capability,                           \
         struct type * __capability))COPYIN_FUN(type))              \
         (uaddr, kaddr)

#define COPYOUT_FUN(type)                                           \
     THREAD_COPYINOUT(curthread, type).ce_copyout
#define COPYOUT_CALL(type, kaddr, uaddr)                            \
     ((int (*)(const struct type * __capability,                    \
         void * __capability))COPYOUT_FUN(type))                    \
         (kaddr, uaddr)
```

Listing 14. In-kernel macros for `copyinout` API calls.

```
#define copyin_foo(uaddr, kaddr) \
     COPYIN_CALL(foo, uaddr, kaddr)
#define copyout_foo(kaddr, uaddr) \
     COPYOUT_CALL(foo, kaddr, uaddr)
```

Listing 15. `copyinout API` function call macros for the type `foo`.

The main part of the `copyinout framework` is the code-generating tool written in C++ that uses the `libclang` library for code analysis. It traverses input Clang AST trees of header files that include all type definitions for which copy functions should be generated and prints prototypes, definitions or implementations of the copy functions (see Listing 16). The AST trees can be generated with the following command:

```
clang -Xclang -ast-dump -fsyntax-only -c header-file
```

The `copyinout framework` includes a helper script that for an input base source tree generates AST trees, runs the `copyinout tool` and places generated functions in the base source tree. This script could be added to the FreeBSD build system to automate code generation.

Currently, the function implementations can be generated in the C language for any architecture (`generic`) or in the assembly language for MIPS and CHERI-MIPS architectures. For

example, Listing 17 presents a `copyin()` function in the assembly language generated by the `copyinout tool` for the type `foo` (see Listing 11) and the native ABI (a hybrid CheriBSD kernel).

```
$ ./copyinout
usage: copyinout prototypes kernel-space-ast
       copyinout definitions native|freebsd32|cheri kernel-space-ast
       copyinout implementations native|freebsd32|cheri generic|mips user-space-ast
kernel-space-ast
```

Listing 16. `copyinout` tool usage.

```
struct foo {
  size_t len;
  __uaddr_array(len) int * __capability array;
} __copyinout;
```

```
LEAF(native_copyin_foo)
    cgetbase t0 , $c3
    blt t0, zero, _C_LABEL(copyerr)
    nop
    GET_CPU_PCPU(v1)
    PTR_L t0, PC_CURPCB(v1)
    PTR_LA t1 , copyerr

    PTR_S t1, U_PCB_ONFAULT(t0)
    cld v0, zero, 0($c3)
    PTR_S zero, U_PCB_ONFAULT(t0)
    csd v0, zero, 0($c4)

    PTR_S t1, U_PCB_ONFAULT(t0)
    cld v0, zero, 8($c3)
    cfromptr $c5 , $ddc , v0
    cld v0, zero, 0($c3)
    li a0, 96
    multu a0, v0
    mflo v0
    csetbounds $c5 , $c5 , v0
    PTR_S zero, U_PCB_ONFAULT(t0)
    csc $c5, zero, 16($c4)

    j ra
    move v0, zero
END(native_copyin_foo)
```

Listing 17. `copyin()` function generated for the type `foo` and the native ABI (a hybrid CheriBSD kernel).

Memory Copying with the copyinout API

Having the `copyinout framework`, we can simplify system call structures and handlers. For the previously discussed `jail` system call, we can replace the `jail` (see Listing 4) and `jail32` (see Listing 7) structures with a single one (see Listing 18). The `jail` system call for the native ABI (see Listing 5) can be modified to use the type-aware `copyin()` function variant `copyin_jail()` (see Listing 19). Since the `copyin_jail()` function is also ABI-independent and can automatically translate a 32-bit ABI object to its native ABI version, we can also modify the `jail` system call handler for the 32-bit ABI (see Listing 8) to simply call the `sys_jail()` function (see Listing 20).

```
struct jail {
    uint32_t                        version;
    __uaddr_str char * __capability path;
    __uaddr_str char * __capability hostname;
    __uaddr_str char * __capability jailname;
    uint32_t                        ip4s;
    uint32_t                        ip6s;
    __uaddr_array(ip4s) struct in_addr * __capability ip4;
    __uaddr_array(ip6s) struct in6_addr * __capability ip6;
} __copyinout;
```

Listing 18. `jail` structure with `copyinout` changes for all ABIs.

```
int
sys_jail(struct thread *td, struct jail_args *uap)
{
    (...)
    int error;
    struct jail j;

    (...)
            error = copyin_jail(uap->jail, &j);
            if (error)
                  return (error);
    (...)
    return (kern_jail(td, &j));
}
```

Listing 19. `jail` system call handler with `copyinout` changes for the native ABI.

```
int
freebsd32_jail(struct thread *td, struct freebsd32_jail_args *uap)
{
    struct jail_args args;

    args.jailp = uap->jailp;

    return (sys_jail(td, &args));
}
```

Listing 20. `jail` system call handler with `copyinout` changes for the 32-bit ABI.

Conclusion

The initial implementation of the `copyinout framework` shows that copy-function generation can improve code quality in system call handlers and eliminate potential kernel memory leaks in paddings. However, the code-generating tool must be improved to support more complex data types and generate assembly copy function implementations for all supported platforms. While work on the framework has been on hold for a while, we hope to resume it and implement these improvements soon.

Future Work

Besides the basic functionality for the `copyinout framework`, there are several ideas on how it can be improved or applied in the future:

- **Assembly copy function implementation optimizations;**
  The current assembly implementations do not use any optimization techniques to minimize the number of registers or cycles used during copying. For example, two half-word adjacent fields could be loaded as one word with one load instruction and one register instead of two load instructions.
- **System call handlers reductions;**
  Compatibility layers implement many system call handlers that only translate objects between ABIs and not introduce any additional logic to their native ABI counterparts. Having the `copyinout API` applied to them, the system call handlers seem to be redundant. For example, the `jail` system call handler with copyinout changes for the 32-bit ABI (see Listing 20) only copies a pointer to a `jail` object from a `jail` system call arguments structure and calls the `jail` system call handler for the native ABI. It would be interesting to apply the `copyinout framework` to system call argument structures as well and remove such system call handlers from the compatibility layers.
- **Cross-platform compatibility layers for emulators.**
  User-mode emulation in QEMU allows running programs for different architectures than a host architecture without full system emulation. Each time a system call is encountered, QEMU translates system call arguments from an emulated ABI version to a host user-space ABI version, performs a system call on a host and translates a result to its emulated ABI version. We could investigate if it is possible to implement a compatibility layer that includes system call handlers for an emulated platform and using the `copyinout framework` can copy and translate system call objects directly from or to an emulated user space.

References

[1] CTSRD. CheriBSD. FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html.
[2] Konrad Witaszczyk. Capability-aware memory copying between address spaces. University of Copenhagen, 2019.
[3] The FreeBSD Project. FreeBSD-EN-18:12.mem. https://www.freebsd.org/security/advisories/FreeBSD-EN-18:12.mem.asc.
[4] The MITRE Corporation. CVE-2017-16994. https://cve.mitre.org/cgi-bin/cvename.cgi?name =CVE-2017-16994.
[5] The MITRE Corporation. CVE-2010-4082. https://cve.mitre.org/cgi-bin/cvename.cgi?name =CVE-2010-4082.
[6] Mateusz Jurczyk. Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking. https://googleprojectzero.blogspot.com/2018/06/detecting-kernel-memory-disclosure.html.
[7] Alexander Popov. Introduce the STACKLEAK feature and a test for it. https://lwn.net/Articles/735584/.
[8] Kees Cook. mm: Hardened usercopy. https://lwn.net/Articles/693745/.
[9] Thomas Barabosch, Maxime Villard. KLEAK: Practical Kernel Memory Disclosure Detection. https://www.netbsd.org/gallery/presentations/maxv/kleak.pdf.

[10] The FreeBSD Project. FreeBSD Architecture Handbook, Chapter 5. The SYSINIT Framework. https://docs.freebsd.org/en/books/arch-handbook/sysinit/.
[11] Robert N. M. Watson, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020.
[12] Robert N.M. Watson, et al. An Introduction to CHERI. Technical Report UCAM-CL- TR-941, University of Cambridge, Computer Laboratory, 2019.

KONRAD WITASZCZYK is a Research Associate at the University of Cambridge working on the CHERI project. He graduated with a BSc degree in Theoretical Computer Science from the Jagiellonian University, an MSc degree in Computer Science from the University of Copenhagen and spent almost 7 years at Fudo Security working with FreeBSD and its security-related technologies. Currently, Konrad lives in Warsaw, Poland.

"Security. You keep using that word. I do not think it means what you think it means."

Transport Layer Security, or TLS, makes ecommerce and online banking possible. It protects your passwords and your privacy. Let's Encrypt transformed TLS from an expensive tool to a free one. TLS understanding and debugging is an essential sysadmin skill you must have.

TLS Mastery teaches what you must know.

Stop fighting with certificates and start using them. Give them enough attention that you can automate and ignore them.

Learn TLS. Because we're sysadmins and lies do not become us.

TLS Mastery by Michael W Lucas

https://mwl.io

Using TLS to Improve NFS Security

BY RICK MACKLEM

Traditionally, NFS has provided very limited security based on the IP address/DNS host name of the client using exports(5). This can be subverted by IP address spoofing and simply does not work for mobile clients without any fixed, well-known IP address or DNS host name. Also, all data normally travels in clear text on the wire and, as such, can easily be sniffed.

RFC2203 was published September 1997 and provided a mechanism to alleviate at least some of the above issues via the use of GSSAPI with Kerberos mechanism, commonly referred to as Kerberized NFS. When used via the "`sec=krb5p`" (KerberosV with privacy), the RPC message's arguments and results are encrypted on the wire. Kerberos works well for user authentication but is less convenient for machine authentication. Unlike NFSv3, NFSv4 requires a "system principal" which is used to maintain the `Open/Byte_range` lock state on the server. Kerberos has the concept of a host-based Kerberos principal of the form "`host/<FQDN-of-machine>@REALM`", for which a keytab entry can be created and copied onto a client to be used as a "system principal". The "`<FQDN-of-machine>`" instance should protect the keytab entry from being used by another client, if compromised. However, this makes such a Kerberos principal useless for a mobile client without a fixed, well-known DNS host name. Also, for "`sec=krb5p`", only the data payloads of the RPCs are encrypted, exposing the RPC headers and making it impractical to offload the encryption/decryption to specialized hardware. In summary, when combined with the administrative effort involved in implementing a Kerberos environment, "`sec=krb5p`" has not been widely adopted and does not work well for mobile clients without a fixed, well-known DNS host name.

In an effort to improve NFS security, an Internet Draft titled "Towards Remote Procedure Call Encryption By Default" has been written, which describes the use of Transport Layer Security (TLS) to encrypt RPC message traffic on the wire along with the use of X.509 certificates for machine authentication. Since TLS is widely adopted, there are already specialized hardware offload solutions, not to mention efficient software implementations. This article describes the FreeBSD 13 implementation of this NFS over TLS, plus presents an example use case for mobile clients, such as laptops.

Implementation

Although I refer to it as NFS over TLS, it is more correctly Sun RPC over TLS, since the implementation is done in the kernel RPC (krpc) and is largely transparent to the NFS layer. OpenSSL's libraries provide a comprehensive implementation of TLS and handling of X.509 certificates in

user space. However, NFS is implemented in the kernel, and passing all NFS RPC messages up into user address space so that they can be handled by the OpenSSL libraries seemed impractical. Fortunately, FreeBSD 13's kernel added kernel TLS (KTLS) [TLS Offload in the Kernel, John Baldwin, *FreeBSD Journal*,May/June 2020 https://issue.freebsdfoundation.org/publication/?m=33057&i=667002&p=12&ver=html5] that performs efficient handling of TLS Application Data Records, including encryption/decryption, within the network stack in the kernel.

This provided the basic mechanism to encapsulate/encrypt the RPC messages in TLS Application Data Records and decrypt/de-encapsulate those RPC messages on the receive end. It does not, however, handle the non-Application Data Records, such as those used for the TLS handshake. To handle non-Application Data Records, rpc.tlsclntd(8) and rpc.tlsservd(8) were implemented in user space for the client and server respectively. These daemons handle upcalls from the kernel done via custom RPCs using the krpc over an AF_LOCAL socket, in a manner similar to what the gssd(8) daemon does for Kerberized NFS. To handle these upcalls, the daemons perform OpenSSL library calls to do the heavy lifting of handling non-Application Data Records, including handling of the TLS handshake. The daemons also use a custom system call to register with the krpc in the kernel, plus the odd case of needing to associate a file descriptor with an already extant socket in the kernel.

When a client wishes to do NFS over TLS, it performs a STARTTLS Null RPC. A Null RPC is an RPC with no arguments or results and is normally assigned RPC Number 0. To do a STARTTLS, the Null RPC request uses a new RPC credential type of AUTH_TLS. For the NFS service in FreeBSD, if rpc.tlsservd is running, the krpc replies with a credential verifier made up of the eight ASCII bytes "STARTTLS". This STARTTLS probe done by the NFS client triggers a TLS handshake to set up TLS on the TCP connection being used for RPC message transport.

The sequence of actions in the server at this point is:

- Block krpc reception on the TCP socket.
- Send the Null RPC reply with the credential verifier of "STARTTLS".
- Do a handshake upcall to the `rpc.tlsservd`.
  In the rpc.tlsservd to handle the handshake:
  - Acquire a file descriptor for the TCP socket.
    At this point the `krpc` has a TCP socket for the client's NFS connection
    but there is no file descriptor reference for it.
    This was one of the more challenging corners of the implementation.
    My solution was to use the daemon's custom syscall to associate a file descriptor with the socket.
    Once done, closure of the socket is relegated to the daemon instead of the krpc.
  - Add a structure to a linked list for the socket file descriptor,
    keyed on a unique 64bit reference number.
  - Call SSL_set_fd() to associate the socket with an SSL context.
  - Call SSL_accept() to do the actual handshake.
  - If the handshake succeeds, do `BIO_get_ktls_send()` and `BIO_get_ktls_recv()`
    calls to check that KTLS is now enabled on the socket.
    If either of these return zero, the handshake is considered failed.
  - Depending upon what command line options were specified for the daemon,
    any X.509 certificate provided by the client is verified and any
    user mapping specified by the certificate is used to create POSIX
    `<uid, gidlist>` credentials for the user.

- Replies to the upcall RPC with a set of flags indicating whether the handshake succeeded, if a verified certificate was received and POSIX user credentials mapped from the certificate, if any.
  Included in the reply is the unique 64bit reference number for the socket, along with the startup date/time for the daemon, so that the kernel can refer to the socket in subsequent upcalls.
  The startup date/time differentiates the reference number from the same reference number that might be used by a previous or subsequent instance of the daemon.

- If the handshake succeeded, mark the krpc socket as using TLS, along with the flags and credentials, if any, in the upcall's reply.
- Unblock kernel RPC reception on the socket.

The socket should now be ready to handle RPC messages, with the KTLS handling the Application Data Record encapsulation/encryption below the sosend() calls and the decryption/de-encapsulation below the soreceive() calls used by the krpc, if the handshake succeeded.

If a non-Application Data Record is at the head of the socket's receive queue, a new MSG_TLSAPPDATA flag for the soreceive() call indicates that the call should return ENXIO so that the non-Application Data Record will remain at the head of the socket's receive queue. The ENXIO return triggers an upcall to rpc.tlsservd to handle the non-Application Data Record. The kernel code blocks reception on the socket by the krpc and then does the handle record upcall to the daemon. The 64bit reference number, along with the daemon's start date/time are passed up in arguments, so that the daemon can identify the correct socket.

- This upcall simply does a SSL_read() with a length argument equal zero. This call always fails, but processes non-Application Data Records at the head of the socket's receive queue before failing.

The third upcall to the daemon is done to shut down and close the TCP socket, with the 64bit reference number and daemon start date/time as arguments.

- This upcall closes the socket and removes the socket's element from the linked list. If not already done, as indicated by `SSL_get_shutdown()`, this upcall also does a `SSL_shutdown()` before closing the socket, to send a Peer Reset TLS record to the client.

Although all of the above is handled by the krpc, the NFS server does use new flags related to TLS that are passed to the NFS server by the krpc for an RPC to determine if the RPC is permitted, based on the following `exports(5)` options.

There are three new `exports(5)` options:

tls - Indicates that the client must use NFS over TLS, but is not required to present any X.509 certificate to the server during TLS handshake.
tlscert - Indicates that the client must use TLS and must have provided a X.509 certificate during TLS handshake that verified.
tlscertuser - Indicates that the client must use TLS, must have provided a X.509 certificate during TLS handshake that verifier and that this certificate must have successfully mapped to a POSIX user

credential (<uid, gid_list>).
This mapping is generated from a login name found in the otherName component of subjectAltName with a "@domain", where "domain" matches the one the server uses.
This mapping is only generated if rpc.tlsservd is started with the -u/--certuser command line option.

If none of the above exports(5) options were specified, TLS is permitted, but not required.

There is also a command line option for rpc.tlsservd that specifies that the daemon require that the rDNS name for the client's IP address match the "DNS" component of subjectAltName in the client's X.509 certificate. This is analogous to what RFC 6125 recommends that a client do to verify the identity of a domain-named application service. Since this option is intended to subvert client IP address spoofing, exports(5) cannot be used, since it is keyed on the client's IP address. As such, this option specifies that all clients doing NFS over TLS satisfy this criterion and failures result in handshake failures. It is the strongest client host identity check but requires that all clients have X.509 certificates that verify and where the DNS component of subjectAltName is correct. All clients must also have fixed, well-known DNS addresses when this option is specified.

The client daemon functions is a similar manner, but with some differences. Unlike rpc.tlsservd, rpc.tlscltnd only requires a certificate if the -m/--mutualverf command line option is specified. The client can also handle multiple certificates stored in different files, in case different NFS over TLS servers require different certificates.

When an NFS mount establishes a new TCP connection to the server, where the "tls" mount option has been specified, the krpc will do the following:

- Send the Null RPC request with the credential of type AUTH_TLS.
- If a Null RPC reply with a credential verifier consisting of the ASCII bytes "STARTTLS" is received.
  - Block kernel RPC reception on the new TCP socket.
  - Do a handshake upcall to the rpc.tlsclntd.
    In rpc.tlsclntd to handle the handshake:
    - Acquire a file descriptor for the TCP socket.
      At this point the krpc has a TCP socket for the client's NFS connection but there is no file descriptor reference for it.
      This is done by the daemon's custom system call, similar to rpc.tlsservd.
    - Call SSL_set_fd() to associate the socket with an SSL context.
    - If the daemon was started with the command line option -m/--mutualverf, SSL_[ctx_]use_certificate_file()/SSL_[ctx_]use_PrivateKey_file() are called to provide a certificate during the handshake.
      An argument for the upcall may override the default names for the certificate/key files.
      The default names are "cert.pem" and "certkey.pem", but may be overridden on a per-mount basis via the "tlscertname" mount option, in case different NFS servers require different certificates.
    - Call SSL_connect() to do the actual handshake.
    - If the handshake succeeds, do BIO_get_ktls_send() and

```
BIO_get_ktls_recv() calls to check that KTLS is now enabled on the socket.
If either of these return zero, the handshake is considered failed.
If the handshake is successful:
- Add a structure to a linked list for the socket file descriptor,
  keyed on a unique 64bit reference number.
- Reply to the upcall RPC indicating the handshake succeeded.
  Included in the reply is the unique 64bit reference number for the
  socket, along with the startup date/time for the daemon, so that the
  kernel can refer to the socket in subsequent upcalls.
else:
- close the socket.
- Reply failure to the kernel, which will result in all subsequent
  NFS RPCs failing with EACCES.
```

- Upon receiving the upcall reply, the krpc sets a flag if the handshake succeeded and unblocks krpc reception on the socket.

For the client, if either the STARTTLS Null RPC or TLS handshake fails for a mount when the "tls" option has been specified, all RPCs will fail with EACCES. This is done so that NFS mounts with the "tls" option specified will not function unless TLS is working for the mount.

Mobile Devices Such as Laptops as a Use Case

A mobile device, such as a laptop, typically accesses the Internet from anywhere with no fixed, well-known IP address. To allow a laptop to mount an NFSv4 file server from anywhere on the Internet requires some reasonable security mechanism. Although NFS over TLS can be used for NFSv3 mounts, enabling NFSv3 mounts from anywhere is awkward, since the Mount protocol uses a dynamically assigned port number via rpcbind whereas NFSv4 mounts only use port #2049. A such, this example will use NFSv4 mounts.

It is possible to use Kerberized NFS with privacy to do a mount from a FreeBSD laptop. The laptop user would need to do commands such as:

```
# sysctl vfs.usermount=1   - done as su/root
# service gssd onestart - done as su/root
% kinit         - to acquire a TGT for the user
% mount -t nfs -o sec=krb5p,nfsv4,minorversion=1 nfsv4-server.uoguelph.ca:/ /mnt
              - done as the non-root user
- Use the mount point.
% umount /mnt
```

By using the mount option "`sec=krb5p`", but not the "`gssname`" mount option, the FreeBSD client will use the "user principal" as the "system principal". This mount breaks badly if the user's TGT expires before the "umount".

As far as I know, this has not been widely adopted, possibly due to the effort required to install Kerberos and maintain a KDC accessible from anywhere on the Internet.

To do this using NFS over TLS requires the generation of a X.509 certificate for the client. Although there are many ways to create/sign X.509 certificates, this can easily be accomplished by a site local CA, managed by the openssl(1) command in FreeBSD 13.

- Create a certificate for the laptop, signed by the site local CA.
  Using `openssl(1)` the commands might be:
  ```
  # openssl genrsa -aes256 -out certkey.pem
  # openssl req -new -key certkey.pem -addext
  "subjectAltName=otherName:1.3.6.1.4.1.2238.1.1.1;UTF8:rmacklem@uoguelph.ca"
  -out req.pem
  # openssl ca -in req.pem -out cert.pem
  ```
- Copy `cert.pem` and `certkey.pem` into a directory named `/etc/rpc.tlsclntd` on the laptop in some secure manner.
- Enable the client daemon, using the certificate.
  - edit /etc/rc.conf and add:
  tlsclntd_flags="-m"
  - Due to the "-aes256" option, rpc.tlsclntd will query for the passphrase when starting, so it may be preferred to start the daemon manually before doing the mount instead of at boot time.
  - to start at boot time, add to `/etc/rc.conf: tlsclntd_enable="YES"`
  - or start it manually before doing the mount via: `# service tlsclntd onestart`
  - Once the laptop is connected to the Internet, the mount can be done as `su/root:`
  ```
  # mount -t nfs -o nfsv4,minorversion=1,tls nfsv4-server.uoguelph.ca:/ /mnt
  ```

Since the client presents a certificate signed by the site local CA, the server can be reasonably assured that the client has a certificate created by the site local CA administrator. The "-aes256" option used when creating the client's private key forces the rpc.tlsclntd to query for a passphrase to be entered for the key when rpc.tlsclntd is started. This subverts a trivial compromise where the laptop is stolen, or the certificate/key files are copied to another client. For the certificate to be used on an unauthorized client, the passphrase would have to somehow be captured/cracked.

It is also possible to revoke a certificate and add it to a CRL if for any reason the laptop should no longer be allowed to do the mount.

For the above example, all RPCs done on the server will be performed using the POSIX credentials (`<uid, gid_list>`) of the login name "rmacklem" on the NFSv4 server. This avoids any need for the laptop to have a uniform uid, gid space with respect to the server. It also limits the risk due to a compromised laptop to files accessible by "rmacklem". This optional configuration may not conform to the Internet draft. A co-author of the draft agrees that mapping client RPC credentials to a specific user based on the X.509 certificate presented during the TLS handshake is useful and has in fact coined the term "TLS Identity Squashing" for it. However, this individual would prefer a database that maps the certificate's <issuerName, serialNumber> tuple to the "user". He argues that putting the "user" in the certificate conflagrates "machine" vs "user" credentials. Being a pragmatist, I feel that putting the "user" in the certificate is just an easy way to implement this. The rpc.tlsclntd could be modified to use a flat file/database to implement this, if that were to become preferred practice.

The above is just one example use case. The command line options on the daemons allow a range of configurations, ranging from only requiring TLS to encrypt RPC messages on the wire to requiring all clients to present verifiable X.509 certificates where the DNS component of subjectAltName must match the rDNS name for the client's IP host address. The client may also verify the authenticity of the NFS server in the manner recommended by RFC 6125 for TLS domain-named application servers.

To set up NFS over TLS on FreeBSD 13, see:
https://people.freebsd.org/~rmacklem/nfs-over-tls-setup.txt
plus the man pages for `rpc.tlsclntd(8)`, `rpc.tlsservd(8)`, `exports(5)`, `ktls(4)` and `mount_nfs(8)`.

RICK MACKLEM For over 30 years, starting in 1980, Rick was the system administrator for a CS department, running BSD systems among others. When the VAX 11/780 running 4.3BSD was being replaced by MicroVAXII systems, there was a need for NFS on 4.3BSD, so Rick implemented that and contributed it to CSRG. A Usenix paper titled "Lessons Learned Tuning the 4.3BSD NFS Implementation" described the first implementation of NFS over TCP done as a part of this work. Now retired, Rick continues to work on NFS implementation for FreeBSD.

Contact Jim Maurer
with your article ideas.
(jmaurer@freebsdjournal.com)

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!

2021 Editorial Calendar

- Case Studies (January-February)
- FreeBSD 13 (March-April)
- Security (May-June)
- Desktop/Wireless/Graphics (July-August)
- Cloud (September-October)
- Embedded (November December)

Find out more at: freebsdfoundation/journal

CASE STUDY

Capsicum Case Study: Got

BY YANG ZHONG

I've been working with Capsicum for some time as part of my internship with the FreeBSD Foundation. This article details my process of applying the Capsicum sandboxing framework to a large program called Got. Along the way, I'll give a simple and concrete introduction to Capsicum: what problems it deals with, the reasoning behind its solutions, and how to use it. We will find that Got is particularly well-suited to Capsicum, and I'll discuss how Got's structures make the program Capsicum-friendly.

Capsicum Concepts

Capsicum exists to fix a straightforward problem with computer programs: they have too much power. I like to think of it like this. In a world without Capsicum, if I log in to my computer and run some program, *there is nothing stopping that program from deleting my entire home directory without warning*. Of course the program probably will never do that intentionally, but it has enough power to do so. This becomes a real concern when thinking about security: if someone finds a vulnerability in a program, they could exploit it to do anything the program can do. Therefore, if the program is wielding an excessive amount of power, the attacker can use it to do an excessive amount of damage.

In comes Capsicum[1], which provides tools to control a program's power. One important concept Capsicum deals with is that of a global namespace. Essentially, a *global namespace* is a group ("space") of objects, each of which have an identifier ("name") that uniquely identifies it among all objects ("global"). An easy example of a global namespace is the file system: the space is the group of all files, and each file's absolute path is its unique 'name'. The FreeBSD operating system has many global namespaces[2], but the file system is ubiquitous and very important; I'll be talking about it a lot from now on.

Capsicum-less programs deal frequently with global namespaces. When these programs want to `open()` a file, they can pass in a file's absolute path. While this seems normal enough, the program is actually using its tremendous power here: "Out of every single file on this computer, I want this one!" While file permissions and such will prevent the program from accessing every file, this amount of power is certainly in the 'delete-your-whole-home-directory' range.

Even worse, when the program is exercising its power like this, it's exercising the power implicitly. It does not say: "I would like to exercise my power to access every file, and here is a 'key' to verify that I have this power"; the program just always has the power. This power to do things 'by default' is called *ambient authority*[3]. When accessing these global namespaces, programs are always exercising their ambient authority.

Therefore, Capsicum recognizes global namespaces as basically an uncontrollable source of power, and so introduces capability mode — a state in which a program cannot use glob-

al namespaces at all, and therefore also have no ambient authority. Programs 'enter' *capability mode* by calling `cap_enter()`[4], and cannot ever leave. In capability mode, a program cannot `open()` new files, and is therefore restricted to file descriptors that it `open()`-ed before calling `cap_enter()`, or file descriptors provided through a connection to a different process. This environment that limits the resources a program has access to is called a *sandbox*, which is why Capsicum is described as a *sandboxing technique*.

Capsicum does more than this. Another important concept is that of capabilities, which in Capsicum's case are objects that extend file descriptors; they let you finely limit what any file descriptor is capable of doing, and serve the more hidden role of making capability mode actually work. There is also the Casper library, which provide common services for Capsicumized programs.

The Target: Got

So, with these tools, we'd like to adapt some existing programs to use them. I was tasked with adapting to Capsicum the version-control system Game of Trees, or Got for short. It's being developed by and for OpenBSD developers, but the FreeBSD project is considering adding Got to the base system. So as part of this effort, it was my job to figure out how to tweak Got to be more amenable to Capsicum, without drastically changing anything: Ideally, we would make structural changes clean enough to incorporate into the upstream version of Got, so that the FreeBSD Capsicum version of Got is as similar to it as possible.

Capsicumizing Got

On a high level, a common pattern for Capsicumized programs has the program be separated into two parts. In the first part, the program acquires the resources it needs; in the second, the program does its 'work' of reading from and writing to those resources. The program enters capability mode right after the first part. Since the program is stuck in capability mode after this, its power is limited in the dangerous second part, in which it works with the external and untrustworthy resources it acquired in the first.

Many programs are not separated in this way. Often, they acquire resources wherever it's convenient, resulting in the two parts being delicately interleaved. Before Got, I dealt with this issue in the program sort(). In these situations, helper libraries like Casper are invaluable, as they exist to solve common Capsicumization problems that would otherwise take a lot of setup work to fix. You can see an simple example of the interleaving issue in *Case studies of sandboxing base system with Capsicum, by Mariusz Zaborski (EuroBSDcon 2017)* on Youtube, part of which describes the process of Capsicumizing the program `bspatch()`.

Fortunately for me, Got is structured in how it gets its files. Got works with two main directories: a repository and a worktree. If you know Git, these are quite similar to Git repositories and worktrees[5]. Got then has the functions `got_repository_open()` and `got_worktree_open()`, responsible for looking for the repository/worktree and returning a `struct - struct got_repo` and `struct got_worktree` respectively — containing information about the two directories[6].

After this point, Got exclusively works within these two directories (and /tmp), which means that it never tries to acquire anything 'new'. This avoids the interleaving problem discussed earlier, but Got still uses the global file system namespace to actually open new files — for example, the `got_repo` struct contains the absolute path to its associated repository, and so Got would open the directory using that path whenever it needs to. This is not compatible with ca-

pability mode.

In that case, must I pre-open every single file inside the two directories, so that I can use them in capability mode? Thankfully not. When you `open()` file, you get its file decriptor. For non-directory files, its descriptor lets you access just that file. However, a directory's file descriptor allows you to access everything inside that directory.

For this purpose, FreeBSD, by way of POSIX, provides the `*at()-family` of system calls. Where the normal calls take in absolute paths, the `*at()` calls take in a file descriptor and a relative path. If I wish to open the file "`/dir/subdir/a`", and I have a file descriptor `fd` for dir, I can call `openat(fd, "subdir/a")`. This form of access is allowed in capability mode, barring some exceptions[7], since we are no longer searching through the global namespace of all files.

It's easy to see how this helps us with Got, as we know that Got will always work within two specific directories! If we pre-open the repository and worktree directories and store their file descriptors inside the `got_repo` and `got_worktree` structs, we can later use those descriptors to open files inside those directories, even in capability mode. In Got, functions that operate on files inside the repository or worktree will take in a `got_repository` or `got_worktree` as a parameter, meaning that the file descriptor we added will be easily accessible there.

```
static const struct got_error
update_blob(struct got_worktree *worktree,
    struct got_fileindex *fileindex, struct got_fileindex_entry *ie,
    struct got_tree_entry *te, const char *path,
    struct got_repository *repo, got_worktree_checkout_cb progress_cb,
    void *progress_arg)
{
    const struct got_error *err = NULL;
    struct got_blob_object *blob = NULL;
    char *ondisk_path;
    unsigned char status = GOT_STATUS_NO_CHANGE;

    truct stat sb;
    if (asprintf(&ondisk_path, "%s/%s", worktree->root_path, path) == -1)
    return got_error_from_errno("asprintf");

    // example of usage
    int opened_file_fd = open(ondisk_path, 0);
```

The above snippet of Got's code shows a function that takes in a `got_worktree struct`, and uses it to construct a path to a file in that directory. I've added an example of how the function would typically use the new path.

Below is the same code, converted to use our new file descriptor strategy.

```
static const struct got_error
update_blob(struct got_worktree *worktree,
    struct got_fileindex *fileindex, struct got_fileindex_entry *ie,
    struct got_tree_entry *te, const char *path,
    struct got_repository *repo, got_worktree_checkout_cb progress_cb,
```

CASE STUDY

```
    void *progress_arg)
{
    const struct got_error *err = NULL;
    struct got_blob_object *blob = NULL;
    char *ondisk_path;
    unsigned char status = GOT_STATUS_NO_CHANGE;

    struct stat sb;
    int path_fd_part = worktree->root_fd;
    char *path_relative_part = path;

    // example of usage
    int opened_file_fd = openat(path_fd_part, path_relative_part, 0)
```

It's quite simple! Simpler than the first one, even, since the asprintf() call is no longer needed. In these types of situations, adapting Got to support Capsicum is easy.

Some functions don't take in these structs, but instead take in an absolute path that they operate on. Adapting these functions to be Capsicum-compatible takes more work, as we must change their parameters from an absolute path to a pair of (relative path, directory file descriptor), in order for the function to be able to access the file in capability mode.

In practice, this is usually only a small problem. The absolute path the function takes in doesn't come from nowhere — it must have been created by using the `got_repo` or `got_worktree structs`, and therefore the file descriptor we need won't be far away. Below is a function whose parameters needed to be changed as a part of Capsicumization:

```
const struct got_error *
got_fileindex_entry_update(struct got_fileindex_entry *ie,
-    const char *ondisk_path, uint8_t *blob_sha1, uint8_t *commit_sha1,
-    int update_timestamps)
+    int wt_fd, const char *ondisk_path, uint8_t *blob_sha1,
+    uint8_t *commit_sha1, int update_timestamps)
{
     struct stat sb;
-    if (lstat(ondisk_path, &sb) != 0) {
+    if (fstatat(wt_fd, ondisk_path, &sb, AT_SYMLINK_NOFOLLOW) != 0) {
         if (!((ie->flags & GOT_FILEIDX_F_NO_FILE_ON_DISK) &&
         errno == ENOENT))
-                return got_error_from_errno2("lstat", ondisk_path);
+                return got_error_from_errno2("fstatat", ondisk_path);
         sb.st_mode = GOT_DEFAULT_FILE_MODE;
         } else {
...
```

Since the parameters of the function changed, we also need to alter all the places where it was called. In some places, the calling function created the path, using the `got_worktree`

struct, that it passes into `got_fileindex_entry_update()`; for these, we already have the necessary file descriptor, and so adapting to the new parameters is easy:

```
...
       * Preserve the working file and change the deleted blob's
       * entry into a schedule-add entry.
       */
-      err = got_fileindex_entry_update(ie, ondisk_path, NULL, NULL,
-      0);
+      err = got_fileindex_entry_update(ie, worktree->root_fd,
+      ie->path, NULL, NULL, 0);
  } else {
...
```

Some calling functions took in the path as a parameter as well, simply passing it through to `got_fileindex_entry_update()`. For these, we must similarly change the calling function's parameters:

```
 static const struct got_error *
-sync_timestamps(char *ondisk_path, unsigned char status,
+sync_timestamps(int wt_fd, const char *path, unsigned char status,
 struct got_fileindex_entry *ie, struct stat *sb)
 {
      if (status == GOT_STATUS_NO_CHANGE && stat_info_differs(ie, sb))
-         return got_fileindex_entry_update(ie, ondisk_path,
+         return got_fileindex_entry_update(ie, wt_fd, path,
          ie->blob_sha1, ie->commit_sha1, 1);
...
```

Ultimately, the path must originate from a function that has access to `got_worktree`, and so the file descriptor can always be threaded through the calls. It's certainly not a clean solution, especially if the thread gets long, but I've yet to find a thread longer than two calls.

Wrap-up

I hope you've been convinced at this point that making Got work with capability mode is simple. While I've only committed to Got the very beginnings of the work needed, I suspect that most of the necessary changes will be similar to what you've seen.

Of course, not every program is so amenable to Capsicum. Fundamentally, capability mode works well with programs that deliberately manage their resources. Got makes its main resources — the worktree and repository directories — into structs in the code. If a function wants to operate on one of these resources, it needs the struct to do so.

In this way, the code is explicitly saying: "This function will need this resource". Additionally, since Got works with few other resources, the code is saying "This function will need this resource only". This explicitness is opposite to ambient authority, and is exactly what capability mode wants! The rest of the work lies in just enforcing these limitations.

CASE STUDY

1. ...Along with other frameworks, with similar goals but different designs, such as seccomp for Linux and pledge/unveil for OpenBSD. Much has already been written comparing these frameworks; Jonathan Anderson's "A comparison of Unix sandboxing techniques" takes a detailed look.
2. You can find a comprehensive list in "Capsicum: practical capabilities for UNIX" by Robert N.M. Watson et. al.
3. "Capability Myths Demolished" by Mark S. Miller et. al. gives a clear description of ambient authority.
4. Practically, you'll be using the 'Capsicum helpers' and calling caph_enter(), but it's essentially the same thing.
5. In fact, Got can be used with normal Git repositories, hence the similar name.
6. One of the big mistakes I made here was that I tried to enter capability mode *before* the got_worktree_open and got_repo_open functions — It did work after a lot of hacking, but it left a huge mess, and later the lead developer of Got helpfully told me that those functions weren't doing anything dangerous anyway so it's okay to call them before entering capability mode; From this I realized that it's very important to *understand the code before trying to apply Capsicum*. It sounds obvious, but I learned it the hard way.
7. The path can't be absolute, the path can't use ".." components to 'escape' out of the directory, and the file descriptor can't be AT_FDCWD.

YANG ZHONG is studying Computer Science at the University of Waterloo. He worked as an intern with the FreeBSD Foundation for the Fall 2020 term as part of the University's co-operative education program and returned for the Spring 2021 term. In his spare time he enjoys writing for *mathNEWS*, the University of Waterloo math faculty's student publication.

Thank you!

The FreesBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.

Are you a fan of FreeBSD? Help us give back to the Project and donate today! **freebsdfoundation.org/donate/**

Please check out the full list of generous community investors at freebsdfoundation.org/donors/

Uranium

Koum Family Foundation

Iridium

Platinum

NETFLIX

Gold

JUNIPer NETWORKS

Silver

BECKHOFF

Microsoft

moz://a

vmware

STORMSHIELD

Tarsnap

PRACTICAL PORTS

Security Scanning a Jail

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

In this installment, I stray a bit from presenting a bouquet of ports and just focus on a single one: security/lynis.

Lynis is a tool for security auditing, hardening, and compliance testing. What lynis does differently from other security scanners is that it tries to detect available components on a system such as a webserver or database. Once it finds them, it checks them further for vulnerabilities, missing patches, etc. This way, the scans are different on each system based on the configured software and purpose. For example, your firewall host may receive different examinations than your backup server. Plugins extend lynis's functionality to cover specific software. A comprehensive security report is generated at the end, eliciting either a pat on the back from your security-minded superiors or your next security sensitivity training.

Available as open source as well as a commercial enterprise tool, lynis scans a range of operating systems for security anomalies.

Available as open source as well as a commercial enterprise tool, it scans a range of operating systems for security anomalies. Sysadmins as well as penetration testers, developers, and auditors can use it to assess if there are any vulnerabilities, not only in installed software but also in their configurations. Since this includes the operating system as well as third-party software, running it on a FreeBSD jail should prove to be an interesting experiment.

For this purpose, I created an iocage jail from a freshly updated FreeBSD 13.0 host. Note that you can use any other jail management framework or build a jail by hand to repeat this experiment yourself. Entering the console, I run "pkg install lynis" and nothing else (not even my favorite shell), just a plain vanilla jail. This way, we can see what lynis detects in a default installa-

tion. If something comes up before any to-be-jailed software is installed, this is something every FreeBSD 13.0 installation (and possibly versions before that) is concerned about—inside or outside a jail.

Before I ran the scan, I looked at the current settings using "lynis show settings". Don't worry about the license-key line, the software works just fine without any limitations in the open-source version. Without further ado, I start the scan by issuing "lynis audit system".

After some initialization, my correctly-detected OS version and hardware platform (amd64) are echoed to the screen. No big surprise there, but it gets interesting further on. The "Boot and Services" section found that by default, 10 services are running ("service -e" will display them in a standard FreeBSD system). Eight modules, including the kernel itself, are loaded, which was actually detected from the host running the jail and forwarded into it. But still, all green so far, until we get to the "Users, Groups and Authentication" section, where the first red warnings are issued. Administrator accounts, unique UIDs, and login shells all seem to be a problem for lynis.

```
[+] Boot and services
------------------------------------
  - Service Manager                                     [ bsdrc ]
  - Checking presence FreeBSD loader                    [ FOUND ]
  - Checking services at startup (service/rc.conf)      [ DONE ]
    Result: found 10 services/options set
```

```
[+] Users, Groups and Authentication
------------------------------------
  - Administrator accounts                              [ WARNING ]
  - Unique UIDs                                         [ WARNING ]
  - Checking chkgrp tool                                [ FOUND ]
    - Checking consistency of /etc/group file           [ OK ]
  - Login shells                                        [ WARNING ]
  - Unique group IDs                                    [ OK ]
  - Unique group names                                  [ OK ]
  - Password hashing methods                            [ OK ]
  - Query system users (non daemons)                    [ DONE ]
  - NIS+ authentication support                         [ NOT ENABLED ]
  - NIS authentication support                          [ ENABLED ]
  - Sudoers file                                        [ NOT FOUND ]
  - PAM password strength tools                         [ OK ]
  - PAM configuration file (pam.conf)                   [ NOT FOUND ]
  - PAM configuration files (pam.d)                     [ FOUND ]
  - PAM modules                                         [ NOT FOUND ]
  - LDAP module in PAM                                  [ NOT FOUND ]
  - Determining default umask
    - umask (/etc/profile and /etc/profile.d)           [ OK ]
    - umask (/etc/login.conf)                           [ WEAK ]
  - LDAP authentication support                         [ NOT ENABLED ]
```

The shells section lists unsecured console TTYs. Note that I only list the offenders here, the rest is either OK-ish or simply not active yet due to missing software. Checking and re-checking the scan after installing additional software for any changes that may have opened an attack vector is good practice.

Lynis not only scolds you for things you should have done better, but it also suggests things like checking /tmp and /var mount points in the "File systems" section. Don't be discouraged if a first scan finds a bunch of issues. Consider it as an overview and a general guideline for improvement. Some of these things are trivial to fix, which subsequent audits should identify as such. If you set up another host in the future, you can already check for these to avoid repeating that mistake.

```
[+] File systems
------------------------------------
  - Checking mount points
    - Checking /tmp mount point                         [ SUGGESTION ]
    - Checking /var mount point                         [ SUGGESTION ]
```

Many sections in the lynis report are empty simply because we did not yet install anything. If there would have been some software prone to be a typical attack target, lynis would scan that more thoroughly and add additional results to the report. Overall, this FreeBSD jail does not appear to have many issues. Looking at what was found, one problem seems to be the ownership of home directories. I'm not worried yet since there are at present no users other than root itself on this jail. I do make a note to myself to re-check this, though.

In the "Kernel Hardening" section, there is a list of TCP/IP related sysctl settings that should have a different setting than the default. This means that i.e. net.inet.icmp.drop_redirect is set to 0 by default, but should have been set to 1 (active). There is probably a reason other than "oversight" why this is not set. Perhaps the reason is to make FreeBSD work by default in as

many networking environments as possible, where this option might prevent or cause problems. Since the whole list is presented, one can easily set each of those to the recommended value and see if networking still runs normally. If it does, keeping this option on is a good idea.

At the end of the scan, the report lists 4 warnings and 15 suggestions. The easiest warning to fix is the last one, running pkg audit -F to fetch the latest security vulnerability database for ports. The link provided at each suggestion and warning gives further details about the issues and their impacts. Not all of these are of the we'll-lose-our-customer-database-if-we-don't-fix-this-at-once type. Some of them are good sysadmin practices like "avoid /tmp running full", so don't just ignore them as non-security related. Often, security incidents happen not because of a single issue, but because a malicious person was able to combine several unrelated problems into a bigger nightmare.

```
[+] Kernel Hardening
------------------------------------
  - Comparing sysctl key pairs with scan profile
    - hw.kbd.keymap_restrict_change (exp: 4)                [ DIFFERENT ]
    - kern.sugid_coredump (exp: 0)                          [ OK ]
    - net.inet.icmp.bmcastecho (exp: 0)                     [ OK ]
    - net.inet.icmp.drop_redirect (exp: 1)                  [ DIFFERENT ]
    - net.inet.ip.accept_sourceroute (exp: 0)               [ OK ]
    - net.inet.ip.check_interface (exp: 1)                  [ DIFFERENT ]
    - net.inet.ip.forwarding (exp: 0)                       [ OK ]
    - net.inet.ip.process_options (exp: 0)                  [ DIFFERENT ]
    - net.inet.ip.random_id (exp: 1)                        [ DIFFERENT ]
    - net.inet.ip.redirect (exp: 0)                         [ DIFFERENT ]
    - net.inet.ip.sourceroute (exp: 0)                      [ OK ]
    - net.inet.tcp.always_keepalive (exp: 0)                [ DIFFERENT ]
    - net.inet.tcp.blackhole (exp: 2)                       [ DIFFERENT ]
    - net.inet.tcp.drop_synfin (exp: 1)                     [ DIFFERENT ]
    - net.inet.tcp.icmp_may_rst (exp: 0)                    [ DIFFERENT ]
    - net.inet.tcp.nolocaltimewait (exp: 1)                 [ DIFFERENT ]
    - net.inet.tcp.path_mtu_discovery (exp: 0)              [ DIFFERENT ]
    - net.inet.udp.blackhole (exp: 1)                       [ DIFFERENT ]
    - net.inet6.icmp6.rediraccept (exp: 0)                  [ DIFFERENT ]
    - net.inet6.ip6.forwarding (exp: 0)                     [ OK ]
    - net.inet6.ip6.redirect (exp: 0)                       [ DIFFERENT ]
    - security.bsd.hardlink_check_gid (exp: 1)              [ DIFFERENT ]
    - security.bsd.hardlink_check_uid (exp: 1)              [ DIFFERENT ]
    - security.bsd.see_other_gids (exp: 0)                  [ DIFFERENT ]
    - security.bsd.see_other_uids (exp: 0)                  [ DIFFERENT ]
    - security.bsd.stack_guard_page (exp: 1)                [ OK ]
    - security.bsd.unprivileged_proc_debug (exp: 0)         [ DIFFERENT ]
    - security.bsd.unprivileged_read_msgbuf (exp: 0)        [ DIFFERENT ]
```

For me, exploring items like "Umask in /etc/login.conf could be stricter like 027" is worth some time. Who knows, maybe that will become the new default in an upcoming FreeBSD release (better than part of a security advisory) once secteam has had time to review it.

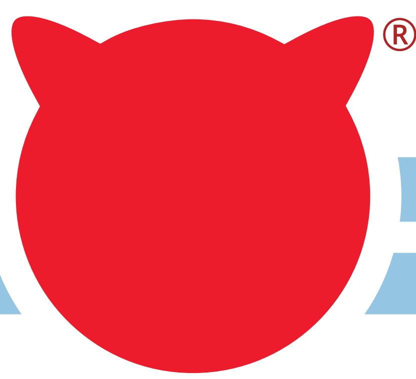
Remember that this small experiment only covered the operating system and may reveal more once the jail is doing what it was set up to do. As additional software and services are run and exposed to the internet or even to users on the local network, it will mean that additional scans are advised. Check out the other modes that lynis provides, including the penetration testing mode, as it may reveal additional items for your security to-do list.

Lynis is not the only tool out there. Wise security-minded folks are known to use multiple tools to cover a wider range of possible issues to find (or confirm) another tools suspicion. Tools like portsentry, nmap, nessus, snort, as well as the not-yet-ported terrascan and openvas-scanner, are all fine tools to detect and help close potential security problems before it is too late. Always be suspicious of these tools not finding everything or not finding the latest incidents. Running them regularly is not an excuse to not check security bulletins for the software you run and subscribe to their notifications.

Oh, I just realized that I did mention more than a single tool here. But who am I to keep these to myself?

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Together with Allan Jude, he is host of the weekly bsdnow.tv podcast.

Support FreeBSD®



Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

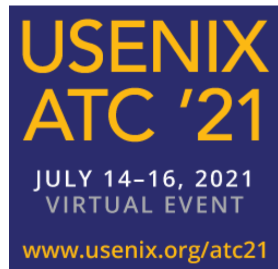
freebsdfoundation.org/donate

Events Calendar

BSD Events taking place through September 2021

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.



USENIX ATC '21
July 14-16, 2021
VIRTUAL

The 2021 USENIX Annual Technical Conference will take place as a virtual event on July 14–16, 2021. USENIX ATC brings together leading systems researchers for the presentation of cutting-edge systems research and the opportunity to gain insight into a wealth of must-know topics, including virtualization, system and network management and troubleshooting, cloud and edge computing, security, privacy, and more. Check out the technical sessions program. USENIX ATC '21 is co-located with the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21). Register for one, and get access to both!

USENIX OSDI '21
July 14-16, 2021
VIRTUAL

The 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21) will take place as a virtual event on July 14–16, 2021. Join us from wherever you are at the premier forum for discussing the design, implementation, and implications of systems software. The symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation. View the technical sessions program. This year, OSDI is co-located with the 2021 USENIX Annual Technical Conference. Register for one, and get access to both!

FreeBSD

EuroBSDcon 2021
September 16–19, 2021
Vienna, Austria

EuroBSDcon is the European annual technical conference gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects.

FreeBSD Fridays
https://freebsdfoundation.org/freebsd-fridays/
FreeBSD Fridays will begin again in July.
Past FreeBSD Fridays sessions are available at: https://freebsdfoundation.org/freebsd-fridays/

FreeBSD Office Hours
https://wiki.freebsd.org/OfficeHours
Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.