FreeBSD® JOURNAL

November/December 2023

FreeBSD 14.0

Linux Boot: Booting into FreeBSD from Linux

FreeBSD Container Images

Kick Me Now with Webhooks

Trip Report: Oslo Hackathon

Interview: Joel Bodenmann

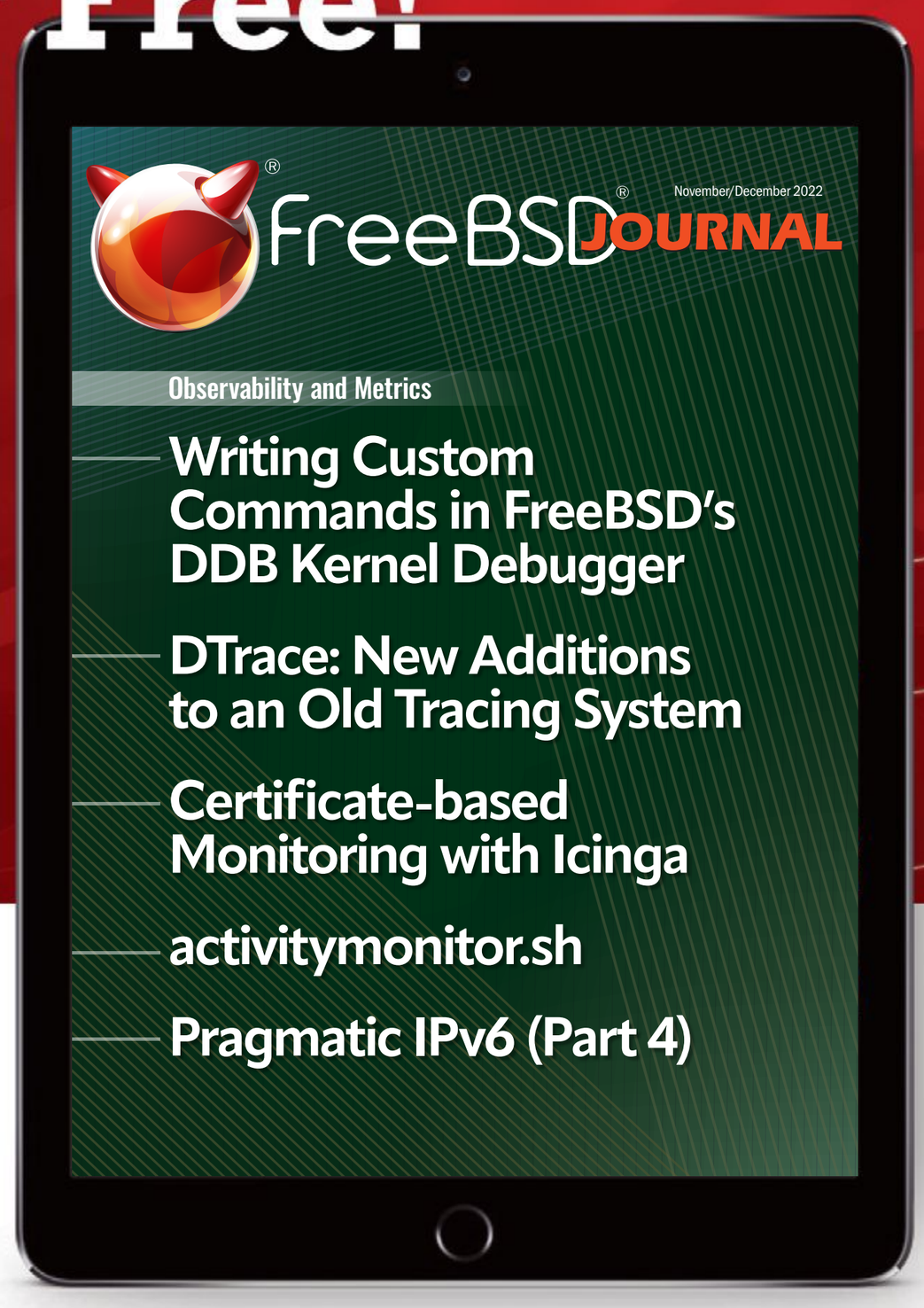
FreeBSD® JOURNAL

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!



2024 Editorial Calendar

- Networking (January-February)
- Development Workflow and CI (March-April)
- Configuration Management Showdown (May-June)
- Storage and File Systems (July-August)
- To come (September-October)
- To come (November-December)

Find out more at: freebsdfoundation/journal

FreeBSD® JOURNAL

S&W PUBLISHING LLC
PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) 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: 720/207-51 42 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Foundation

A New Release is On the Way!

Welcome to the September/October issue. As I write this, FreeBSD is putting the finishing touches on its next major release: 14.0. Stay tuned, as articles in the November/December issue will cover many of 14.0's exciting new features.

In the meantime, the current issue provides some great reading while you're waiting for 14.0 to hit a CDN server near you.

Tom Jones continues with his column aptly named "Recollections." In this installment, Tom interviews Warner Losh, a long-time FreeBSD contributor and famous for melting laptops--among other stories.

Benedict Reuschling provides readers with a detailed walkthrough of using poudriere(8) to build local packages--from setting custom options to deploying signed package repositories to multiple client machines. Also, on the packages front, Charlie Li presents a brief history of packaging extensions in the Python ecosystem and the work to adapt FreeBSD's ports tree to the most recent iteration.

To add jails to the mix, Alonso Cárdenas details the use of FreeBSD jails as the basis for a cybersecurity training platform using the existing open source tools Wazuh and Caldera.

One of the great things about the FreeBSD community is meeting up with members at events around the world. This issue contains a trip report from CCCamp 2023 and the November/December issue will highlight a report from EuroBSDCon 2023.

We enjoy hearing from readers and benefit from your communications with us. If you have feedback on published articles, suggested topics for future articles, or would like to write for the Journal, please email us at info@freebsdjournal.com.

John Baldwin
Chair of *FreeBSD Journal* Editorial Board

FreeBSD® JOURNAL
November/December 2023

FreeBSD 14.0

LinuxBoot: Booting FreeBSD from Linux

BY WARNER LOSH

After maintaining the FreeBSD UEFI boot loader for years at Netflix, our hardware designer asked me to improve system booting resilience. After failing to find a solution with the stock UEFI firmware, I looked for alternatives. Most alternative firmwares boot the OS using a stripped-down Linux kernel, so I implemented LinuxBoot support for FreeBSD. I'll describe how I did that, but first I'll explain what LinuxBoot is, where it came from, what it does, and where it fits in. In a second article, I'll describe the nuts and bolts of creating LinuxBoot firmware images that boot FreeBSD.

Over the past five years a new variation on booting has matured and become popular. Spurred on by a desire to reduce the attack surface for the boot phase, Linux now boots Linux. This seemingly awkward arrangement has advantages over traditional UEFI booting. Some new embedded environments offer only LinuxBoot. Facebook, Google, IBM, Microsoft, and Apple have supported these efforts to improve security, reduce boot complexity, and provide a common platform largely independent of underlying architecture. FreeBSD 14.0 offers preliminary support for booting FreeBSD/aarch64 and FreeBSD/amd64 with a new variation of /boot/loader, called "loader.kboot," which uses LinuxBoot.

(A note on terminology: I use "aarch64" and "amd64" because they are more visually distinct than "arm64" and "amd64," which are easily confused.)

Three major themes have led us to the point where LinuxBoot is gaining popularity.

How We Got Here

Three major themes have led us to the point where LinuxBoot is gaining popularity: initial simplicity, uncontrolled growth, and a desire to return to a simpler time. All three themes have contributed to the complex booting ecosystems that have sprung up on both x86 and embedded systems. LinuxBoot attempts to simplify those ecosystems somewhat, though having the entire Linux kernel involved usually doesn't make one immediately think of simplicity.

In the days before the IBM PC, most systems either required a bootstrap to be entered manually or the automated boot ROMs were simple enough to load a boot sector and it would load the rest of the system with it. This process of using simple loaders to load progressively more complex loaders is called "bootstrapping" the system, from the old saying "pulling yourself up by your own bootstraps." In time, this was shortened to "booting" the system.

In 1982, IBM's release of the IBM PC only slightly improved on prior systems by providing both the bootstrapping code in its ROMs and other basic I/O. It called these services the BIOS from the term used by CP/M systems that preceded it. This is where we get the term "BIOS" and why there's some confusion surrounding whether it means "the firmware used

to bootstrap the system" or whether it only refers to the pre-UEFI style of booting on the PC platform. Partisans of both stripes are sure they are right, but few recognize the history behind this ambiguity. So, I'll use "CSM" or "CSM booting" to refer to this style of bootstrap. The UEFI Standard uses the term "CSM" to describe legacy booting, and it is unambiguous.

In time, all kinds of additional features were added to CSM booting: a partition table for disks, the MP Table for processor configuration, APM for power management, SMBIOS for metadata about the system, run time services for PCI, PXE network booting, and ACPI to unify many of the prior features. The interfaces for these services were tied to x86-specific mechanisms. The system evolved into a very complex ecosystem, riddled with quick hacks, special cases, and subtly different interpretations.

When Intel designed the IA-64 CPU architecture in the late 1990s, it quickly discovered that the revolutionary architecture couldn't use the vast majority of techniques from the CSM ecosystem. The whole booting ecosystem had to be replaced. This was the genesis of Unified Extensible Firmware Interface (UEFI) booting. At first, it was just on the Intel x86 (re-branded IA-32) and IA-64 architectures. During the early 2000s, UEFI firmware slowly displaced the legacy systems on Intel x86 systems, oftentimes being able to do either the old CSM booting or the new UEFI booting. By 2006, Intel had released, via its TianoCore project, the EDK2 open source development kit for UEFI firmware creation, prompting even more OEMs to adopt UEFI.

In time, all kinds of additional features were added to CSM booting.

Also, during the 1990s and 2000s, another booting ecosystem was evolving for embedded systems. Initially, the embedded space had dozens of different bootloaders, all subtly different in their interface to later stages of booting. Mangus Damm and Wolfgang Denk created Das U-Boot in 1999, first for PowerPC but later ARM, MIPS, and other architectures. U-Boot started out small and simple, more flexible than its competitors, and relatively easy to extend because it was open source. It quickly became the universal bootloader because of its simplicity, support, and rich feature set. It set the standard for booting, and drove many features in Linux, including flattened device tree (FDT) support. This boot system had nothing in common with either CSM or UEFI. It was so easy to use that all its competitors have faded into obscurity. Where are redboot, eCos, CFE, yaboot, or YAMON today? Footnotes on a Wikipedia page at best.

In 2011, ARM introduced aarch64, its 64-bit version of the ARM platform. Both U-Boot and EDK2 vied for dominance to bootstrap the system, while FDT and ACPI vied for dominance to enumerate system devices. Low-end, embedded systems tended to use U-Boot with FDT, while higher-end, server-class systems used UEFI and ACPI. Eventually, UEFI booting started to win out, especially when U-Boot started to ship a minimal UEFI implementation sufficient to boot Linux via UEFI. ACPI and FDT merged (you can specify ACPI nodes now with FDT properties). And through it all, EDK2/UEFI grew more and more complex to support SecureBoot, iSCSI, more NICs, RAM disk support, initramfs support, and too many other features to list.

And that doesn't even count all the bootloaders using a stripped-down version of the Linux kernel, such as coreboot, slimboot, LinuxBIOS, and others, some of which I'll describe below. Nor does it begin to describe the variation between commercial BIOSes. By 2017, Google decided to do something about this situation and started project NERF to simplify this mess and harden security. The name stands for Non-Extensible Reduced Firmware, as opposed to UEFI's Unified Extensible Firmware Interface. It is also slang from computer gaming for a change that downgrades the power or influence of a game element to achieve a better balance or improve enjoyment of the game. These efforts would later become LinuxBoot.

Linux Booting

Booting Linux with Linux has a very long history, but space allows only a brief summary. In the mid-1990s, when the kexec(2) family of system calls was added to Linux, it was used to increase uptime and/or reliability of servers and embedded systems. Ron Minnich and Eric Biederman started LinuxBIOS at Los Alamos in the 1990s to use a Linux kernel in the Firmware to boot the system. This evolved into coreboot, used by Chromebooks and several open platform laptops. In the process, coreboot became modular to allow binary blobs alongside the open source components because CPU manufacturers have resisted opening up the early processor initialization code, providing only binary blobs to both open and closed source firmware creators. EDK2, U-Boot and closed-source firmware have also developed a modular system to allow these binary blobs to exist alongside other components.

They wanted to create a common framework using widely deployed and reviewed code.

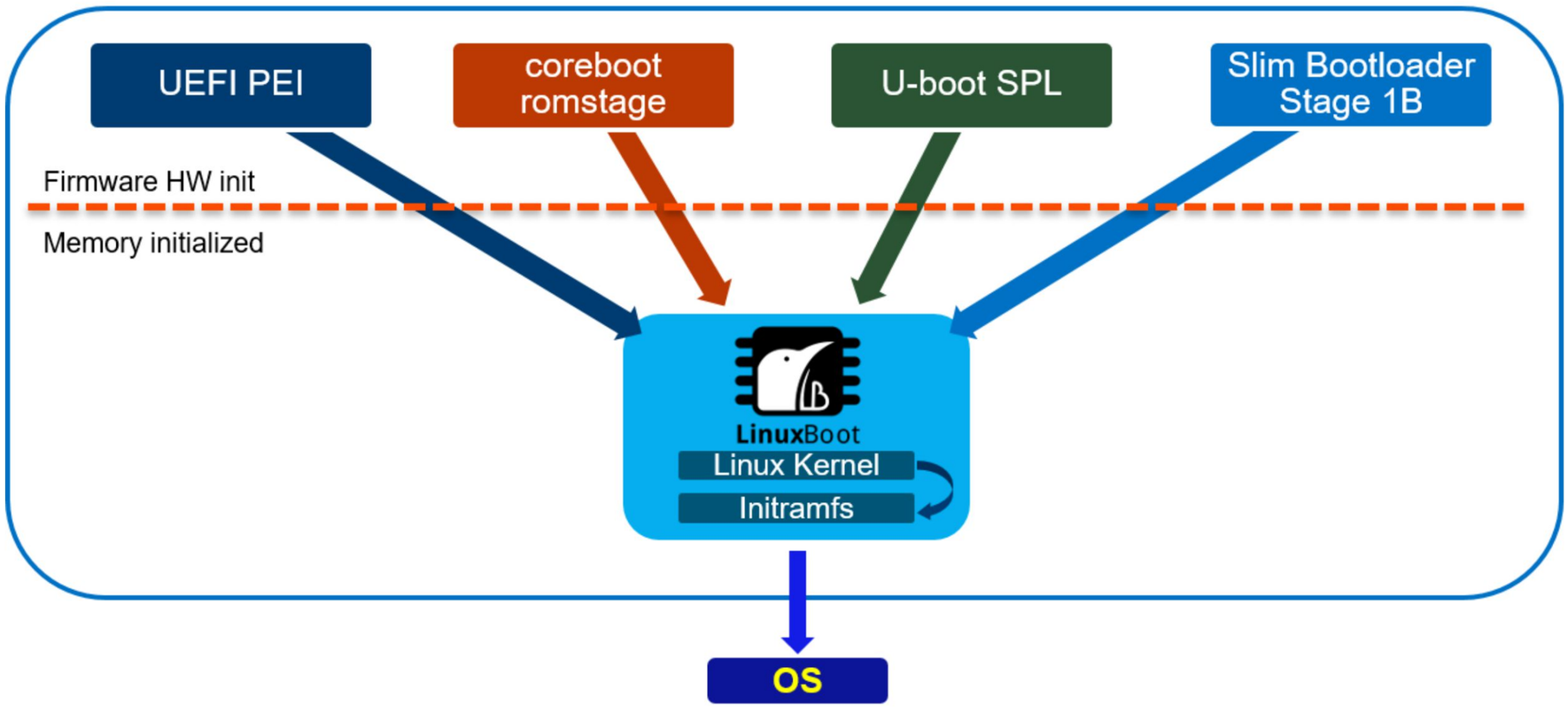
Project NERF Becomes LinuxBoot

Google's NERF project, headed by Ron Minnich, evolved into the LinuxBoot project: a series of scripts that help create firmware images that boot the final OS with the Linux kernel. The project had several bigger-picture goals behind it, however. Google wanted to create an open source firmware where every component was freely available. They wanted to simplify the UEFI booting environment, which they felt had grown too complex with too many potential security vulnerabilities, by replacing it with the hardened Linux kernel. They wanted to create a common framework using widely deployed and reviewed code; minimize the unavoidable, binary-only, non-source portions of the bootloader; unify booting for the ARM and other embedded systems as much as possible; eliminate redundant code, speeding the boot; and create a more modular and customizable boot experience than traditional firmware or even EDK2 provided. They wanted a reproducible build, which ensures anybody can run the exact same binary whether they download it or build it themselves.

The result was a modular system that supported many bootloaders. The earliest stages of initializing the CPU were handled by CPU-specific and bootloader-specific code. LinuxBoot defined what parts of the system were initialized there and which parts were deferred to the Linux kernel. This setup allowed CPU vendors to continue to ship binary-only blobs that initialized the low-level clocks, memory controllers, auxiliary cores, etc. that a modern

CPU needs to become functional. EDK2, coreboot, U-Boot, and slim boot all support these protocols, so the Linux kernel boots with all of them, without special code for any of them. LinuxBoot also provided u-root, a ramfs builder written in go for finding and loading the final and a few other tools to manipulate firmware images. I'll discuss these tools and how to use them in the second part of this article.

SPI Flash



Source: https://www.linuxboot.org

While not completely successful at replacing the entire bootloader with Linux, LinuxBoot minimized the amount retained. For example, with UEFI, only the Pre-EFI Initialization (PEI) phase initializing the processor, caches, and RAM, and UEFI's runtime services remained. LinuxBoot eliminated all of the thinly tested UEFI DEX drivers. The Linux kernel takes over with the memory and base hardware initialized, but without any of the other things that a more traditional firmware might initialize, like resources for PCI devices.

In addition to better security and more control over the firmware, LinuxBoot uses the well-tested, highly reviewed Linux drivers required for Linux to run on the platform. With LinuxBoot, SOC vendors and system integrators can optimize their time to market by writing drivers only for Linux. UEFI DEX drivers need not be created at all. Programmers with Linux driver skills are much easier to find than those who can write a UEFI DEX driver. The Linux kernel has been audited by thousands of researchers, compared to relatively few who have studied the EDK2 UEFI code base. These advantages, however, require other operating systems supporting UEFI to adapt. Their UEFI bootloaders do not work with the sliver of UEFI that remains. This means that to boot at all on these systems, an OS must create a new loader to support LinuxBoot.

Some simpler operating systems boot with LinuxBoot using the basic ELF loading that Linux's kexec-tools package has to offer. Very old versions of BSD and Plan9 have been booted this way. FreeBSD/powerpc, which runs on a processor from a simpler time with a well-defined OpenFirmware interface, also loads this way. Windows, however, cannot boot this way, and the LinuxBoot community is researching ways around it. FreeBSD/amd64 and FreeBSD/aarch64 cannot boot this way either.

FreeBSD kernels for amd64 and aarch64 require metadata available only to the bootloader. On amd64, the bootloader sets the system into long mode after capturing information about the system's memory layout and other data, which can only be accessed before

entering long mode. The kernel relies on this data and cannot operate without it. On both amd64 and aarch64, the bootloader has to inform the kernel of the address of the UEFI system table and other system data. The bootloader tunes the kernel by setting "tuneables." The loader pre-loads dynamic kernel modules and passes things like initial entropy, UUID, etc. into the kernel. None of this specialized knowledge is present in the kexec-tools that can only load an ELF binary and jump to its start address.

FreeBSD and LinuxBoot

FreeBSD's history with booting via Linux goes back over a decade. In 2010, the FreeBSD PS/3 port used the PS/3's "another OS" option to start. Nathan Whitehorn, a FreeBSD developer, added the necessary glue to the FreeBSD bootloader to set up memory for FreeBSD's kernel. He created a small Linux binary similar to Ubuntu's kboot from their PS/3 support package. This Linux binary was statically linked and included the few system calls needed to read the FreeBSD kernel off the PS/3 disk. It included a small libc (similar to mucl or glibc) and command line parsing support. However, the structure of its sources assumed only PowerPC.

While working at Netflix, I began experimenting in 2020 to see how hard it would be to boot FreeBSD with Linux. Netflix runs a large fleet of servers installed throughout the world. Through years of constant refinement, Netflix created a very robust system that corrects common problems automatically. Even after these refinements, booting issues caused too many costly RMAs. Experiments using UEFI scripting to improve boot-time reliability provided only marginal improvements. Because flash drives contained the scripts, only a few trivial cases improved. Flash drives can fail read-only, confusing both the UEFI firmware and the scripts into doing the wrong thing. As long as the scripts remained on the drives, progress was impossible.

Netflix began experimenting in 2020 to see how hard it would be to boot FreeBSD with Linux.

LinuxBoot offered an attractive alternative to UEFI because it resided inside the firmware on the motherboard, eliminating the components most prone to failure. Netflix wanted me to create a fail-safe environment that could phone home status information about the machine, reprovision the machine using surviving NVMe drives, and provide a flexible platform to enable remote debugging, diagnostic images, etc.

I had several goals with booting FreeBSD from Linux:

1. It had to be built within the FreeBSD build system.
2. It had to provide full access to host resources.
3. It had to boot with a stock kernel (if possible).
4. It had to use the UEFI boot interface (i386 CMS booting, and arm U-Boot binary booting would not be supported).
5. It had to run as init/PID 1.
6. It also had to run well when called from shell scripts to support booting different kinds of images not necessarily based on FreeBSD.

Getting FreeBSD booting from Linux on modern architectures like amd64 and aarch64 required several changes to the relatively modest PS/3 kboot base. The loader needed four types of changes: refactoring the existing kboot following the MI/MD model, expanding

support for accessing host resources, refactoring UEFI boot code to be used by both the UEFI loader.efi and the LinuxBoot loader.kboot, and retiring technical debt within the bootloader.

MI/MD Changes

Several areas needed the classic MI/MD split where common MI code interfaces with per-architecture MD code that implements a common API. Linux has a much larger difference in system calls between architectures than FreeBSD. Program startup needs slightly different assembler glue between architectures. Different linker scripts are needed. The loader metadata, while mostly similar, has architectural differences. Finally, the handoff from the Linux kexec reboot vector to the kernel differs. I'll cover these last two below in the Refactoring UEFI Booting section.

The first three of these changes are needed to create Linux binaries. To create static binaries, I wrote the C runtime support that provides the glue between the Linux kernel handoff and a more traditional main routine. I wrote a bit of per-architecture assembly, coupled with a standard startup routine that calls main. To accomplish this a standard C interface to the system calls allowed the MD part of FreeBSD's mini libc for Linux to be small. I created a small amount of per-architecture assembler for system calls. I added a framework for Linux's per-architecture ABI differences, the largest being in the termios interface. This reflects Linux's complicated history of binary compatibility. A per-architecture linker script produces a Linux ELF binary. These elements combine to make loader.kboot, Linux ELF binary. The new libsa drivers (see below) interface to this libc.

Accessing Host Resources

The original loader.kboot code accessed some host resources, but it was incomplete. I wanted to boot either off of a raw device, or via a kernel or loader residing in the filesystem of the host system. The bootloader has always supported a number of different ways to specify where files come from but before refactoring, adding new one was hard. With some changes to refactor the existing code, I added the ability to access any block device from its Linux name. The name "/dev/sda4:/boot/loader" reads the file /boot/loader that's within the fourth partition on the sda disk, for example. In addition, "lsdev" now lists all of the eligible Linux block devices. The bootloader discovers zpools. For example, "zfs:zroot/kboot-example/boot/kernel" specifies a kernel to boot. Finally, it can be convenient to put the kernel and/or bootloader directly in the Linux initrd. The bootloader itself uses this feature to get necessary data from the /sys and /proc filesystems. Any mounted filesystem can be accessed with "host:<path-to-file>". So, you could boot from "host:/freebsd/boot/kernel" or read the Linux memory usage with "more host:/proc/iomem." The loader also supports mapping the "/sys/" or "/proc/" prefixes to the host's /sys and /proc filesystems, regardless of active device.

Loader.kboot can replace /sbin/init inside the Linux initrd. Init is the first program to run and must do extra steps to prepare the system. Loader.kboot will notice when it is running as init and do these extra steps before starting: mounting all the initial filesystems (/dev, /sys, /proc, /tmp, /var), creating a number of expected symbolic links, and opening stdin, stdout and stderr. The loader runs either in this environment or as a process launched from one of the standard Linux startup scripts. At present, loader.kboot is unable to fork and execute Linux commands.

Refactoring UEFI Booting

Reflecting the long history of booting in general, FreeBSD's boot process had co-evolved with its kernel for the past 30 years in the case of amd64 and for the past 20 years or so in the case of aarch64. Neither of these architectures has been around for this entire time, of course, but amd64 inherited many of the idiosyncrasies of i386, and aarch64's boot, while vastly cleaner, is the product of 20 years of embedded FreeBSD systems. To boot successfully in this complex environment, loader.kboot needed to recreate all these quirks. It follows the UEFI protocols by creating the same metadata structures that loader.efi, our UEFI bootloader, creates.

These efforts started with amd64 since it was more readily available for experimentation. I selected the UEFI + ACPI boot environment to emulate. UEFI was the newer, more flexible interface and seemed to have fewer special cases. While theoretically the FreeBSD kernel could boot from either UEFI or CSM and not know which one it booted from, the practical reality differed. The kernel expected to get UEFI-derived data in certain ways, and BIOS-derived data in slightly different ways. It became clear early on that trying to support both was limiting progress as oftentimes two different paths needed to be written and debugged. Since UEFI will be with us for a long time (even though with LinuxBoot only a tiny sliver of UEFI survives), and CSM might not be, I decided to limit my support to UEFI on amd64.

The efforts started with amd64 since it was more readily available for experimentation.

Despite this simplification, progress was still too slow as I had to discover via trial and error all the quirks the amd64 kernel depended on from the bootloader. Fellow FreeBSD developer Mark Johnston suggested that aarch64 would be easier to get working, since that had a simpler set of interfaces. This proved to be true. Once I had the basic translation from the UEFI data structures to FreeBSD's loader metadata working, I got much further booting aarch64. There were only a couple of bugs that I will talk about later. I'm planning on getting amd64 working next now that aarch64 works.

The loader needs a few hundred lines of code to set up the metadata from UEFI. This code, unsurprisingly, expects to run in a UEFI runtime. It allocates memory using UEFI APIs, gets memory information from UEFI, and fetches the ACPI tables in ways specific to UEFI. I needed to refactor this code so that it could create the proper metadata structures from the /sys and /proc filesystems on Linux. In addition, Linux provides both FDT and ACPI data with device descriptions in ACPI only. This tricked FreeBSD into thinking no devices are present, however, since it favored FDT for device enumeration when both are present. Linux only provides data necessary to do another kexec via FDT, but no device data.

In addition to the normal UEFI data structures and booting, after the kexec, Linux leaves the hardware in a state subtly different from either a cold or warm boot from the firmware. So the system is not quite in a fully reset state. Normally, this doesn't matter—we can boot from that state and put all hardware into its correct state. But there were problems.

UEFI Boot Services was my first problem. When Linux exits UEFI's "boot services," it creates memory mappings where the virtual address (VA) does not match the physical address (PA). FreeBSD's loader.efi always creates 1:1 mappings where PA is equal to VA (so called PA = VA). Since the memory mapping may only be set once, FreeBSD's kernel must use the map Linux creates. The kernel would panic because the mapping was not PA = VA. Fortunately, the panics were due to restrictive assertions left over from debugging loader.efi. Removing the assertions exposed a bug where PA was used instead of VA, but the kernel booted after I fixed that bug.

The second problem I encountered was the "gicv3" issue, where "almost reset" coupled with a device erratum spelled big trouble. There is a design flaw with the gicv3 interrupt router: once it has been started (and Linux starts it fully when booting), it can't be stopped short of a full system reset and initialization (which kexec cannot do). To work around this problem, the FreeBSD kernel had to reuse this memory. Linux passes the gicv3 state data via the UEFI system table structure. This table contains a list of physical addresses reserved for gicv3 use. FreeBSD parses this table, ensures it matches what the gicv3 is using, and marks them as reserved so FreeBSD's memory allocation code doesn't hand them out. Everything worked with QEMU; however, when we tried to run in an aarch64 machine, I was very surprised to discover this problem. Thankfully, the Linux community had previously discovered this problem and had a set of patches that I could use to fix FreeBSD in a similar way.

One not so surprising thing that I discovered during this project was that the bootloader had a lot of copied and pasted code to implement path and device name parsing.

Retiring Technical Debt

One not so surprising thing that I discovered during this project was that the bootloader had a lot of copied and pasted code to implement path and device name parsing. Copies of this code differed in non-obvious ways. Sometimes the changes were bug fixes, but software archaeology showed other copies retained the bugs. Other times, new bugs were introduced in the copies. It's understandable that this would be the fate of the loader. When porting to a new platform, it's easy just to copy code from a working loader and tweak it a little for the new environment. Little thought was given to the long-term effects of lack of refactoring. Once the loader could boot a kernel, why spend more time on the loader? Turns out that this strategy and these attitudes were harmful. The filename parsing code, for example, had been copied from environment to environment so that there were about 10 copies when I started. A common routine was needed for them all—well, all but one, which had a legitimate reason for differing since its device specification varied from the usual "diskXpY:" used elsewhere. Bugs in parsing led me to refactor all this code into one location (so I could fix bugs once). This allowed the novel use of "/dev/XXXX" to work for the "device name" when accessing raw devices. It also lets "host" prefix work without needing a unit number. The loader's filename parser is way more flexible than I'd ever known.

Conclusion

Adapting the FreeBSD bootloader and kernel to boot in this new environment proved to be straightforward. The large number of small tasks for Linux host integration, coupled with FreeBSD's undocumented bootloader-to-kernel handoff, provided the biggest challenges for this phase of the project. The unexpected hardware erratum added to the excitement and caused the largest change needed to the kernel. With FreeBSD/aarch64 successfully booting on real hardware, we can proceed to the next phase of the project: creating our own firmware and finding the remaining FreeBSD/amd64 bugs. Even with these limitations, we have used loader.kboot to download an installer ramdisk to provision a system and reboot the result. It has also been incorporated into last summer's loader continuous integration GSoC project.

Next Up

In the next article, we'll create a LinuxBoot firmware image that boots FreeBSD. I'll explain how firmware images are packaged, the tools used to create and manipulate them, and how to reflash your firmware images if you are brave enough. I'll help you select the right tools from the many options Linux provides for creating an initrd, and give a sample script to find and boot your FreeBSD system. With luck, you'll have a simpler, faster, more secure firmware by the end.

WARNER LOSH has been contributing to the FreeBSD project for many years. He's contributed many features and fixes to the bootloader. His interest in Unix history extends to how bootstrapping evolved alongside Unix and its many derivatives. He lives in Colorado with his wife Lindy, who loves drawing cats and dogs, and his daughter, who plays an assortment of brass instruments. He can often be found walking dachshunds.

FreeBSD Container Images

BY DOUG RABSON

OCI container engines such as containerd or podman need images. A container image is a read-only directory tree which typically contains an application with supporting files and libraries. Running this image on a container engine makes a writable clone of the image and executes the application in some kind of isolation environment such as a jail.

Images are distributed via registries which store the image data and provide a simple REST API to access images and their metadata. The registry APIs, image formats and metadata are standardised by the Open Container Initiative which largely replaces earlier docker formats.

Running this image on a container engine makes a writable clone of the image and executes the application in some kind of isolation environment such as a jail.

OCI Images

Images are represented as a sequence of layers, each of which is stored as a compressed tar file. To unpack the image, we start with an empty directory and then unpack each layer in sequence, allowing later layers to add files or change files from an earlier layer. Typically, the result of this process is cached by the container engine.

In addition to the layer data, two additional metadata objects are used. The manifest lists the layers and can contain annotations to describe the image. The image config describes the target operating system and architecture and allows a default command to be used for running the image.

All of this is stored in a 'content addressable' structure where the hash of a component is used to name it. For example, a small base image I use for statically linked applications looks like this:

```
$ ls -lR
total 6
drwxr-xr-x  3 root  dfr    3 Sep  8 10:36 blobs
-rw-r--r--  1 root  dfr  275 Sep  8 10:36 index.json
-rw-r--r--  1 root  dfr   31 Sep  8 10:36 oci-layout

./blobs:
total 25
drwxr-xr-x  2 root  dfr  6 Sep  8 10:36 sha256

./blobs/sha256:
total 950
-rw-r--r--  1 root  dfr    1143 Sep  8 10:36
190e4f8bf39f4cc03bf0f723607e58ac40e916a1c15bd212486b6bb0a8c30676
-rw-r--r--  1 root  dfr     496 Sep  8 10:36
5657eb844c0c0142aa262395125099ae065e791157eaa1e1d9f5516531f4fe30
-rw-r--r--  1 root  dfr   34916 Sep  8 10:36
5af368a2a6078dc912135caed94a6375229a5a952355f5fea60dad1daf516f78
-rw-r--r--  1 root  dfr  911102 Sep  8 10:36
fdb4ee0a131a70df2aae5c022b677c5afbacb5ec19aa24480f9b9f5e8f30fd18
```

All the metadata files in this bundle are in json format as described here. The top-level index.json file links to the manifest using its hash:

```
$ cat index.json | jq
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest": "sha256:190e4f8bf39f4cc03bf0f723607e58ac40e916a1c15bd212486b6bb0a8c30676",
      …
    }
  ]
}
```

This manifest describes two data layers, one with just the FreeBSD standard directory structure and one containing minimal support files such as /etc/passwd and ssl certificates. It also links to the config which has the target operating system and architecture.

Using a content-addressable format like this makes it easier to share storage space and reduce the amount of data downloaded when using multiple images derived from the same base.

The OCI image specification also allows for multi-architecture images which are just lists of manifests:

```
{
    "schemaVersion": 2,
    "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
    "manifests": [
        {
            "mediaType": "application/vnd.oci.image.manifest.v1+json",
            "size": 1116,
            "digest":
"sha256:598b927b8ddc9155e6d64f88ef9f9d657067a5204d3d480a1b1484da154e7c4",
            "platform": {
                "architecture": "amd64",
                "os": "freebsd"
            }
        },
        {
            "mediaType": "application/vnd.oci.image.manifest.v1+json",
            "size": 1118,
            "digest":
"sha256:ac732db0f4788d5282a8d16fefbea360d937049749c83891367abd02801b582",
            "platform": {
                "architecture": "arm64",
                "os": "freebsd"
            }
        }
    ]
}
```


FreeBSD Base Images

To make it easier to work with containers on FreeBSD, there is a need for suitable base images. The traditional FreeBSD release process generates a small number of packages intended for installing a fully featured FreeBSD OS on a physical or virtual host. We could use the base.txz package to build our base image but this results in a gigabyte sized image, more than 90% of which is not needed by most applications. Most Linux distributions offer much smaller base images — the official Ubuntu image, for instance, is about 80MB.

Most Linux distributions offer much smaller base images — the official Ubuntu image, for instance, is about 80MB.

Fortunately, the pkgbase project has been working to make a fine-grained package set which subdivides the traditional base.txz tarball into hundreds of much smaller packages. Currently, this consists of many packages for individual libraries and utilities along with two larger packages, FreeBSD-runtime which contains the shell along with a selection of core utilities and FreeBSD-utilities which has a larger set of commonly used utilities.

Early on, I created a "minimal" image using pkgbase which included FreeBSD-runtime, plus SSL certificates and pkg. This is about 80MB and contains enough functionality for sim-

ple shell scripts as well as the ability to install packages. This compares favourably with similar Linux images although it doesn't come close to the busybox-based alpine image which is just 7.5MB.

Since then, I made a small family of images, partly inspired by the distroless project:

- "static" which contains just SSL certificates and timezone data. This can be used as a basis for statically linked applications.
- "base" which extends "static" by adding a selection of shared libraries to support a wide variety of dynamically linked applications.
- "minimal" which adds the FreeBSD-runtime package and package management as before
- "small" which adds FreeBSD-utilities for broader support of shell-based applications.

To support a variety of FreeBSD versions, I embed the version into the image name, e.g., "freebsd13.2-minimal:latest" includes packages from the most recent version of the releng/13.2 branch while "freebsd13-minimal:latest" is built from stable/13. I build all these images with support for amd64 and arm64 architectures and the container engine will automatically select the correct image from the manifest list.

Security

It is important that container images can be verified that they have a trusted origin and have not been tampered with while they are being transferred to the container engine.

An image's manifest typically contains the SHA256 hashes of the image's data layers as well as the hash of the corresponding image config. This means that the hash of the manifest can be used to uniquely identify the image. This can be used to verify the image, e.g., by listing trusted image hashes in a trustable location.

Alternatively, the hash can be used to create a signature which can prove that the image is trusted by the owner of some public key.

Alternatively, the hash can be used to create a signature which can prove that the image is trusted by the owner of some public key. Two common mechanisms are in use for this — the sigstore facility used by podman uses PGP to create an image signature and provides a mechanism to associate a set of images with a signature store which can either be a local directory or a trusted website. This can be used when an image is pulled to verify that it matches the signature. An alternative to sigstore is cosign which stores the signatures alongside the images in the image repository.

Limitations and Future Work

While these images are useful, they contain a fairly arbitrary choice of which packages are installed. Initially, I included support for the alpha.pkgbase.live package repository which simplified extending an image by installing extra packages. Unfortunately, this project lost its funding and for a while, there was no publicly available pkgbase repository. Thankfully, this has been resolved with pkgbase packages available from the standard FreeBSD package repository.

The current mechanism for building images uses pkg to install pkgbase packages into image layers. This is convenient and keeps a record of what was installed into the image. Unfortunately, the pkg metadata is stored in a sqlite database and this does not support

reproducible builds. The sqlite database includes the timestamp a package was installed — this can be overridden to some suitable constant time but even then, the sqlite database is not reproducible.

A larger issue is credibility — I host these images in my own personal repositories at docker.io and quay.io but from the perspective of potential users, there is no reason to trust that the images are trustworthy. Even though I can build images using packages from the FreeBSD package repository these images are not signed or supported by the FreeBSD project.

In my opinion, this is a significant barrier for potential users of FreeBSD container engines and blocks moving these projects from their current 'experimental' state to something which can be considered for production. This has been confirmed with several recent conversations about supporting FreeBSD as a platform for open source projects which build and use images.

Ideally, as well as hosting pkgbase package sets, the FreeBSD project should build FreeBSD container images, either hosting an image registry or making these images available on a public repository such as docker.io. I plan to prototype additions to the release building infrastructure to integrate container image building into the existing pkgbase framework which may help to move this forward.

DOUG RABSON is a Software Engineer with more than thirty years of experience ranging from 8-bit text adventure games back in the 1980s to tera-byte-per-second distributed long aggregation systems in the 2020s. He has been a FreeBSD project member and committer since 1994 and is currently working on improving FreeBSD support for modern container orchestration systems such as podman and kubernetes.

Kick Me Now with Webhooks

BY DAVE COTTLEHUBER

What Is a Webhook And Why Would I Want One?

A webhook is an event-driven remote callback protocol over HTTP allowing scripts and tasks to be trivially invoked from almost any programming language or tool.

What's great about webhooks is their prevalence and their simplicity. With a simple HTTP URL, you can request a remote server to dim the lights, deploy code, or run an arbitrary command on your behalf.

The simplest of webhooks can just be a bookmarked link in a smart phone web browser, or a more complicated version might require strong authentication and authorization.

While larger automation toolsets exist, such as Ansible and Puppet, sometimes something simpler is sufficient. Webhooks are just such a thing, allowing you to run a task on a remote computer, on request, securely. Invoking a webhook can be considered "kicking," hence the article title.

With a simple HTTP URL, you can request a remote server to dim the lights, deploy code, or run an arbitrary command on your behalf.

Integrations

There are no official standards yet, however commonly, webhooks are sent via POST with a JSON object body, using TLS encryption, often secured with signatures against tampering, network forgery, and replay attacks.

Common integrations include Chat services such as Mattermost, Slack, and IRC, software forges like Github and Gitlab, generic hosted services like Zapier or IFTT, and many home automation suites like Home Assistant as well. The sky's the limit, as webhooks are sent and received almost everywhere.

While you could write a minimal webhook client or server in an hour, there are many options already available in almost every programming language today. Chat software often provides inbuilt webhook triggers that can be invoked by users, often by a `/command` style syntax. IRC servers are not forgotten either, mostly via daemons or plugins.

One less obvious advantage for webhooks is the ability to demarcate security and privileges. A low-privileged user can call a webhook on a remote system. The remote webhook servce can run also at a low privilege, managing validation and basic syntax checking, before

invoking a higher privilege task once the requirements have been verified. Perhaps the final task has access to a privileged token that allows rebooting a service, deploying new code, or letting the kids have another hour of screen time.

Common software forges such as GitHub, GitLab, and self hosted options also provide these such that they can be triggered including the name of the branch, the commit, and the user who made the change.

This allows, with relative ease, constructing tools that update sites, reboot systems, or trigger more complicated toolchains as required.

The Architecture

The typical arrangement comprises a server listening for incoming requests and a client that submits requests along with some parameters, possibly including some authentication and authorization.

The Server

First up, let's discuss the server side. Typically, this will be a daemon listening for an HTTP request that matches certain conditions for it to be processed. If these conditions are not met, the request is rejected with the appropriate HTTP status codes, and for a successful submission, parameters can be extracted from the approved request and then custom actions are invoked as required.

The Client

As the server uses HTTP, almost any client can be used. cURL is a very popular choice, but we will use a slightly more pleasant one called gurl which has built-in support for HMAC signatures.

The Message

The message is typically a JSON object. For those who care about replay or timing attacks, you should include a timestamp in the body, and validate it before further processing. If your webhook toolkit can sign and validate specific headers, that's an option also, but most don't.

The Security

The body of the HTTP request can be signed with a shared secret key, and this resulting signature then provided as a message header. This provides both a means of authentication and also proof that the request has not been altered in transit. It relies on a shared key to enable both ends to verify the message signature independently using the additional HTTP header with the body of the message.

The most common signature method is `HMAC-SHA256`. This is a combination of two cryptographic algorithms — the familiar `SHA256` hash algorithm that gives a secure digest

The body of the HTTP request can be signed with a shared secret key, and this resulting signature then provided as a message header.

of a larger message, in our case the HTTP body, and the HMAC method, which takes a secret key and mixes it with a message to produce a unique code, a digital signature, if you like.

These functions are combined to produce a high-integrity check if the message has been tampered with. It's like a digital seal over the contents and confirms that the message must have been sent from a party that knows the shared secret.

Note that using both TLS encryption and a signature provides both confidentiality and integrity of the enclosed message, but not availability. A well-positioned attacker could interrupt or flood the intervening network and messages would be lost without notification.

Common practice is to include a timestamp in the body of the webhook, and as this is covered by the HMAC signature, timing and replay attacks can be mitigated.

Note that a non-timestamped body will always have the same signature. This can be useful. For example, this allows pre-calculation of the HMAC signature, and using an unchanging HTTP request to trigger remote actions, without needing to make the HMAC secret available on the system issuing the webhook request.

Putting it Together

We'll install a few packages to help, a webhook server, the well-known tool curl, and finally gurl, a tool that makes signing webhooks easy.

```
$ sudo pkg install -r FreeBSD www/webhook ftp/curl www/gurl
```

Let's get our server up and running, with this minimal example, save it as **`webhooks.yaml`**. It will use the **`logger(1)`** command to write a short entry into **`/var/log/messages`** with the HTTP **`User-Agent`** header of the successful webhook.

Note that there is a **`trigger-rule`** key that ensures the HTTP query parameter, **`secret`**, matches the word **`squirrel`**.

Currently we have no TLS security and no HMAC signature either, so this is not a very secure system yet.

```
---
- id: logger
  execute-command: /usr/bin/logger
  pass-arguments-to-command:
  - source: string
    name: '-t'
  - source: string
    name: 'webhook'
  - source: string
    name: 'invoked with HTTP User Agent:'
  - source: header
    name: 'user-agent'
  response-message: |
    webhook executed
  trigger-rule-mismatch-http-response-code: 400
  trigger-rule:
    match:
      type: value
```

```
    value: squirrel
    parameter:
      source: url
      name: secret
```

And run **`webhook -debug -hotreload -hooks webhook.yaml`** in a terminal. The flags used should be self-explanatory.

In another terminal, run **`tail -qF /var/log/messages | grep webhook`** so that we can see the results in real time.

Finally, let's kick the webhook using curl, first without the query parameter, and then again, with it:

```
$ curl -4v 'http://localhost:9000/hooks/logger'
*   Trying 127.0.0.1:9000...
* Connected to localhost (127.0.0.1) port 9000
> GET /hooks/logger HTTP/1.1
> Host: localhost:9000
> User-Agent: curl/8.3.0
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< Date: Fri, 20 Oct 2023 12:50:35 GMT|
< Content-Length: 30
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
Hook rules were not satisfied.
```

Note how the failed request is rejected using the HTTP status specified in the **`webhooks.yaml`** config file and the returned HTTP body explains why.

Providing the required query and secret paramter:

```
$ curl -4v 'http://localhost:9000/hooks/logger?secret=squirrel'
*   Trying 127.0.0.1:9000...
* Connected to localhost (127.0.0.1) port 9000
> GET /hooks/logger?secret=squirrel HTTP/1.1
> Host: localhost:9000
> User-Agent: curl/8.3.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 20 Oct 2023 12:50:39 GMT
< Content-Length: 17
< Content-Type: text/plain; charset=utf-8
<
webhook executed
* Connection #0 to host localhost left intact
```

The hook is executed and we can see the result in syslog output.

```
Oct 20 12:50:39 akai webhook[67758]: invoked with HTTP User Agent: curl/8.3.0
```

Using HMACs to Secure Webhooks

The HMAC signature described earlier, when applied over the HTTP body and sent as a signature, is tamper-proof, providing authentication and integrity, but only of the body, not of headers. Let's implement that. Our first step is to generate a short secret and modify **webhook.yaml** to require verification.

```
$ export HMAC_SECRET=$(head /dev/random |  sha256)
```

We'll use a more memorable secret of **n0decaf** one for this article, but you should use a nice strong one.

Replace the **webhook.yml** file with this one, which will extract two JSON values from the payload (which is signed, and therefore trusted), and pass them to our command for execution.

```
---
- id: echo
  execute-command: /bin/echo
  include-command-output-in-response: true
  trigger-rule-mismatch-http-response-code: 400
  trigger-rule:
    and:
    # ensures payload is secure -- headers are not trusted
    - match:
        type: payload-hmac-sha256
        secret: n0decaf
        parameter:
          source: header
          name: x-hmac-sig
  pass-arguments-to-command:
  - source: 'payload'
    name: 'os'
  - source: 'payload'
    name: 'town'
```

And use **openssl dgst** to calculate the signature over the body:

```
$ echo -n '{"os":"freebsd","town":"vienna"}' \
    | openssl dgst -sha256 -hmac n0decaf
SHA2-256(stdin)= f8cb13e906bcb2592a13f5d4b80d521a894e0f422a9e697bc68bc34554394032
```

With the body and the signature, now let's make the first signed request:

```
$ curl -v http://localhost:9000/hooks/echo \
    --json '{"os":"freebsd","town":"vienna"}' \
    -Hx-hmac-sig:sha256=f8cb13e906bcb2592a13f5d4b80d521a894e0f422a9e697bc68bc34554394032

*   Trying [::1]:9000...
* Connected to localhost (::1) port 9000
> POST /hooks/echo HTTP/1.1
> Host: localhost:9000
> User-Agent: curl/8.3.0
> x-hmac-sig:sha256=f8cb13e906bcb2592a13f5d4b80d521a894e0f422a9e697bc68bc34554394032
> Content-Type: application/json
> Accept: application/json
> Content-Length: 32
>
< HTTP/1.1 200 OK
< Date: Sat, 21 Oct 2023 00:41:57 GMT
< Content-Length: 15
< Content-Type: text/plain; charset=utf-8
<
freebsd vienna
* Connection #0 to host localhost left intact
```

On the server side with **-debug** mode running:

```
[webhook] 2023/10/21 00:41:57 [9d5040] incoming HTTP POST request from [::1]:11747
[webhook] 2023/10/21 00:41:57 [9d5040] echo got matched
[webhook] 2023/10/21 00:41:57 [9d5040] echo hook triggered successfully
[webhook] 2023/10/21 00:41:57 [9d5040] executing /bin/echo (/bin/echo) with arguments ["/
bin/echo" "freebsd" "vienna"] and environment [] using  as cwd
[webhook] 2023/10/21 00:41:57 [9d5040] command output: freebsd vienna

[webhook] 2023/10/21 00:41:57 [9d5040] finished handling echo
< [9d5040] 0
< [9d5040]
< [9d5040] freebsd vienna
[webhook] 2023/10/21 00:41:57 [9d5040] 200 | 15 B | 1.277959ms | localhost:9000 | POST /
hooks/echo
```

Separately calculating the signature each time is error-prone. gurl is a fork of an earlier project and adds automatic HMAC generation as well as some niceties around handling and processing JSON.

The signature type, and the signature header name are prepended to the secret and joined by **:**. This is exported as an environment variable so that its not directly visible in shell history.

```
$ export HMAC_SECRET=sha256:x-hmac-sig:n0decaf
$ gurl -json=true -hmac HMAC_SECRET \
  POST http://localhost:9000/hooks/echo \
  os=freebsd town=otutahi

POST /hooks/echo HTTP/1.1
Host: localhost:9000
Accept: application/json
Accept-Encoding: gzip, deflate
Content-Type: application/json
User-Agent: gurl/0.2.3
X-Hmac-Sig: sha256=f634363faff03deed8fbcef8b10952592d43c8abbb6b4a540ef16af0acaff172

{"os":"freebsd","town":"otutahi"}
```

As we can see above, the signature is generated for us, and adding JSON key=value pairs is straightforward without needing quoting and escaping.

Back comes the response, pretty-printed for us: the HMAC has been verified by the server, the values of the two keys extracted and passed as parameters to our **`echo`** command, and the results captured and returned in the HTTP response body.

```
HTTP/1.1 200 OK
Date : Sat, 21 Oct 2023 00:50:25 GMT
Content-Length : 16
Content-Type : text/plain; charset=utf-8

freebsd otutahi
```

More complex examples are provided in the port's sample webhook.yaml or the extensive documentation.

Securing Webhook Contents

While using HMACs prevents tampering with the message body, it's still visible in plain text to those dastardly hackers.

Let's add some transport-layer security, using a self-signed TLS key and certificate, for the webhooks server on **`localhost`** and relaunch the webhook server:

```
$ openssl req -newkey rsa:2048 -keyout hooks.key \
  -x509 -days 365 -nodes -subj '/CN=localhost' -out hooks.crt

$ webhook -debug -hotreload \
  -secure -cert hooks.crt -key hooks.key \
  -hooks webhook.yaml
```

The **`curl`** command will need an additional **`-k`** parameter to ignore our self-signed certificate, but otherwise things proceed as before:

```
curl -4vk 'https://localhost:9000/hooks/logger?secret=squirrel
'
*   Trying 127.0.0.1:9000...
* Connected to localhost (127.0.0.1) port 9000
* ALPN: curl offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_128_GCM_SHA256
* ALPN: server accepted http/1.1
* Server certificate:
*  subject: CN=localhost
*  start date: Oct 20 13:05:09 2023 GMT
*  expire date: Oct 19 13:05:09 2024 GMT
*  issuer: CN=localhost
*  SSL certificate verify result: self-signed certificate (18), continuing anyway.
* using HTTP/1.1
> GET /hooks/logger?secret=squirrel HTTP/1.1
> Host: localhost:9000
> User-Agent: curl/8.3.0
> Accept: */*
>
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
< HTTP/1.1 200 OK
< Date: Fri, 20 Oct 2023 13:12:07 GMT
< Content-Length: 17
< Content-Type: text/plain; charset=utf-8
<
webhook executed
* Connection #0 to host localhost left intact
```

gurl has no such option and expects you to do things properly. For production usage, it is much better to use a reverse proxy such as nginx or haproxy to provide robust TLS termination, and allows using public TLS certificates, via Let's Encrypt and similar services.

Updating a Website with Github and Webhooks

For this to work successfully, you'll need both to have your own domain, and a small server or virtual machine to host the daemon on.

While this article can't cover the full details for setting up your own website, TLS encryption certificates, and DNS, the steps below will largely be similar for any software forge.

You will need to setup a proxy server, such as Caddy, nginx, haproxy, or similar, with working TLS. A great choice is to use the ACME protocol, via Let's Encrypt, to maintain this for you.

You'll need to adjust your proxy server to route appropriate requests to the webhook daemon. Consider restricting IP addresses that can access it, and restricting HTTP methods, as well. Github's API has a **`/meta`** endpoint to retrieve their IP addresses, but you'll need to keep this up to date, however.

Enable the webhook service as follows, using the same options we used before, and start your daemon via **`sudo service webhook start`**

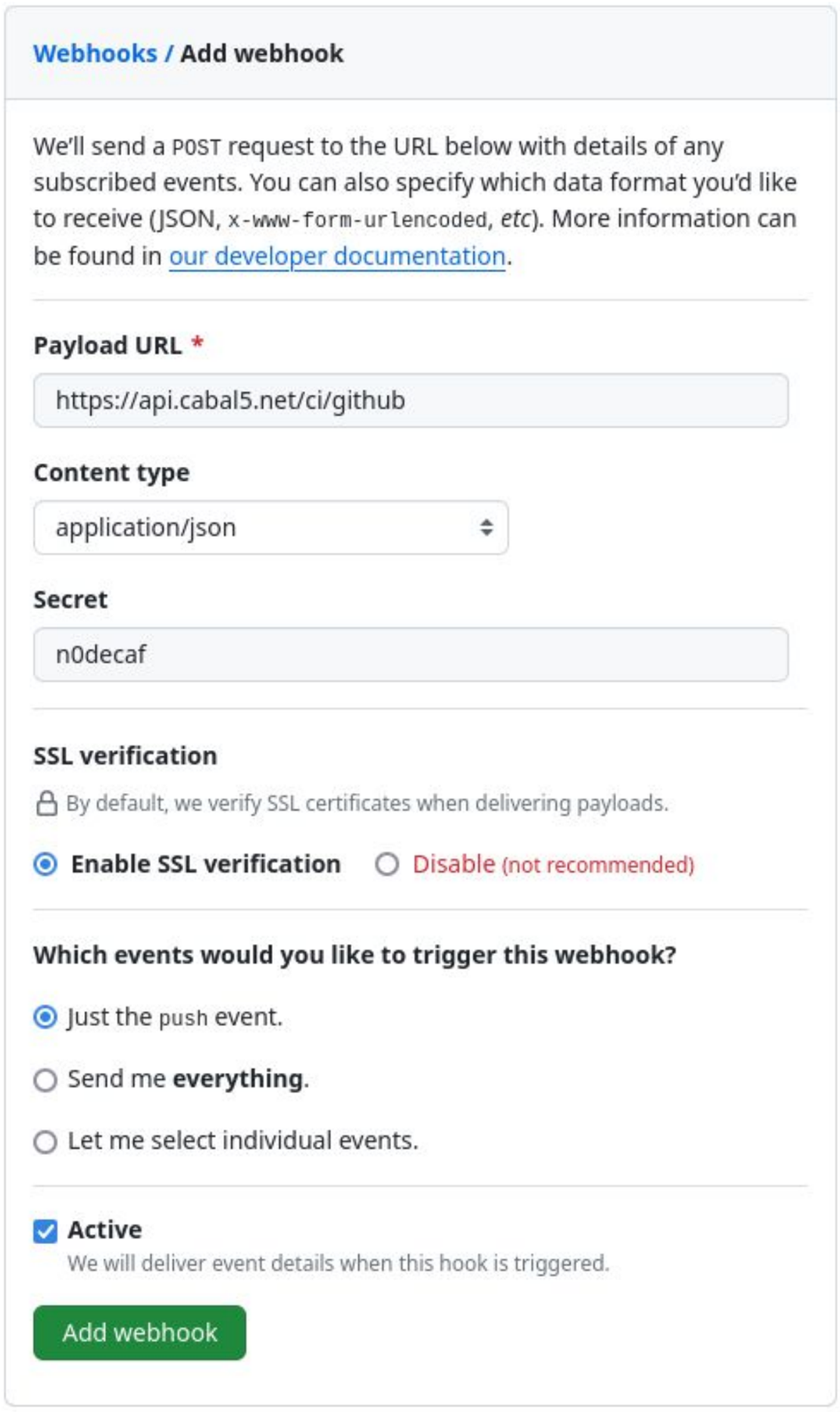
```
# /etc/rc.conf.d/webhook
webhook_enable=YES
webhook_facility=daemon
webhook_user=www
webhook_conf=/usr/local/etc/webhook/webhooks.yml
webhook_options=" \
 -verbose \
 -hotreload \
 -nopanic \
-ip 127.0.0.1 \
-http-methods POST \
-port 1999 \
-logfile /var/log/webhooks.log \
"
```

You'll need to validate from outside, that the URL and webhook daemon are accessible.

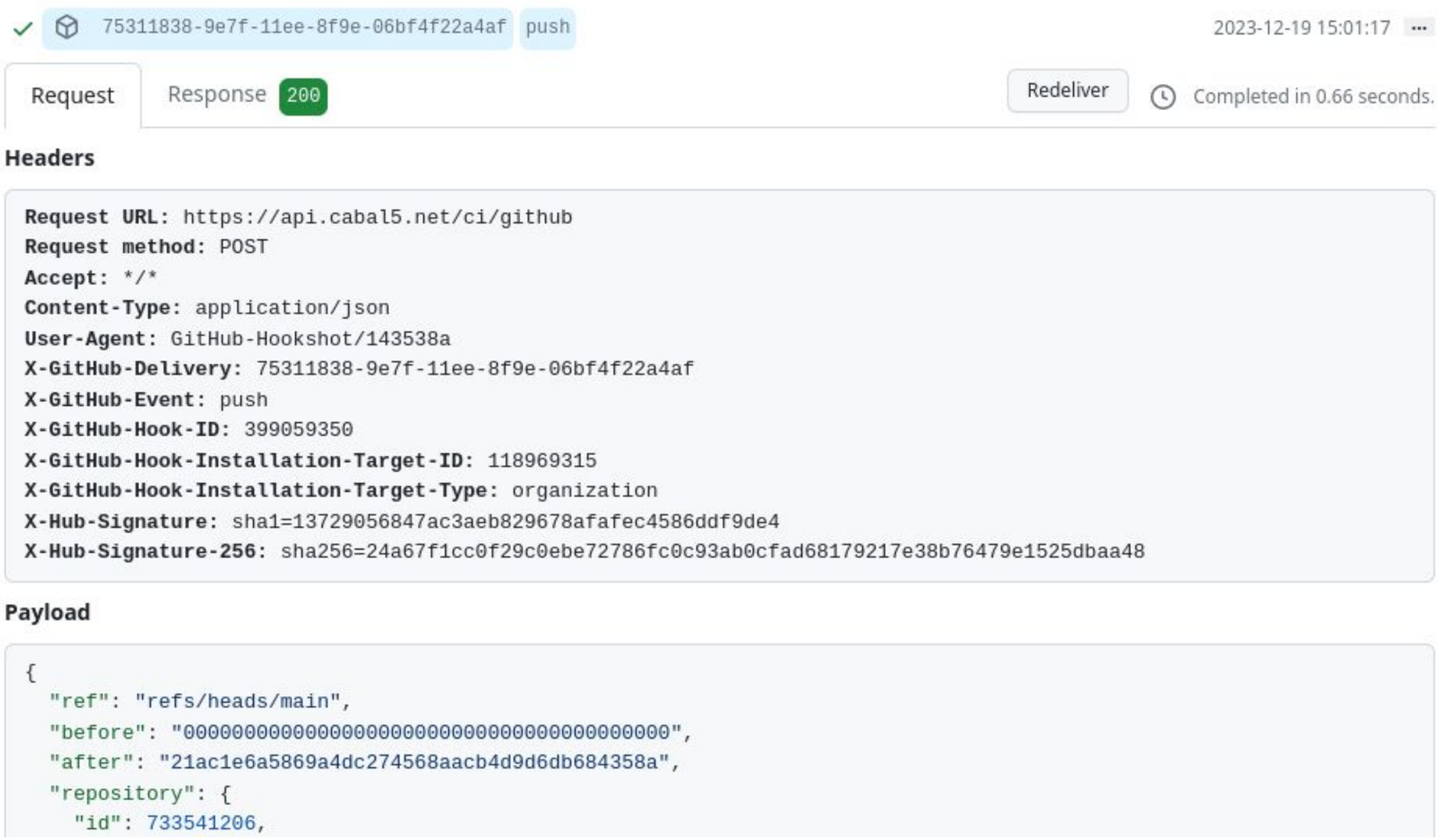
On your software forge side, create a new JSON format webhook, with the shared HMAC secret, to be invoked on every push to your repository.

For example, using Github you must provide a:

- payload URL, pointing to the external URL for your proxied internal webhook daemon
- content-type **`application/json`**
- the shared secret, such as **`n0decaf`** in the examples



Once the webhook has been created on the GitHub side, you should be able to confirm that a successful event was received, and then on your next push of code, you can check on the Github website for the request that GitHub sent, and the response your daemon provided.



✓ 75311838-9e7f-11ee-8f9e-06bf4f22a4af push 2023-12-19 15:01:17 ···

Request | Response 200 | Redeliver | Completed in 0.66 seconds.

Headers

```
Cache-Control: no-transform
Content-Length: 0
Date: Tue, 19 Dec 2023 15:01:17 GMT
Permissions-Policy: interest-cohort=()
Referrer-Policy: strict-origin-when-cross-origin
Strict-Transport-Security: max-age=15552000
X-Content-Type-Options: nosniff
X-Frame-Options: sameorigin
X-Frontend: f01
X-Powered-By: FreeBSD
X-Sni: api.cabal5.net
X-Ua-Compatible: IE=Edge
X-Xss-Protection: 1;mode=block
```

Handling Secrets for Webhook Scripts

Often, you will need to run some script that requires access to various secrets. There are many possibilities, but one simple one is to add a **`webhook_env_file`** path entry to the **`rc.conf`** settings. This file is a shell-quoted list of key=value parameters, that are included into the environment of the webhook daemon, by the FreeBSD rc.d system, and thus available to any and all webhook scripts that are invoked.

```
# /usr/local/etc/webhook/environment
CI=true
CI_SECRET="b@dw0lf"
```

Add **`webhook_env_file="/usr/local/etc/webhook/environment"`** to your rc.conf settings, and restart the daemon, to make these available to subsequent webhook invocations.

Running Scripts

A simple website update script could be as small as:

```
#!/bin/sh -eu
set -o pipefail
cd /var/www/my-awesome-website
git reset --hard
git clean -fdx
git pull --ff-only --tags
```

Or as complicated as your imagination, using validated parameters extracted from the JSON body of the signed webhook request.

Let me know what interesting webhook kicks you invent!

DAVE COTTLEHUBER has spent the last 2 decades trying to stay at least 1 step ahead of The Bad Actors on the internet, starting off with OpenBSD 2.8, and the last 9 years with FreeBSD since 9.3, where he has a ports commit bit, and a prediliction for using jails, and obscure functional programming languages that align with his enjoyment of distributed systems, and power tools with very sharp edges.

- Professional Yak Herder, shaving BSD-coloured yaks since ~ 2000
- FreeBSD ports@ committer
- Ansible DevOops master
- Elixir developer
- Building distributed systems with RabbitMQ and Apache CouchDB
- Enjoys telemark skiing, and playing celtic folk music on a variety of instruments

Hackathon

Oslo in October

BY TOM JONES

Hackathon's are a small-scale event where hackers (or developers) get together and conduct a marathon hacking session. Many parts of a hackathon track a marathon, a dedicated group come together and at an arranged time, they act separately, but towards a common goal. Like a foot marathon, a hackathon has lots of snack breaks, and at the end we all join for a party.

Unlike a foot race we are focused on our computers. Eirik from Modirum joined us for a hackathon in Aberdeen in 2022 and so enjoyed the idea that he demanded to host a hackathon from Modirum's offices in Oslo. It took some arranging, and I missed the April goal by about 6 months. We met in October 2023 in Oslo, Norway to have a ports focused hackathon.

This is the first focused hackathon I have run. In the past, people have always asked me "what is the theme?" I gave in this time and agreed that we would focus on Ports and Infrastructure. I was a little sneakily keeping a wide enough topic that anyone could come and join us and still be on theme if asked.

The call went out for FreeBSD hackers interested in three days working on ports from the beautiful city of Oslo.

Modirum kindly hosted us from their offices that were well situated to be easy to reach for the hackathon with lots of places nearby for the of the most important meals. The offices are are a mixture between office and hackerspace with a collection of fascinating toys for Operating System nerds. A first for me was the bookcase filled with 286 and 386 motherboards. Eirik's collection has a home in Modirum's offices and featured running 386s on which we watched old demos one evening after the pub, and a Mac SE connected to the internet via some ESP powered WiFi magic.

The call went out for FreeBSD hackers interested in three days working on ports from the beautiful city of Oslo.

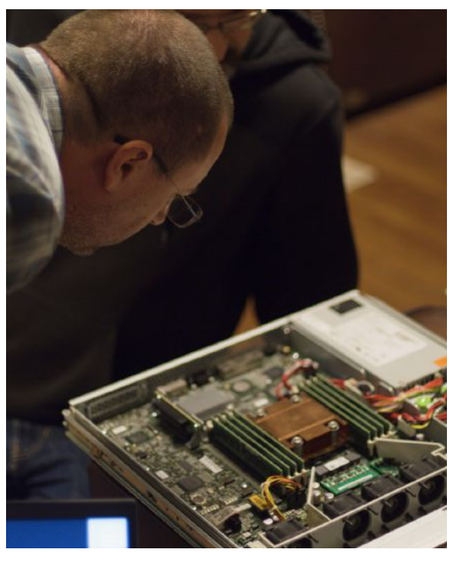
In total we were 6 FreeBSD project members, two from outside FreeBSD, and Eirik and varying members of his team. This was a great size to keep things fluid, but small enough to make going out for dinner easy.

Hackathons are better when they are smaller, fewer attendees than a DevSummit. It is great to get a lot of people together, but as the size of the group grows, conversations amplify, and the balance between rare insights and focused quiet time to do

work shifts.

Beyond the FreeBSD project members that joined us, we also were joined by Trenton (who wrote his own report that follows this one) and Harold from the NetBSD project. Adding some more hackers outside of FreeBSD developers allowed our conversations to cover ground which might sometimes be unspoken by those who have been in the project for a long time. Sometimes you have been involved too long to ask 'stupid' questions, but new people have no such built-in fear.

Harold was quickly given an old Sun server from Eirik's vintage computing collection to play with - NetBSD does claim to run everywhere after all - but this one will need some work to get going.



The hackathon ran over three days, Wednesday to Friday, and some of us also stayed in Oslo through the weekend and enjoyed an outing to Oslo's outdoor building Museum accessed by an excellent ferry across the harbor.

Hackathons are great small events, and as the organizer of this one, I have to admit that I skimped on the organization this time. Rather than see that as a bad thing, I want you to take away that all a hackathon needs is a room and some hackers. I still dream of there being a FreeBSD development meetup somewhere in the world every month of the year. If you would like that, too, you can help by organizing one.

I'm happy to help remotely, for advice you can email me thj@freebsd.org and I'll share the few secrets to running a successful hackathon that I know. And now, here's Trenton Schulz's take on the event.

Oslo in October Too

BY TRENTON SCHULZ

During the summer, I was going through Vermaden's valuable news looking for interesting tidbits of FreeBSD-related news. Then, I saw "FreeBSD Hackathon Oslo." Here it was, a FreeBSD hackathon right in my backyard. I should try to attend this.

Looking at the wiki page, I saw that it was primarily targeted at FreeBSD developers (people I associated with a @freebsd.org email address), but I did maintain some ports and I had a couple of other ideas that I could work on, and I might be able to get some guidance on those projects too. I could at least ask and see if I could be a guest.

So, I put the dates on my calendar, and then I made a to-do item to "consider the FreeBSD hackathon." I would revisit this item in my to-do list and think that, "yes, I should make a final decision." But it wasn't until the weekend before the hackathon when I tuned into the BSD Now podcast and heard Benedict and Tom talk about how the hackathon

would focus on ports infrastructure that I made up my mind that "yes, why haven't I asked yet?" I double-checked my calendar and saw that I needed to lecture and lead meetings for some of the hackathon days, but Friday was open. Yes! I emailed Tom Jones and he was very welcoming. I put up a vacation day, a vacation hacking on FreeBSD.

On the Friday, I arrived at the nice offices of Modirum, the host for the hackathon, in Youngstorget in Oslo. Olivier let me in and after greetings were exchanged with all the other participants, I got on with hacking. For me, this was mostly updating ports that I maintain. This included the Jotta CLI port (net/jotta-cli), which I use as a tertiary backup solution at home. I tested it out locally, and it worked well, but forgot to test the i386 version and put the wrong checksum in. But after a review from Olivier, it was committed. One port updated.

Then, I saw that there was an upstream update for JuliaMono (x11-fonts/juliamono); the monospace font that I use. So, I went through the steps and updated the port. That was also committed after lunch. Even though these were low-hanging fruit, it felt good to get them committed and out for others as well. Then I decided to scratch some more challenging itches.

I've recently started using Beeper, a unified messaging client. They have a Linux client, so I thought I would try to see if I could run it on FreeBSD and build it into a port. The Linux client is an appimage, so I unpacked it and tried to run it. Unfortunately, the executable had problems resolving libraries inside the unpacked appimage. I tried to trace, but it seemed to find the actual library, so something more was happening. Realizing that this might be a bit more in-depth problem than I could solve in a couple of hours, I put it aside to look at later. I have since found out that there is a web interface to Beeper and one can use a Matrix client, so creating a port may not be as exciting.

One of the benefits of being at a hackathon is all the spontaneous conversations that show up around the table.

Having put aside Beeper, I looked at a port that I had adopted, audio/logitechmediaserver. I had a bug report where the build had failed, but the port had signaled all was well until it came to packaging when it identified many missing files. This led me to diving deep into the port infrastructure makefiles, where I learned that the port had done a clever thing in redefining a target since the port itself uses perl for building. I then spent some time trying to figure out how to rework the target so that it would throw an error. I didn't finish this before we went out for dinner.

One of the benefits of being at a hackathon is all the spontaneous conversations that show up around the table. One gets a nice collection of history, news, funny stories, technical information, advice, and even some gossip. Not just about FreeBSD internals, but on a variety of subjects including electronics, travel, Norway, and world events. You are soaked in all this information as you look through source code. A lot of camaraderie naturally shows up during the hackathon.

One can also get great advice and wisdom in a quick side discussion. For example, I had a quick discussion with Olivier about my logitechmediaserver conundrums which yielded additional philosophy about testing and maintaining ports that I found very helpful: given that one is the maintainer of a port, you are likely to encounter the biggest build issues first. I need to weigh the advantages and disadvantages of how easy it needs to be to diagnose port issues for others versus the extra work to flag these corner cases. The 3-minute conversation gave me a lot to think about.

Overall, even though I was only at the hackathon for one day, I found it wonderful to focus only on solving FreeBSD problems for many hours. I was also surprised how quick the time went and was further surprised when I had to catch my bus home. There were so many additional things I wanted to work on. Thank you to everyone at the hackathon for being so welcoming and helpful and to Modirum for being excellent hosts!

The whole experience made me realize that attending a hackathon was helpful even if I was not an "official" FreeBSD developer. If you are a person who is hacking on FreeBSD and you see a FreeBSD hackathon in your area, reach out those attending and find out if you can attend. You will have a great experience.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.

TRENTON SCHULZ is a senior research scientist at the Norwegian Computing Center. His research interests include human-computer interaction, human-robot interaction, and universal design of ICT. He is very happy when he can combine his research interests with using FreeBSD.

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

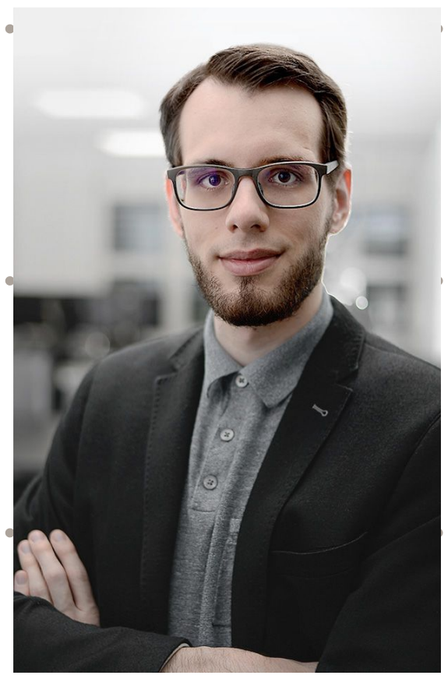
FreeBSD™
FOUNDATION

INTERVIEW

New Ports Committer: Joel Bodenmann (jbo@freebsd.org)

INTERVIEWED BY TOM JONES

TJ: Hi Joel, welcome to the project. Could you give me a little background on yourself and the sort of technology projects you enjoy working on?



JBO: I'm an electronics engineer mainly focusing on embedded systems. Usually I like working with systems that are designed to perform a specific task rather than generic computation.

TJ: FreeBSD isn't famously a great platform for embedded systems development, have you tried to work or do work style projects on FreeBSD?

JBO: I think I have to disagree with your statement. The reason why I got to FreeBSD is exactly to use it as a platform for embedded systems development. Although the world of "embedded" has changed drastically the past few years, the embedded systems I usually work with are comparably low-resource, real-time systems (i.e., your typical microcontroller based systems with < 120MHz CPU, < 128kB RAM). As such, these systems are not designed to run FreeBSD themselves, but they also wouldn't run Linux in any practical sense (nor would the requirements allow for that).

You still need a (desktop) host system for the actual development as well as surrounding support infrastructure tho. These embedded projects I usually work on can take several years from first-meeting to deployed product. One of my mood points with using Linux as a development platform is that the Linux ecosystem is in a constant state of flux. You setup your development workflow and environment on Linux based systems only to be forced to upgrade a few months later which requires to revalidate your entire workflow.

FreeBSD is well known for it's stability and coherence. Rather than implementing a new system or component from scratch because the previous one is not "good enough" anymore, FreeBSD tries to design those systems and tooling to be maintainable and expandable, vastly reducing development system management overhead.

For some projects, I do need a non-microcontroller-based system to which the microcontroller-based systems talk (i.e., for long-term data logging, orchestration etc.). In those scenarios the same benefits of FreeBSD apply: You setup and validate your system once and then just perform smaller maintenance tasks over the years, whereas with Linux-based systems I never knew what I would wake up to the next day.

A crude summary would basically state that I like to spend my time working on the actual development/engineering of the system I am working with rather than updating, debugging and revalidating my development platform just because the underlying init system, audio subsystem, hypervisor or most popular container system or similar changed three times in two years. FreeBSD checks all the boxes here. Since I migrated all of my servers, network

infrastructure and workstations to FreeBSD, I never had to guess whether my development infrastructure still boots and works when I enter the office the next day. It's just rock solid.

TJ: Have you taken on a particular area of focus in the ports tree?

JBO: So far, I have been mainly focused on handling the "low-hanging fruits" with the idea that this allows me to familiarize myself with the basics of the ports system as well as the workflows and infrastructure. At the same time, this frees up resources of the more experienced ports committers so they can spend their efforts on more complex matters.

As I am getting more comfortable with this, I hope to take on more involved tasks in the coming year such as updating ports of libraries with many consumers.

As such, my answer to your question is: No, I try to help wherever I can. I have no doubt that with increased experience I will soon find something larger to work on :)

TJ: The ports collection is massive and a lot of small tools can do heavy lifting. What are some of the low-hanging ports you've worked on so far?

Do you have any suggestions for others to find low-hanging fruit and easier first ports?

JBO: Personally, I considered two scenarios to be low-hanging fruits so far:

1. PRs with patches that already have maintainer approval, where the patch updates an existing port to a new upstream minor version. Here I'd recommend to stay clear of "fundamental" ports with a high number of consuming ports in the beginning to reduce the risk of triggering massive fall-outs.

   My reasoning here is that comparably little can go wrong with simple upstream minor version bumps and maintainers tend to be careful not to break their ports and already have the experience in what to watch out for specifically with regard to their upstream.
2. PRs which introduce a new port. These PRs tend to be "low priority" so there's comparably little pressure to get them landed quickly. Furthermore, I identified these as a great way of learning about the different systems and mechanisms that the ports framework provides that I might not have been in contact with yet.

"So far, I have been mainly focused on handling the "low-hanging fruits" with the idea that this allows me to familiarize myself with the basics of the ports system"

As for what I have worked on so far: I intentionally tried to touch various ports from various domains. I am not focusing on a particular category or type of port. Instead, I try to handle PRs of ports I know rely on something I have not yet worked with. This is a great way of getting comfortable with the various Mk/Uses/* scripts.

TJ: When you look to the future of FreeBSD what do you think the main priorities of the project should be from a porters perspective?

JBO: I think that a big part of what makes FreeBSD such an attractive OS to a variety of users is the fact that we usually follow more old-school principles. The FreeBSD project tends

to take the "slow but steady" approach rather than constantly jumping on the latest hype train re-inventing things constantly. There are, of course, drawbacks to this approach as well. For example, the ports framework lacks some features that might be considered "basic" by modern standards such as sub-packages, suggesting optional installs and similar. But I'd argue that exactly because these things are not rushed, we end up with a much more stable and easier to maintain system.

As such, rather than answering your question with a concrete list of steps or goals that need to be accomplished, my main recommendation is to stay true to this approach. Anything that ends up proving to be necessary will happen eventually, but it will usually do so in a non-forced way allowing for proper design, implementation and testing resulting in efficient use of our limited man power.

There are many bullets that one can dodge by simply not rushing or forcing progress. The expression "slow is smooth; smooth is fast" applies here in my opinion.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.

WeGet letters
by Michael W Lucas

The .0 Release is a Metaphorical Tire Change

BY MICHAEL W LUCAS

The *Journal* received a tsunami of letters this month. Once we composted the complaints about the We Get Letters columnist, that left two. Yes, your complaints are composted. This is a highly responsible publication, so I insist that all derogatory emails are printed for my personal meticulous perusal, edification, and education. I have reserved space on my office wall for mounting the most creative, well-reasoned complaints so that they may remind me to "do better." Only one complaint letter has received that honor, however, and I had to write it myself. You don't know enough about me to insult me effectively or eloquently.

Anyway. The two surviving letters both fretted about the freshly hatched FreeBSD 14. It's a brand-new release that you should have already been running for months in production, because open-source Unix is a community effort and if you touch the software, you catch community and must contribute, except you won't will you, no—you've waited for a .0 release and expect your application stack to work just fine atop it without a shudder or shiver. I won't retread that ground, partly because I previously ranted about it in this very column, but also because you didn't listen to it then, so you certainly won't listen to it now. Fear of a dot-0 release means you misunderstand modern system administration.

This is a highly responsible publication, so I insist that all derogatory emails are printed for my personal meticulous perusal, edification, and education.

System administration in a modern enterprise is like performing an oil change on a vehicle doing a hundred and twenty down the freeway. 120 miles an hour, or kilometers, you might ask? When you're lying on your back on one of those oversized mechanic's skateboards, clenching the oil wrench in your teeth and wishing you'd worn shoes with wheels on the heels so you wouldn't have to work quite so hard holding your legs up, it doesn't matter. Occasionally the driver gets bored with weaving between the desktop users guilty of the unspeakable crime of Using The Road While Obeying The Speed Limit Even Though I'm A CEO, so he sideswipes a pothole just to hear your skull bounce off the transmission housing. Wear a helmet. When the oil change is complete, you get to change the spark plugs and flush the coolant. From below, of course. Raising the hood would impair the driver's vision, and you can't possibly interfere with the corporate mission, whatever *that* is.

The .0 release is a metaphorical tire change, that's all. The trick is to wait until the driver claims there's a stretch of smooth road ahead and to place the jack snugly between your knees.

Doing any of this successfully means understanding your operating system. I don't mean the configuration files. Configurations change. You need to understand what the operating system is doing. That means you need a knowledge of DNS and the shell and virtualization and filesystems and debugging. If you want to truly learn this stuff, go read some of Julia Evans' zines. She knows what she's writing about and can communicate it clearly and simply, unlike certain bloviating tech authors staggering around this joint who confuse worthiness with word count and believe that artsy book covers can compensate for the insipidness beneath said cover. Copying a log message into a search engine cannot replace an understanding of how the software works. You still won't understand the error message, mind you, but the discussions around that error will make sense and that comprehension will guide you into making the problem less agonizing. Yes, less agonizing. In systems administration we don't fix problems, we patch around them. Everything is connected to everything in a churning pot of boiling spaghetti logic, and straightening out one section further tangles other sections. Fortunately, we're well along the way to replacing the operating system with the web browser, a course of action that will unquestionably benefit us all—*us* being system administrators, that is. Web developers will be the new system administrators, and as they're charging fiercely towards achieving "serverless" they won't know who we are other than "the people you must pay for no reason or our app stops working." It's a win all around.

Copying a log message into a search engine cannot replace an understanding of how the software works.

It's not all bad news. Not entirely. Your view of the potholes can be described only as splendid. You will accumulate complex traumas incomprehensible to the passengers, other drivers. This will drive you to develop eccentric coping strategies that render you wholly unsuitable for mainstream society. That might seem like bad news, at least until you meet people. The camaraderie amongst those who exchange pothole stories cannot be exceeded—if you can make yourself interact with them, that is. Plus, you can occasionally tweak something hydraulic to make the driver's seat shoot six inches straight up so the CEO bonks his head. "It's a known Oracle bug. Feel free to come down here and see for yourself."

Do try not to snigger when saying that.

The truth is, what would you be doing if you weren't a system administrator? We all know you'd go home, lie on your oversized skateboard, and roll beneath your own system to change its oil, wishing someone would drive it. Someone will. One day, someone will see your work and say "Hey, if I take that and destroy all that makes it clever or worthwhile, it will make my extremely niche problem less agonizing." Keep working!

In its purest form, systems administration is a disorder that benefits civilization, meaning that society has no interest in alleviating it or even developing a vaccine. Besides, your cop-

ing strategies are flat-out weird and make everyone else uncomfortable. They blame the repeated knocks to your head, illustrating yet again how people leap eagerly at explanations that are simple, elegant, and wrong. Everyone's happiest if you remain quarantined with your computers, separate from the uncontaminated population who are all busy anyway playing the latest phone game even though we know it's nothing but a knock-off of *Civilization* or *Doom* or *Solitaire*. Maybe Spacewar, if they consider themselves sophisticated.

Given all this, *why* are you worrying about a .0 release?

.

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

MICHAEL W LUCAS is author of over fifty books, including the forthcoming *Run Your Own Mail Server*. He wants to quit all this and become a pencil smuggler, but his defective coping strategies won't permit it. Learn more at https://mwl.io.

2024 Events Calendar

BSD Events taking place through March 2024

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.

State of Open Con 2024

February 6-7, 2024
London, UK
https://stateofopencon.com/

SOOCon24 is the UK's Open Technology Conference that focuses on community in Open Source Software, Open Hardware & Open Data. It has set itself up to be one of the most inclusive community open source events to date.

SCALE 21X

March 14-17, 2024
Pasadena, CA
https://www.socallinuxexpo.org/scale/21x

SCaLE is the largest community-run open-source and free software conference in North America. It is held annually in the greater Los Angeles area. Drew Gurkowski will also be hosting a FreeBSD workshop during the conference.
