

FreeBSD® JOURNAL

May/June 2024

Configuration Management Issue

# Introducing Hashicorp Vault

# rdist

# Submitting GitHub Pull Requests to FreeBSD

NEW Column

# Embedded FreeBSD

NEW Column

# Adventures in TCP/IP

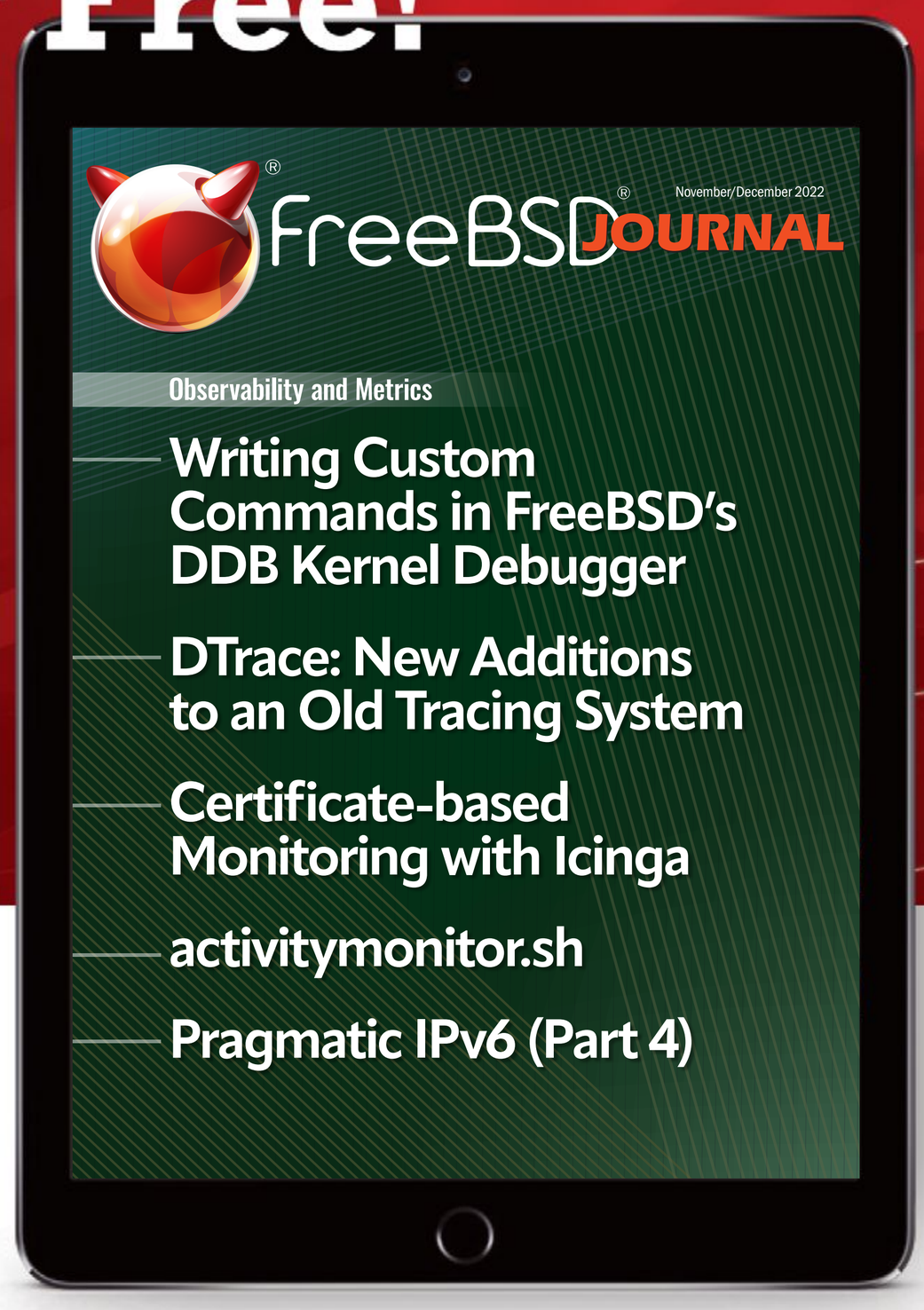
# FreeBSD® JOURNAL

# 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

**Editorial Board**

John Baldwin • Member of the FreeBSD Core Team and Chair of FreeBSD Journal Editorial Board

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

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

Benedict Reuschling • FreeBSD Documentation Committer and Member of the FreeBSD Core Team

Jason Tubnor • BSD Advocate, Senior Security Lead at Latrobe Community Health Service (NFP/NGO), Victoria, Australia

Mariusz Zaborski • FreeBSD Developer

**Advisory Board**

Anne Dickison • Director of Communications, Events, and Operations, FreeBSD Foundation

Justin Gibbs • Founder of the FreeBSD Foundation, President and Treasurer of the FreeBSD Foundation Board

Allan Jude • CTO at Klara Inc., the global FreeBSD Professional Services and Support company

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*

Kirk McKusick • Lead author of *The Design and Implementation* book series

George Neville-Neil • Past President of the FreeBSD Foundation Board, and co-author of *The Design and Implementation of the FreeBSD Operating System*

Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of AsiaBSDCon, and Assistant Professor at Tokyo Institute of Technology

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

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

Copyright © 2024 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

**Dear Readers,**

Welcome to the 2024 May/June issue! As I'm writing this, I'm listening to the FreeBSD Day livestreams from community members. It's been interesting hearing the different stories on how folks got involved, how they are using or contributing to FreeBSD, and what they are excited about in the FreeBSD world.

The best part is hearing their love and passion for FreeBSD. In this issue, authors span a range of experiences with the Project, from a recent GSoC participant to a few folks who have decades of experience with FreeBSD. Even though they have different experience levels, they share their knowledge and dedication on topics they have experience in.

Though this issue doesn't focus on FreeBSD Day, if you missed the recent celebrations, you can visit our FreeBSD Day page to see all the live streams and other related content.

I don't like to play favorites on authors or subjects, but I am keen on reading Warner's article on Submitting GitHub Pull Requests to FreeBSD because I personally want to start contributing to FreeBSD documentation. Maybe I'll write an article about that experience!

I want to mention one more thing. Many of you already know about the sudden passing of Mike Karels. It's a great loss for this community. I always saw him as a gentle soul, deeply connected to the history of FreeBSD while also being involved in its current development and support. Recently, Mike took on the role of Deputy Release Engineer to support Colin in his new position as Lead Release Engineer. In memory of Mike, I'd like to ask everyone to reach out to others in the community with kindness. Let's lend a helping hand to support each other and fill in the gaps — as Mike did — in this wonderful Project. Although we can never replace Mike, we can continue to carry on his passion for FreeBSD and keep his memory alive.

I hope you enjoy this issue!
**Deb Goodkin**
FreeBSD Foundation Executive Director

FreeBSD® JOURNAL

May/June 2024

Configuration Management Issue

9 **mfsBSD in Base**
*By Soobin Rho*

12 **rdist**
*By Cy Schubert*

17 **Hashicorp Vault**
*By Dave Cottlehuber*

28 **Submitting GitHub Pull Requests to FreeBSD**
*By Warner Losh*

3 **Foundation Letter**
*By Deb Goodkin*

5 **In Memory of Mike Karels**
*by the FreeBSD Foundation*

6 **We Get Letters**
*by Michael W. Lucas*

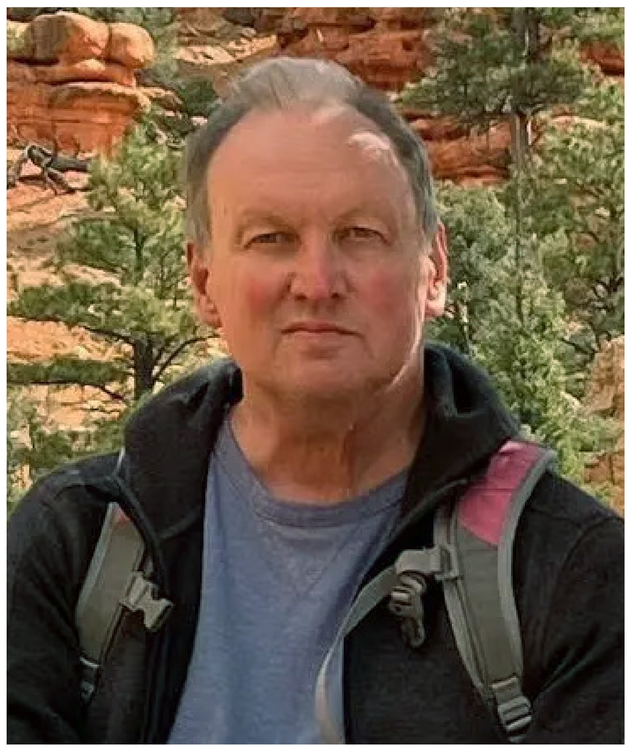
NEW Column
37 **Embedded FreeBSD: Breadcrumbs**
*By Christopher R. Bowman*

NEW Column
41 **Adventures in TCP/IP: TCP Black Box Logging**
*By Randall Stewart and Michael Tüxen*

48 **Practical Ports: Developing Custom Ansible Modules**
*By Benedict Reuschling*

56 **Events Calendar**
*By Anne Dickison*

# In Memory of
# **Mike Karels**



We are deeply saddened about the passing of Mike Karels, a pivotal figure in the history of BSD UNIX, a respected member of the FreeBSD community, and the Deputy Release Engineer for the FreeBSD Project. Mike's contributions to the development and advancement of BSD systems were profound and have left an indelible mark on the Project.

Mike's vision and dedication were instrumental in shaping the FreeBSD we know and use today. His legacy will continue to inspire and guide us in our future endeavors.

Our thoughts are with Mike's family, friends, and all who knew him. He will be greatly missed.

See the notification prepared by the family for more insights into his incredible life.

In lieu of flowers, the family has asked for donations to the Foundation to fund future FreeBSD projects.

## Articles about Mike:

BSD-Urgestein: Michael J. Karels mit 68 Jahren gestorben

# WeGet letters

by Michael W Lucas




Dear Letters Column,

My employer has dozens of servers, and I don't know how many operating systems. One of them has an uptime longer than *I* do, and nobody dares touch it. But some doofus left a computer magazine in the bathroom, the boss found it, and now his brain has latched onto "configuration management" as the solution to all our problems when what the datacenter really needs is a backpack nuke. How can I make him understand that these tools are not for environments like ours?

—I'm Already Doomed,
Asking You Can't Hurt

Dear Doomed,

"Asking me can't hurt." As if there's a limit to how much pain a sysadmin can experience, or how doomed they can be. Doom is not an integer value that can overflow. Doom is a social construct, and yours is fully built.

We've all seen the propaganda on configuration management. Deploy dedicated-purpose, highly tuned servers with a single command! Adjust computation clouds with a simple playbook! Seamlessly and transparently migrate from server to server! *Containers!* That's fine for people starting from a green field, but most system administrators work in environments best described as "baroque" if not "antediluvian." I find myself with a green field only when I personally raze the earth and wait for clover to grow. Not grass. Lawns are a climate atrocity. Unless you own sheep. Or goats, but if you own any kind of goat, you won't have a lawn for long, which demonstrates that any force for good is also an agent of desertification. Besides, who wants to wait for clover before installing a datacenter? Bulldoze away the rubble of that razed kindergarten and get on with your day.

Configuration management is one of those things where the advertised ideal is the enemy of reduced agony. Yes, the Canadian Hockey League can devops up a whole fleet of web servers to dynamically manage the increased load of their nation's entire citizenry simultaneously watching the last game of the Memorial Cup, and I told they can also devops up additional mental health facilities to handle the crushing depression when the London Knights lose to the Saginaw Spirit — who aren't even Canadian! You? Not so much. Dynamic purchasing is a prerequisite for dynamic provisioning, and you clearly lack both.

But you *can* deploy configuration management, and not in a malicious compliance sense. Skip the magic pixie dust of managing the entire server fleet. Your fleet couldn't be managed with a chair, a whip, and a flamethrower. But the painful parts of your systems can be taken under control.

Configuration management is a sysadmin tool. So, use it to fit your needs. Start with a handful of systems. Configure a management account with access so that your management system can ping those hosts. Congratulations — you've achieved malicious compliance! That serves your need with management, but it doesn't fit your management needs.

Each server is its own special snowflake, albeit a snowflake with rabies. When you start bringing these systems under control, start with something comparatively simple, with known good values, that's mostly consistent across Unix variants. There's a cliché about problems: "it's always DNS." It's always DNS because sysadmins don't understand DNS, and don't consistently update /etc/resolv.conf when nameservers change. That's where I always start. You're not only bringing systems under initial configuration management, you are auditing current DNS configurations as a prerequisite to that project. Your manager will love it. Group your hosts by operating system and bring their resolver under your management. If you're kind, comment the file.

```
# under configuration management
# your changes will be overwritten without a human ever seeing them
search mwl.io tiltedwindmillpress.com
nameserver 203.0.113.53
nameserver 2001:db8::53
```

Congratulations! You have DNS resolution under control. Will it change often? Hopefully not. But you could now change it trivially. If you want people to take you seriously you must always implement your threats, so schedule a monthly configuration management run to update resolv.conf.

You can legitimately claim your hosts are under configuration management, but you haven't used it to make your life easier. Look at another common service that every host has but is often configured inconsistently: SSH. Your organization probably has rules like "no password-based authentication." If it doesn't, wait until you have a security incident then propose it. Never waste a good crisis! The simplest way to lock down SSH and make sure it remains locked down is to bring sshd_config under centralized management. Yes, every operating system has its own sshd_config tweaks, because before integrating software Unix maintainers feel compelled to rub it in their armpits so it smells like them, but management systems use templates to accommodate such unhygienic behavior. You could probably recite the default sshd_config while sleeping through your commute, so make your managed configuration looks nothing like the default.

```
#Configuration Under Management
#Manual Changes Will be Overwritten
Port 9991
PasswordAuthentication no
Subsystem       sftp    /usr/libexec/sftp-server
```

Any sysadmin thinking "I'll just comment out the default option" will feel alarm all the way down their brainstem upon seeing this.

Piece by piece, you can bring broad sections of your environment under your control.

Changes to managed services will become trivial. Coworkers will see that. Discussions of changing unmanaged services will turn into "how can we bring this service under management?" Use those discussions to implement necessary changes in the environment, or to get yourself a better fourth monitor. Doom is a social construct, but with configuration management you can transform it into a protective shell. Or a battering ram. At the very least, you can share that pain.

Deploying configuration management has a rarely discussed but horrid side effect, however: whoever controls the environment, *controls the environment*. Any change must go through you. People can't permanently enable password authentication on that public-facing server, but that doesn't mean they won't whine at you about it. They'll expect you to participate in problem-solving, and nobody can survive becoming known as a problem-solver. That ineradicable reputation stain will serve only to get you the title of Company Scapegoat.

Fortunately, you know what goats are agents of. Start grazing.

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




**MICHAEL W LUCAS** is the author of *Networking for System Administrators* and a multitude of other crimes against civilization. A collection of these columns, Dear Abyss, will launch on Kickstarter soon, proving premeditation. See it for yourself at https://mwl.io/ks.


# Books that will help you. Or not.

"While we appreciate Mr Lucas' unique contributions to the Journal, we do feel his specific talents are not being fully utilized. Please buy his books, his hours, autographed photos, whatever, so that he is otherwise engaged."

— John Baldwin
*FreeBSD Journal* Editorial Board Chair

**https://mwl.io**



# mfsBSD in Base

BY SOOBIN RHO

mfsBSD is an in-memory FreeBSD.

What makes mfsBSD different is that it runs an instance of FreeBSD completely in memory — hence the mfs (memory file system). Of course, this means we can boot into FreeBSD without affecting our existing drives at all when using mfsBSD. We can, for example, use it for troubleshooting cloud or on-prem servers[1]. It's fascinating to see what kind of problems people have managed to solve using mfsBSD (search mfsBSD on the mailing lists). My personal favorite would be installing FreeBSD in a system where there's only a single drive available for whatever reason. I would use FreeBSD installation media if I could just stick in a USB drive, but if access to external devices was prohibited, I would first build an mfsBSD image; install the image to the drive; boot into mfsBSD; and run bsdinstall. Here's an example:

First, build mfsBSD:

```
# The patch set that integrates mfsBSD into base is under review and is available at:
# https://reviews.freebsd.org/D41705
cd /usr/src/release
make mfsbsd-se.img

# se here refers to mfsBSD special edition, which comes packed with
# dist files - base.txz and kernel.txz - which are required for bsdinstall.
cd /usr/obj/usr/src/${ARCH}/release/
ls -lh
```

Then, write mfsBSD to the drive:

```
# Install the mfsBSD image to your target drive.
# Replace ada0 with your target drive.
dd if=./mfsbsd-se.img of=/dev/ada0 bs=1M
reboot
```

Boot into mfsBSD and then execute **bsdinstall**:

```
# Copy the special edition dist files so that bsdinstall can use them for installation.
mkdir /mnt/dist
mount /dev/ada0p3 /mnt/dist
mkdir /usr/freebsd-dist
cp /mnt/dist/<version>/*.txz /usr/freebsd-dist/
bsdinstall
```

This is possible because mfsBSD loads a FreeBSD system entirely in memory. After mfsBSD has been loaded, the original disk can be modified completely as all mfsBSD files are now running in memory. As Matuška describes in his 2009 white paper, "mfsBSD is a

toolset to create small-sized but full-featured mfsroot based distributions of FreeBSD that store all files in memory."[2]

## Brief History of mfsBSD

Martin Matuška wrote mfsBSD. Looking back at the repository's commit history, he made the first ever commit on November 11, 2007, which is around the time FreeBSD 7.0 BETA was released. "This project [mfsBSD] is based on ideas from the Depenguinator project," which was a 2003 project by Colin Percival to create a toolset to remotely install FreeBSD on dedicated servers that only offer Linux distributions[3]. Matuška wanted to provide the Depenguinator functionality for the FreeBSD 6.x, and that's how mfsBSD started.

Since then, Matuška maintained https://mfsbsd.vx.sk/ to distribute images of mfsBSD as it gained popularity and has maintained its source code on GitHub[4] for the past seventeen years, fixing an uncountable number of bugs along the way, and adding support for `zfsinstall`, compressed tar of `/usr`, and so on.

In May, 2023, one year before this article was written, a Google Summer of Code project to integrate mfsBSD into base began.

## Google Summer of Code

How did it all start? I was reading *Hacker News*. (probably procrastinating on my college assignments at the time). That's how I first came across Google Summer of Code (GSoC). One of the top comments there said that FreeBSD was one of the participating organizations, and what I found interesting about the commentary was that there was no mention of anything else, as if everything else was self-explanatory.

In May, 2023, one year before this article was written, a Google Summer of Code project to integrate mfsBSD into base began.

I got hooked right away. The most surprising thing about FreeBSD was that macOS is a derivative of FreeBSD, and also that Netflix uses FreeBSD for its CDN. The GSoC application process involves submitting a project proposal. It was strongly recommended that applicants find a project idea from each of the organizations' project idea lists (unless you had your own idea that you'd like to pursue). I had a look at the list, and the mfsBSD project was the most interesting to me because other project ideas seemed closer to kernel development than I was comfortable with.

After shooting an email to my mentors, I got email back from Joseph Mingrone and Juraj Lutter; had a brief zoom call; and a few weeks later I got an acceptance from GSoC. After that, we had what's called a community bonding period, at which all of the contributors and mentors gathered around and had a virtual meeting for half an hour introducing ourselves. That was Friday, May 12, 2023.

## mfsBSD in Base

Three months and twenty-two days later, the project to integrate mfsBSD into base finally came to completion, after a lot of back and forth between debugging (a lot of the bugs were resolved by googling and digging through all the past GitHub Issues), and testing (sh scripts with two of my laptops, Thinkpad T440 and P17), and me asking too many questions to my mentors. A set of three patches were pushed to Phabricator.[5]

Basically, the first commit "mfsBSD: Vendor import mfsBSD" imports mfsBSD as **`contrib/mfsbsd`**. The main commit "release: Integrate mfsBSD image build targets into the release tool set" adds **`mfsbsd-se.img`** and **`mfsbsd-se.iso`** targets in **`release/Makefile`** as **`release/Makefile.mfsbsd`**. The last commit "release(7): Add entries for the new mfsBSD build targets" adds corresponding entries on **`share/man/man7/release.7`**. What this means is that we can now build mfsBSD in the same release Makefile we use for building all the FreeBSD installation media, such as cdrom, dvdrom, memstick, and mini-memstick, as **`make release WITH_MFSBSD=1`**.

This means is that we can now build mfsBSD in the same release Makefile we use for building all the FreeBSD installation media.

Now, the patch set is under review. mfsBSD previously existed outside the FreeBSD release tool chain, and only the release versions have been produced. What I envision with this patch set is to make mfsBSD images available as a part of base, and with this, we will be able to build custom mfsBSD images by invoking **`cd /usr/src/release && make release WITH_MFSBSD=1`**, which will then create **`mfsbsd-se.img`** and **`mfsbsd-se.iso`** at **`/usr/obj/usr/src/${ARCH}/release/`**.

## References


1. Matuška, Martin. (2022). FreeBSD on Hetzner dedicated servers — VX Weblog. [online] Available at: https://blog.vx.sk/archives/353
2. Matuška, Martin. (2009). mfsBSD Toolset to create memory filesystem based FreeBSD distributions. [online] Available at: https://people.freebsd.org/~mm/mfsbsd/mfsbsd.pdf
3. Colin Percival. (2003). Depenguinator — FreeBSD remote install. [online] Available at: http://www.daemonology.net/depenguinator/
4. Matuška, Martin. (2024). mmatuska/mfsbsd. [online] GitHub. Available at: https://github.com/mmatuska/mfsbsd
5. https://reviews.freebsd.org/D41705


**SOOBIN RHO** is a college senior at Augustana University in South Dakota. Born in South Korea but raised in Dubai, he ended up going to a college in the US and is now a part-timer at The Bancorp's Cybersecurity Department, which he will join as an Information Security Analyst after graduation. He has been a FreeBSD contributor since 2023 Google Summer of Code.

# rdist

BY CY SCHUBERT

## What is RDIST?

To quote the man page, "rdist is a program to maintain identical copies of files over multiple hosts." rdist is a general-purpose tool that can be utilized for multiple purposes, such as maintaining consistent copies of files across the network, like rsync and unison do, or as a distributed configuration management tool like cfengine, ansible, puppet, salt, or other configuration management tool.

## Why RDIST?

To understand why *rdist* understanding a little bit of its history will give us a better idea why it was created.

Written in 1983-1984 by Ralph Campbell at UCB, *rdist* first appeared in 4.3BSD (in 1986) and was one of the first applications to address the issue of distributed software management. During the late 1980s and 1990s it was distributed with almost every commercial UNIX. It became the standard for remote platform administration at the time.

*rdist* predates other software of its kind. It also predates rsync. Rsync is a backup tool used to backup or clone directory trees while *rdist* is generally used as a network file distribution application. Each is tailored to its design purpose.

With all these options, why use *rdist*?

- *rdist* is lighter weight than any of the subsequent configuration distribution applications such as ansible.
- *rdist* integrates easily into shell scripts and Makefiles.
- rsync is not able to distribute to multiple hosts in parallel like *rdist* can. Nor can rsync synchronize files using a configuration file, like *rdist* does. Rsync and *rdist* are designed for different purposes, rsync for backup and cloning of files while *rdist* works better as a configuration management tool.

On the flip side, why would one want to use another tool instead? As *rdist* is lightweight when compared to tools like cfengine and ansible, its ability to configure remote nodes on the network is limited to distributing files and performing simple post-distribution tasks. Whereas a heavier weight tool can perform pre-distribution tasks, this can be addressed through simple shell scripting or a Makefile.) As a personal example of this, I manage my ipfilter firewall rules using a configuration tool (called ipfmeta) to generate the firewall configuration files from a rules file and an objects file using a Makefile. The Makefile uses rdist to distribute the generated files to remote firewalls as defined in rdist's Distfile. One can think of a Distfile similar in relation to *rdist* as a Makefile is to make. Unlike rsync, *rdist* distributes files using rules coded in its Distfile.

## How Does RDIST Work?

Just as make(1) parses its Makefile to build applications, *rdist* parses its Distfile to describe what files or directories are to be distributed and any post-distribution tasks to be

performed. Initially, *rdist* used the insecure rcmd(3) interface for network communication. rcmd() would make a connection to a remote rshd(8). When the connection is made it spawns an rdistd(8) remote file distribution server to perform the distribution functions on the remote server. This is similar to how ssh's sftp-server provides remote function to sftp.

The Berkeley "r" commands, such as rsh are insecure. Today's implementation of *rdist* can use ssh as a transport instead of rsh. With ssh one can use ssh keys or GSSAPI (kerberos) authentication. Unlike ansible where connection is made under your own account and privilege escalation is done using "become," *rdist* must connect directly to root on the destination server. To facilitate this PermitRootLogin can be set to prohibit-password in sshd_config, forcing the use of either ssh keys or Kerberos tickets.

*rdist* does no authentication itself. It relies on the transport for this. Compared to ansible, it also relies on the ssh transport for authentication and it relies on su(1), sudo(1) or ksu(1) for privilege escalation.

*rdist* can be used to manage application files in a service account such as mysql, oracle or other application account. Replace root with the desired account name.

rdistd(8) must be in the user's search path ($PATH) on the target server.

*rdist* negotiates a protocol version. The systuils/rdist6 port/package uses the RDIST version 6 protocol while sysutils/rdist7 (alpha) uses the RDIST version 7 protocol.

## Installing RDIST

To install *rdist*, simply,

```
pkg install rdist6
```

or

```
pkg install rdist7
```

Or using ports, using rdist7 as an example,

```
cd /usr/ports/sysutils/rdist7
make install clean
```

## Using RDIST

As noted previously, *rdist* uses a configuration file similar to how host make uses its configuration file. We must build our Distfile.

There are three types of Distfile statement.

### The Distfile

Like make, *rdist* looks for a file named Distfile or distfile. Just as we can override the name of the Makefile make uses, we can override the name of the *rdist* Distfile.

Distfiles contain a sequence of entries that specify the files to be distributed (copied), to which nodes those files are to be copied, and the post-copy operations that are to be performed following the distribution of the files.

### Variables

One or more items can be assigned to a variable using the following format.

```
<variable name> '=' <name list>
```

For example,

```
HOSTS = ( matisse root@arpa )
```

This defines the strings matisse and root@arpa to the variable HOSTS.
Another example assigns three directory names to the variable called FILES.

```
FILES = ( /bin /lib /usr/bin /usr/games )
```

### Distributing Files to Other Hosts

The second statement type tells *rdist* to distribute files to other hosts. Its format is,

```
[ label: ] <source list> '->' <destination list> <command list>
```

<source list> is the name of a file or a variable. <destination list> is a list of hosts to which the files will be copied. While <command list> are a list of rdist instructions to be applied to applied to the copy operation.

The optional label identifies the statement for partial updates when the label is referenced from the command line.

For example, from my firewall Distfile,

```
install-ipf: ipf.conf -> ${HOSTS}
      install /etc/ipf.conf ;
      special "chown root:wheel /etc/ipf.conf; chmod 0400 /etc/ipf,conf" ;
```

This tells *rdist* to install ipf.conf to the nodes listed in the HOSTS variable. The install command line tells *rdist* the file is to be installed to /etc/ipf.conf.

The special command line tells *rdist* to run chown and chmod following the copy operation.

The install-ipf label can be addressed on the *rdist* command line, limiting the operation to just that operation, i.e. rdist install-ipf.

The command list includes keywords such as install, except, special and cmdspecial.

| | |
|---|---|
| install | Identifies where to install target files. |
| notify | List email addresses to be notified upon completion of the copy operation. |
| except | An exception pattern of files not to be copied. |
| except_pat | Same as except but using a regexp pattern. |
| special | Shell commands to be executed after each file is copied. |
| cmdspecial | Shell commands to be executed after all files in a rule have been copied. |

A simple example follows. It copies my working copy of this article to a directory in my FreeBSD working directory tree.

```
HOSTS = ( localhost )

FILES = ( /t/tmp/rdist.odt )

${FILES} -> ${HOSTS}
        install /home/cy/freebsd/rdist/rdist.odt ;
```

Here we are copying the file /t/tmp/rdist.odt to /home/cy/freebsd/rdist/rdist.odt on my laptop. Of course, a simple cp(1) command would suffice, but this simple example gives us a taste of how to copy single files. Also note that the destination is a file by the same name. If the destination was a directory, i.e., /home/cy/freebsd/rdist, it would remove all the files and subdirectories in the target directory, replacing it with a single rdist.odt file. Be careful when specifying target files or directories. This would be like,

```
rsync -aHW -delete /t/tmp /home/cy/freebsd/rdist
```

Unanticipated results can make for a bad day.
The rdist(1) man page provides a better example:

```
        HOSTS = ( matisse root@arpa)

         FILES = ( /bin /lib /usr/bin /usr/games
                  /usr/lib /usr/man/man? /usr/ucb /usr/local/rdist )

         EXLIB = ( Mail.rc aliases aliases.dir aliases.pag crontab dshrc
                  sendmail.cf sendmail.fc sendmail.hf sendmail.st uucp vfont )

        ${FILES} -> ${HOSTS}
                  install -oremove,chknfs ;
                  except /usr/lib/${EXLIB} ;
                  except /usr/games/lib ;
                  special /usr/lib/sendmail "/usr/lib/sendmail -bz" ;

        srcs:
        /usr/src/bin -> arpa
                  except_pat ( \\.o\$ /SCCS\$ ) ;

        IMAGEN = (ips dviimp catdvi)

        imagen:
        /usr/local/${IMAGEN} -> arpa
                  install /usr/local/lib ;
                  notify ralph ;

        ${FILES} :: stamp.cory
                  notify root@cory ;
```

In the example above, files listed in the FILES variable will be copied from the localhost to the machines listed in the HOSTS variable. Except for files listed in the EXLIB variable, /usr/games/lib and a pattern. After each file is copied, sendmail with a -bz option is run.

Typically, special is used to run shell commands, but in the example above, /usr/lib/sendmail is executed (as if it were a shell), passing the quoted arguments to sendmail.

Three files in /usr/local will be copied to /usr/local/lib on the target systems, with an email to ralph when the copy has been completed.

A time stamp file is touched when the job completes, sending an email to root@cory. Time stamp files are used to avoid gratuitous copies. For example, if any of the listed files is newer than the time stamp file, the file is copied. (Conversely, ansible uses a checksum.)

## Gotchas

As mentioned, things can go wrong if one is not careful. Like rsync, *rdist* does not verify the source file is the same type of object (file or directory) as the target. It is easy to replace a destination file with a directory or replace a destination directory with a file. Like rsync, it can render a system unusable. Be careful and test in a sandbox or jail.

## Summary

*rdist* is an excellent tool when used in conjunction with scripts, makefiles, or other tooling in scenarios when no one tool can do everything, combined with other tools as I do to manage my ipfilter firewalls, ipfmeta, make Makefiles, *rdist* Distfiles, and git rdist integrates nicely to create a lightweight application. In the case of integration with heavier weight tools like ansible or cfengine which don't integrate with scripts and Makefiles, *rdist* fills that unique niche. *rdist* follows the original UNIX philosophy of a single tool for a single purpose that can be integrated with other tools to create new tools and applications.

## Bibliography


- https://man.freebsd.org/cgi/man.cgi?query=44bsd-rdist&sektion=1&apropos=0&manpath=FreeBSD+14.0-RELEASE+and+Ports
- https://www.magnicomp.com/download/rdist/overhaul.pdf
- https://www.cs.umb.edu/~ckelly/teaching/common/project/linux/sys_admin/p7_rdist.pdf

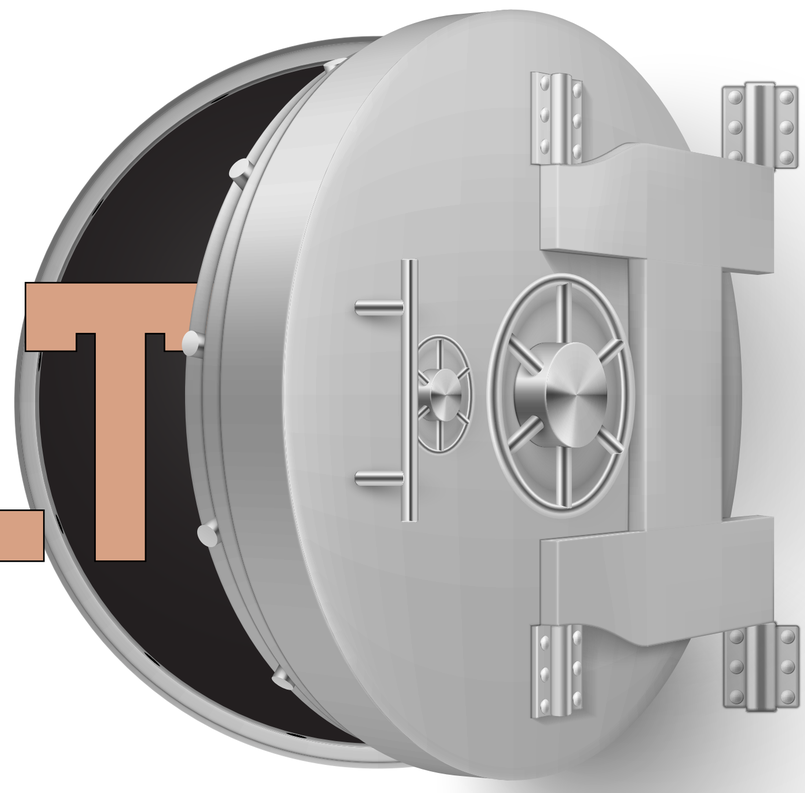

---

**CY SCHUBERT** is a FreeBSD src and ports committer. His career began over fifty years ago, writing and maintaining electrical engineering applications written in Fortran. His experience includes IBM MVS (mainframe) systems programming, writing extensions to the MVS kernel and Job Entry Subsystem 2 (JES/2). His career took a turn down the UNIX path thirty-five years ago moving to SunOS, Solaris, Tru64, NCR AT&T, DG/UX, HP-UX, SGI, Linux and FreeBSD systems administration.

Cy's FreeBSD journey also began thirty-five years ago. After trying a Linux distro with Linux kernel 0.95 and seeing it didn't support UNIX domain sockets, he tried an experimental Linux kernel. After a disastrous month of restoring EXTFS filesystems corrupted by the experimental kernel, Cy posted a query on the FreeBSD and NetBSD USENET newsgroups. The only person to reply to Cy's question was Jordan Hubbard from the FreeBSD project. Since Jordan was the first and only person to answer, Cy decided to try FreeBSD first. He's been using FreeBSD since 2.0.5. He became a ports committer in 2001 and a source committer eleven years ago. He is currently employed by a Canadian subsidiary of a large managed services provider.

# VAULT

BY DAVE COTTLEHUBER



Working from home is the new normal. But from a security perspective, things just got a lot more complicated. Gone are the secure offices with carefully manicured security perimeters and 24x7 physical security.

Security professionals talk about the categories of risks we care about as our "threat landscape", or "security posture".

That's a fancy way of saying that we can make decisions such as not caring about GCSB, KGB, CIA, Mossad, and other Government-funded attackers, but we do care about forgetting a laptop on the train, or having somebody's office broken into, to steal high value items. And we care about passwords and credentials. A lot.

We do care about forgetting a laptop on the train, or having somebody's office broken into, to steal high value items.

## But Where To Store the Secrets?

Of particular interest, both to attackers, and to ourselves as sysadmins or developers, is managing, and rotating, secrets.

How many environments have you seen, where database credentials haven't changed in, well, forever? Where long-running batch jobs use the same password across multiple systems? Or where changing one of these can result in unexpected downtime, as a cascading butterfly effect ripples across the company, and ultimately your career?

Clearly sticky notes, and credentials checked into git repos, are neither scalable, nor secure. We need something that is reasonably generic, and yet supports a wide range of use cases.

So what's a sensible DevOps engineer supposed to do?

One solution is a typical password store, like BitWarden or 1Password, and to extend these tools across a team of people, and into various command-line tools. While these solve a part of the problem, they don't solve enough of it.

They are primarily user-facing tools, and are not easily wired up to a complex pipeline of git repos, puppet and ansible deployments, and ensuring that credentials are rotated regularly, and only accessible to appropriate users and systems.

## Secrets Management Platforms

Sometimes called Key Management Systems, these are commonly found integrated into Cloud vendors. Azure, Amazon, Google, Oracle all offer tightly coupled and well-integrated tools for their platforms.

However if you're reading the FreeBSD journal, you are more likely to be viewing the idea of giving all your secrets to companies, with extreme trepidation. Ideally we'd like to manage our own secrets, without relying on a third party in a foreign country.

## Trade Offs

These systems focus on enabling operations teams to distribute, and manage, key material, in a highly secure, and controlled fashion.

They aim to address the challenge of securely managing and orchestrating secrets in modern, complex environments where applications, systems, and users require access to very sensitive information.

For example, they may integrate changing a secret, to triggering a rollover of container using that secret, to the new version.

They may enable a batch job to decrypt briefly financial information, and then return it updated safely encrypted again.

Or it may provide a one-use token, that allows a newly provisioned service, to connect to a given database, in a way that we can be sure that token was neither hijacked in transit, nor able to be used by more than one instance.

When a new instance is provisioned or deployed, it will get a new one-time use token, unique to this instance, and this time.

This type of functionality is ideal for separating the deployment of systems and associated secrets, from direct access to those secrets. This type of smoke and mirrors is often called cubby-hole deployment — where a one-use token, tightly bound to a single deploy, is injected during deployment by an automation toolset. This token is used at instance boot, to fetch the runtime secret, dedicated and bound to this instance, perhaps by IP address or other caveats.

At startup a master key is needed to unlock all other keys.

## Getting Started

This article devotes itself to introducing Hashicorp Vault, which has a feature-parity open fork called OpenBao, similar to the Terraform / OpenTofu licensing fork as well.

There are other tools, but Vault is ported to FreeBSD, and I'm sure OpenBao will land soon too.

## Vault's Sweet Spot

Vault, and other KMS, are not great places to keep your Webstorm IDE license, or a scanned copy of your passport. It's not end-user friendly, and it's definitely not usable on a mobile device.

But if you're managing servers, databases, and networks, it's fantastic. It can be easily integrated with Terraform, Chef, Puppet, Ansible, and almost anything that has, or uses, a command-line or terminal interface.

## Internals

Vault stores all keys and values encrypted on disk. At startup, therefore, a master key is needed to unlock all other keys. To avoid having a single lightweight key, Vault uses SSS, or Shamir's Secret Sharing, to split a large complex key into separate secrets, that can be

recombined to unlock the vault. Try the Shamir Demo in a modern WASM-capable web browser.

Cleverly, SSS allows a configurable degree of redundancy — say, 3 of 5 keys are needed to unlock the vault. Thus, your 3 main sysadmins can unlock it, but in the absence of any one, you can beg your lawyer or accountant to loan their key if needed, and reach your quorum of 3. Each admin submits their unlock key locally, and an API challenge is used to prevent any single admin from obtaining all segments of the master key.

Once the vault is unlocked, from a user perspective, it functions largely like any other HTTP-accessible key-value store. We can store small files like ssh private keys, or TLS certificates, the usual passwords, or even get vault to generate ephemeral passwords for a limited time, and limited purpose.

## Getting Started

While vault supports complex deployments, with consensus protocols and multiple servers, I've found a small highly reliable physical server with a hot standby and a zfs replicated backup, to be sufficient. Of course, I rely on Tarsnap for a fully offline backup — an absolute essential requirement for something as critical as all of our secrets!

## Install and Configure

The usual incantations must be performed as root:

```
# pkg install -r FreeBSD security/vault
# mkdir -p /var/{db,log}/vault /usr/local/etc/vault
# chown root:vault /var/{db,log}/vault /usr/local/etc/vault
# chmod 0750 /usr/local/etc/vault
# chmod 0770 /var/{db,log}/vault
```

The **rc.conf** settings, you can use **sysrc(8)** for this, or your preferred ops toolkit.

```
# /etc/rc.conf.d/vault or where-ever you prefer
vault_enable=YES
vault_config=/usr/local/etc/vault/vault.hcl
```

And vault's config file. There are of course many options, most of this is self-explanatory. For our test deployment, we will disable TLS and use the loopback IP.

```
# /usr/local/etc/vault/vault.hcl
default_lease_ttl = "72h"
max_lease_ttl = "168h"

ui = true
disable_mlock = false

listener "tcp" {
  address = "127.0.0.1:8200"
  tls_disable = 1
  tls_min_version = "tls12"
  tls_key_file = "/usr/local/etc/vault/vault.key"
  tls_cert_file = "/usr/local/etc/vault/vault.all"
```

```
}

storage "file" {
  path = "/var/db/vault"
}
```

Now run the daemon in the foreground:

```
$ vault server -config /usr/local/etc/vault/vault.hcl
==> Vault server configuration:

Administrative Namespace

             Api Address: http://127.0.0.1:8200
...
```

In a new terminal, let's check the status:

```
$ export VAULT_ADDR=http://localhost:8200/
$ vault status
vault status
Key                Value
---                -----
Seal Type          shamir
Initialized        false
Sealed             true
Total Shares       0
Threshold          0
Unseal Progress    0/0
Unseal Nonce       n/a
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Note that the vault is uninitialised, and still sealed. Let's fix that:

```
$ vault operator init --key-shares=3 --key-threshold=2
Unseal Key 1: jjcVgHTjWw3j4BsyDhugvS9we5t5qMAhJL8bSWzySjbG
Unseal Key 2: WfMeZPA7ixleQAMeeAqyey+gwrxDn9WNfSvdKzdLMaeA
Unseal Key 3: V9cd1eVBH6mstyoS2pbD6S80R7NJVz7jPvlPOcLOUVlw
Initial Root Token: hvs.RAeqzETRhOXOImMPw7xrXbAl

$ export VAULT_TOKEN=hvs.RAeqzETRhOXOImMPw7xrXbAl

$ vault status
```

```
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       3
Threshold          2
Unseal Progress    0/2
Unseal Nonce       n/a
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Note that the vault is now initialised, and also still sealed. So let's fix that next, using the newly generated key shards:

```
$ vault operator unseal
Unseal Key (will be hidden):
Key                Value
---                -----
Seal Type          shamir
Initialized        true
Sealed             true
Total Shares       3
Threshold          2
Unseal Progress    1/2
Unseal Nonce       6ce4351d-012b-df3f-a176-34d266f00795
Version            1.14.1
Build Date         2023-11-04T05:16:56Z
Storage Type       file
HA Enabled         false
```

Repeat the unsealing with a different key each time until **`sealed`** changes to **`false`**. The final step is to enable auditing, because security people love logs.

```
$ vault audit enable file path=/var/log/vault/audit.log
Success! Enabled the file audit device at: file/
```

Feel free to tail this, there are no secrets ever stored here, so it's only an audit log of requests.

### Shamir Secret Ring

Now that you've unsealed the vault, distribute your secrets by encrypted avian carrier, to your chosen secret keepers. Some suitable ceremony is required, and also to ensure these secrets are adequately protected, both against incompetence or less, as well as Mossad and North Korean agents.

By now, you should be ready to store secrets.

### Storing Secrets

Vault has a concept of engines - there's a simple key-value storage, also one for ssh certificates, AWS and Google Cloud integrations, RabbitMQ, PostgreSQL, and many more. Each one needs to be separately enabled.

```
$ vault secrets enable -version=2 kv
Success! Enabled the kv secrets engine at: kv/
```

From here on in, we need to specify both the engine type, and it's mount path. It's possible to retrieve data as JSON, or yaml as well, and even to store files directly.

```
$ vault kv put -mount=kv blackadder scarlet_pimpernel="we do not know"
=== Secret Path ===
kv/data/blackadder

======= Metadata =======
Key                Value
---                -----
created_time       2024-05-12T23:04:50.283028044Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1

$ vault kv get -mount=kv -format=json blackadder
{
  "request_id": "48141452-8f8f-b497-9c53-1af71e24e2a5",
  "lease_id": "",
  "lease_duration": 0,
  "renewable": false,
  "data": {
    "data": {
      "scarlet_pimpernel": "we do not know"
    },
    "metadata": {
      "created_time": "2024-05-12T23:04:50.283028044Z",
      "custom_metadata": null,
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  },
  "warnings": null
}

$ vault kv put -mount=kv blackadder scarlet_pimpernel="comte de frou frou"
```

```
=== Secret Path ===
kv/data/blackadder

======= Metadata =======
Key                Value
---                -----
created_time       2024-05-12T23:08:22.369551931Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            2

$ vault kv get -mount=kv -format=yaml blackadder
data:
  data:
    scarlet_pimpernel: comte de frou frou
  metadata:
    created_time: "2024-05-12T23:08:22.369551931Z"
    custom_metadata: null
    deletion_time: ""
    destroyed: false
    version: 2
lease_duration: 0
lease_id: ""
renewable: false
request_id: 686965d9-811f-8689-d75f-a02f7dded9a7
warnings: null

$ vault kv put kv/blackadder scarlet_pimpernel=@/etc/motd.template
```

## Role-based Access

Vault can be configured to require github authentication, and delegate roles and authentication to something other than LDAP. Many of you will rejoice at this news. With Github authentication, 2FA can be enforced for all users, so this represents a reasonable trade-off for small teams.

```
$ vault auth enable github
vault auth enable github
Success! Enabled github auth method at: github/

$ vault write auth/github/config organization=skunkwerks
Success! Data written to: auth/github/config

$ vault write auth/github/map/teams/admin value=admins
Success! Data written to: auth/github/map/teams/admin
```

Place this small policy file in **/usr/local/etc/vault/admins.hcl**

```
# grant members of github admins group all rights in the kv/ mount
path "kv/*" {
  capabilities = ["create", "read", "update", "delete", "list"]
}
```

And then enable it within vault:

```
$ vault policy write admins /usr/local/etc/vault/admins.hcl
Success! Uploaded policy: admins
```

You can of course make more restrictive policies, in rights, or in paths, or in selected mounts, for various groups, such as a deployment bot.

Finally, each user that wishes to use github auth, for vault, must go to https://github.com/settings/tokens and add a new personal token with privileges of **admin- read:org**.

This can be used now to generate your vault login token via

```
$ vault login  -method=github token=$GITHUB
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key                    Value
---                    -----
token                  hvs....
token_accessor         ...
token_duration         72h
token_renewable        true
token_policies         ["admins" "default"]
identity_policies      []
policies               ["admins" "default"]
token_meta_org         skunkwerks
token_meta_username    dch

$ vault kv get -mount=kv -format=yaml blackadder
...
```

## Vault in Automation

Automation tools such as Chef, Puppet, Ansible, and more can use vault to store secrets, for decryption at deployment time, or even in some circumstances, only to be decrypted at runtime.

Let's look at the first case, decrypting secrets at deploy time. Effectively, this extends the templating capabilities of automation tools, and relies on being able to trigger service re-starts, after having pushed new and updated secrets.

We can use vault in 4 ways:

- Ansible and similar tools can store secrets in vault, and only decrypt them at deploy time, using lookups

- `rc.d` framework scripts can use app roles to fetch their credentials at startup, letting the local root user have a delegated token that only permits issuing cubby-hole tokens. The daemon itself will only be able to retrieve its own credentials
- vault can issue time- and IP-bound dynamic credentials that are revoked on expiration, for daemons, cronjobs, & batch scripts that are time limited
- we can also template out files at runtime, using vault agent

### Ansible

There are a number of plugins for ansible, and confusingly, there is an internal ansible "vault" module that is not compatible with Hashicorp Vault.

Install the plugin, and use the typical `lookup` functionality:

```
super_secret: "{{lookup('hashivault', 'kv', 'blackadder', version=2)}}"
```

### rc.d App Roles

An AppRole is a built-in authentication method, specifically for machines and applications to authenticate to Vault, and then subsequently obtain a token that *only* then allows fetching relevant secrets. This is often called a cubby-hole credential, as it only permits unwrapping the outer layer, to get a key inside.

These can be restricted by time, a limited number of uses, and more. Our trusted root process generates this restricted secret id, and passes it and the role id to the daemon to fetch its own credentials. The generation of the secret id can be set up in such a way that only these credentials can be minted.

Once again, we enable the `approle` mount, before creating our app-specific credentials, as it is a form of authentication. For convenience, this approle will re-use the existing `admins` group policy used earlier, but it should have a more restrictive one, specifically for this daemon/service.

```
$ vault auth enable approle
Success! Enabled approle auth method at: approle/

$ vault write auth/approle/role/beastie \
        secret_id_ttl=60m \
        token_num_uses=10 \
        token_ttl=1h \
        token_max_ttl=4h \
        secret_id_num_uses=40 \
        policies="default,admins"
Success! Data written to: auth/approle/role/beastie

$ vault read auth/approle/role/beastie/role-id
Key        Value
---        -----
role_id    6caaeac3-d8fa-a0e3-83ba-7d37750603c2

$ vault write -f auth/approle/role/beastie/secret-id
```

```
Key                   Value
---                   -----
secret_id             8dd54c92-fe54-0d6d-bee6-e433e815aaa1
secret_id_accessor    cb9bc17c-c756-42b3-c391-b61ebde12bff
secret_id_num_uses    0
secret_id_ttl         0s
```

If we wanted to make these secrets usable within a hypothetical **beastie** daemon, these two parameters can be put in an **/etc/rc.conf.d/beastie** file, which can be secured to only be readable by root.

```
beastie_enable=YES
beastie_env="
  ROLE_ID=6caaeac3-d8fa-a0e3-83ba-7d37750603c2
  SECRET_ID=8dd54c92-fe54-0d6d-bee6-e433e815aaa1
  SECRET_PATH=kv/beastie
  VAULT_ADDR=http://localhost:8200/
"
```

The **/usr/local/etc/rc.d/beastie** script runs a pre-cmd that fetches the secret as root, and injects it into the child environment.

```
start_precmd=${name}_vault
beastie_vault() {
    # Authenticate with Vault using the approle
    VAULT_TOKEN=$(vault write auth/approle/login role_id="$ROLE_ID" \
      secret_id="$SECRET_ID" \
      -format=json | jq -r '.auth.client_token')

    # Retrieve the secret from Vault
    export BEASTIE_SECRET=$(vault kv get -field=data -format=json ${SECRET_PATH} | jq -r .)
}
```

### Agents

Vault also provides an agent mode, which does a lot of the credential management for you, and supports templating of simple config files.

## Sealing the Vault

Typically, a vault is left unsealed and running for months on end, barring patching and upgrades. In the event of a security incident, it suffices to halt the server that runs the vault daemon entirely, or optionally issue a **seal** command. This shuts vault, and unloads the master secret key.

```
$ vault operator seal
Success! Vault is sealed.
```

**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





**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

FreeBSD FOUNDATION

# Submitting GitHub Pull Requests to FreeBSD

BY WARNER LOSH

The FreeBSD Project recently started supporting GitHub pull requests (PRs) to make it easier to contribute. We found that accepting patches via our bug tracker Bugzilla resulted in far too many useful contributions being ignored and growing stale, so contributors should prefer GitHub PRs for changes, leaving bugs in Bugzilla. While Phabricator works well for developers, we've also found it's easy to lose track of changes from outside contributors there. Unless you are working directly with a FreeBSD developer who has told you to use Phabricator, please use GitHub instead. GitHub PRs are easier to track, easier to process, and more familiar to the wider open source community. We hope for faster decisions, fewer dropped changes, and a better experience for all.

Since FreeBSD's volunteers have limited time, The Project has developed standards, norms, and policies to use their time efficiently. You'll need to understand these to submit a good PR. We have some automation which helps submitters fix the common mistakes, allowing the volunteers to review nearly ready submissions. Please understand we can only accept the most useful contributions and some contributions cannot be accepted.

Next, I'll cover how to turn your changes into a Git branch, how to refine them to meet the FreeBSD Project's standards and norms, how to make a PR from your branch, and what to expect from the review process. Then I'll cover how volunteers evaluate PRs and tips for perfecting your PR.

> We hope pull requests make it easier to get changes into FreeBSD and provide quick decisions when there are issues.

This article focuses on commits to the base system, not the documentation or ports trees. These teams are still revising the details for these repositories.

## Project Standards

The Project has detailed standards for various aspects of the system. These standards are described in the FreeBSD Developer's Handbook and the FreeBSD Committer's Guide. Coding standards are documented in FreeBSD manual pages. By convention, manual pages are divided into sections. All the style manual pages are in section 9 for historical reasons. References to manual pages are traditionally rendered as the name of the page, followed by its section number in parentheses, for example style(9) or cat(1). This documentation is available on any FreeBSD system with the man command, or online.

The Project strives to produce a well-documented integrated system that covers both a kernel that controls the machine as well as a user-space implementation of common Unix utilities. Contributions should be well written with relevant comments. They should include updates to the relevant manual pages when the behavior changes. When you add a flag to a command, for example, it should be added to the manual page as well. When new functions are added to a library, new man pages should be added for the functions. Finally, The Project views the metadata in the source code control system part of the system, so commit messages should conform to the project's standards.

The Project's Standard for C and C++ code is described in style(9). This style is often referred to as "Kernel Normal Form" and is adopted from the style used in Kernighan and Ritchie's *The C Programming Language*. It's the standard that research unix used, as continued within the CSRG at Berkelely who produced the BSD releases. The FreeBSD project has modernized these practices over the years. This style is the preferred style for contributed code, and describes the style used in most of the system. Contributions which change this code should follow this style except for a few files that have their own style. Lua and Makefiles also have their own standards, found in style.lua(9) and style.Makefile(9) respectively.

Commit messages follow the form favored by the Open Source communities that use git. The first line of the commit message should summarize the entire commit, but do so in 50 or so characters. The rest of the message should describe what changed and why. If what changes is obvious, only explaining why is preferred. The lines should be 72 characters or fewer. It should be written in the present tense, with an imperative tone. It ends with a series of lines that Git calls "trailers" which The Project uses to track additional data about the commits: where they came from, where details about the bug can be found, etc. The Commit Log Message section of the Committer's Guide covers all the details.

Contributions should be well written with relevant comments.

## Unacceptable Changes

After a few years of experimenting with accepting changes via GitHub, The Project has had to establish some limits to accepting contributions via GitHub from people who have not yet earned write access to the project's repositories. These limits ensure that the volunteers that verify and apply the changes to make the best use of their time. Consequently, The Project is unable to accept:

- Changes too large to review  on GitHub
- Typos in comments
- Changes discovered by running static analyzers over the tree (unless they include new test cases for the bugs the static analyzers found). Exceptions can be made on a case-by-case basis for "obviously correct" fixes in parts of the system that do not interact well with our testing harness.
- Changes that are theoretical, but have no specific bug or articulable behavior defect.
- Performance optimizations that aren't accompanied by before / after measurements to show improvement. Micro-optimizations are rarely worth it, as compiler and CPU technology often makes them obsolete (or even slower) in only a few years.

- Changes that are contentious. These need to be socialized on the **freebsd-arch@freebsd.org** or most appropriate mailing list first. GitHub provides a poor forum for discussing these sorts of issues.

PRs should make the Project better in some, user-visible way.

## Evaluation Criteria

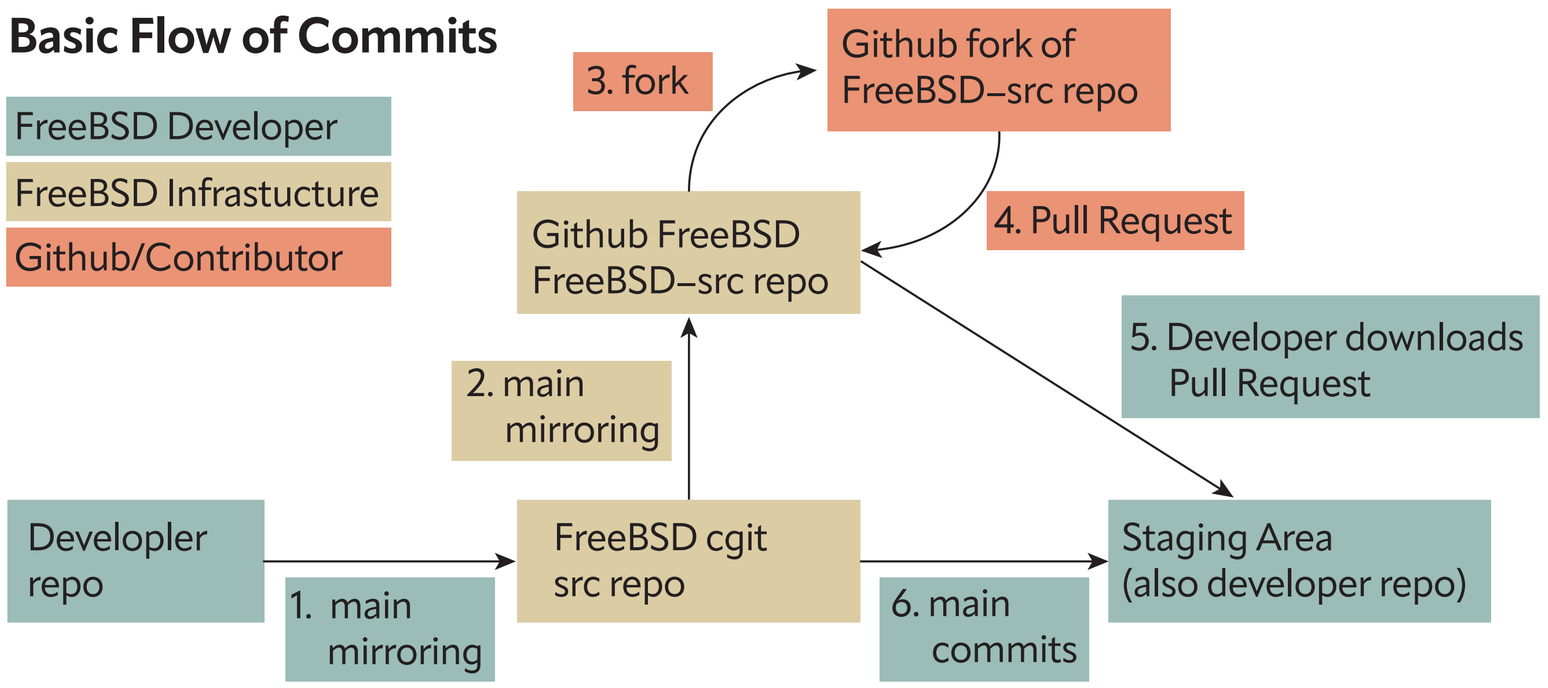
- Is the change one that the Project is accepting?
- Is the scope/scale of the change good?
  - Are there a reasonable number of commits (say less than 20)?
  - Are each of the commits a reviewable size (say less than 100 lines)?
- Does C and C++ code confirm to style(9) (or the file's current style)
- Do changes to lua confirm to style.lua(9)
- Do changes to Makefiles conform to style.Makefile(9)
- Do changes to man pages pass both `mdoc -Tlint` and `igor`?
- Have contentious changes been discussed in the proper mailing list?
- Does **make tinderbox** run successfully?
- Do the changes fix a specific, articulable problem or add a specific feature?
- Are the commit messages good?

While avoiding these pitfalls:

- Do the changes introduce new test regressions?
- Do the changes introduce behavior regressions?
- Do the changes introduce a performance regression?

## Overview of the Process

At a high level, contributing to FreeBSD is a straightforward process, though getting into the details can obscure this simplicity.




1. FreeBSD developers push commits directly to the FreeBSD repository, which is hosted in the FreeBSD.org cluster.
2. Every 10 minutes, the FreeBSD src repository is mirrored to the freebsd-src GitHub repo.
3. A user wanting to create a PR will create a branch in their fork of the freebsd-src repo
4. The changes on a user branch are used to create a FreeBSD PR.
5. A FreeBSD developer reviews the PR, provides feedback, and may request changes from the user.
6. A FreeBSD developer will push the changes into the FreeBSD src repo.

## Prepping for Submitting Pull Requests

You'll need to create a GitHub account, if you don't already have one. This link will walk you through the process of creating a new GitHub account. Since many people already have a GitHub account for other reasons, we'll skip delving into the details.

The next step is forking FreeBSD's repository into your account. Using the Github web interface is the easiest way to create a fork and to explain since you will only need to do this once. Changes to your fork do not affect FreeBSD's repo. Users can fork repositories by clicking the "Fork" button as shown in **Figure 1**. You will want to click on the highlighted "Create a new fork" menu item. This will bring up a screen similar to **Figure 2**. From here, click the green "Create Fork" button.

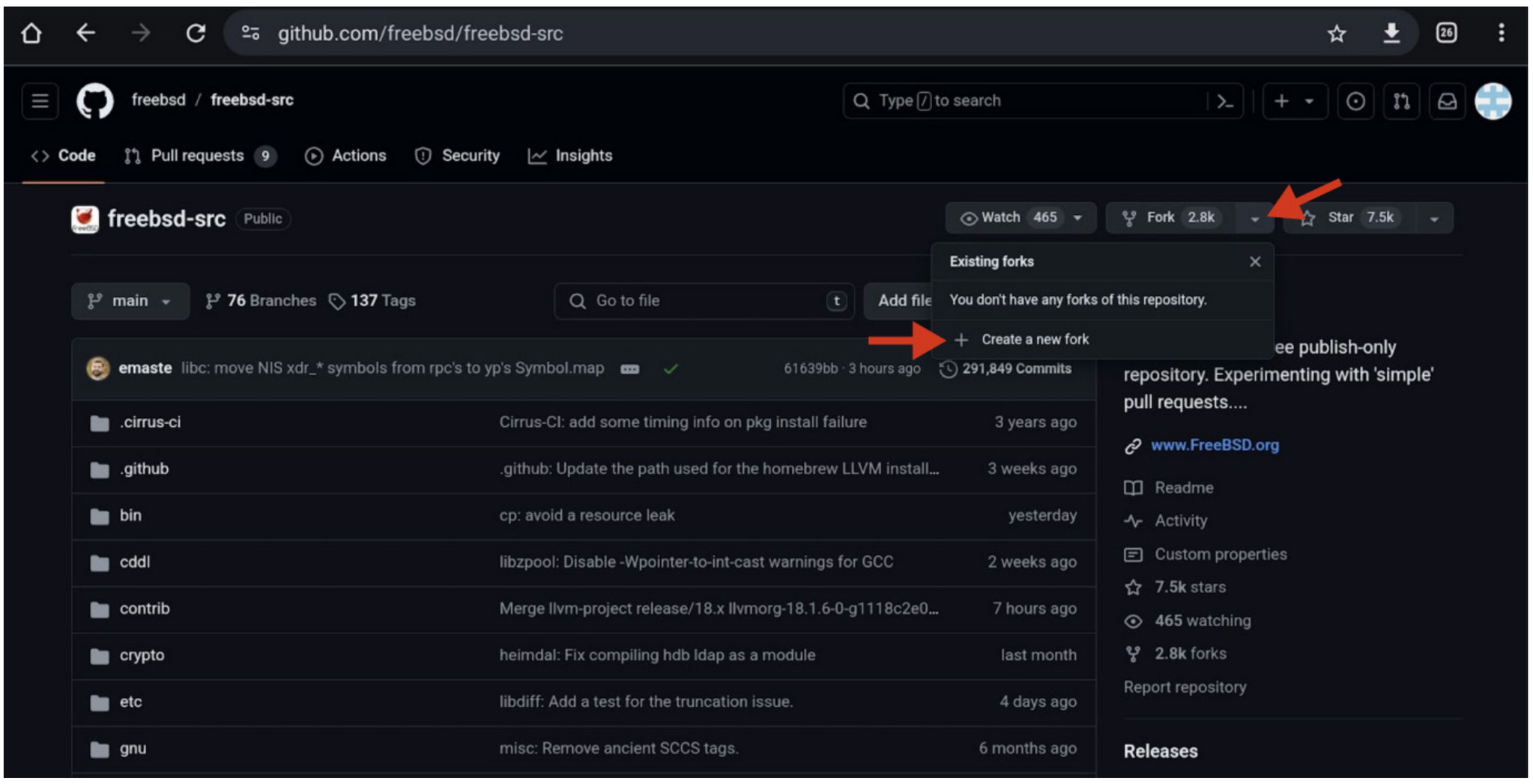



**Figure 1:** After clicking the down arrow next to Fork, you'll see a pop-up for the create fork dialog.

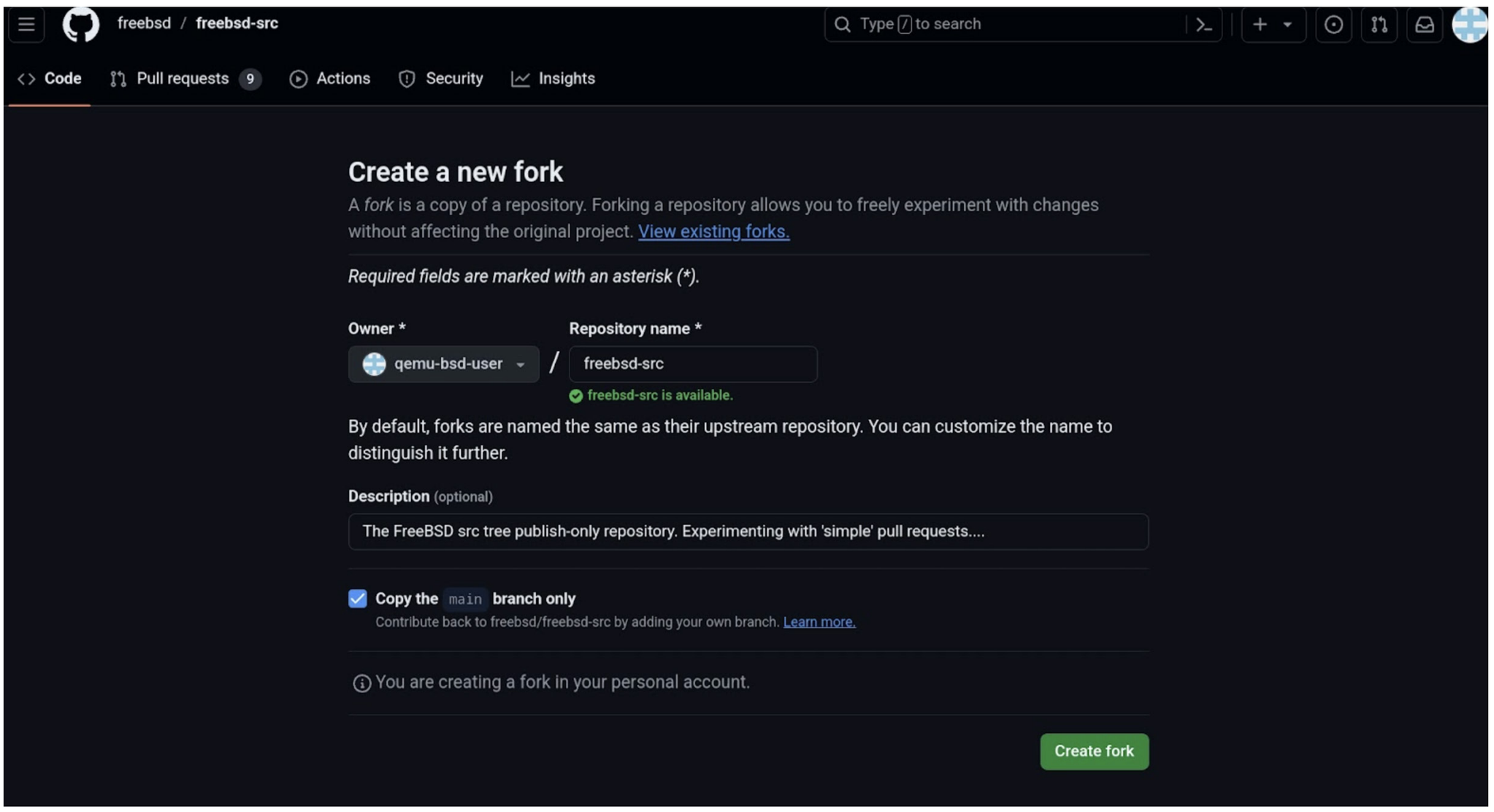



**Figure 2:** Creating a Fork, part 2.

Once you click on "Create Fork," the GUI will redirect to the newly forked repository. You can copy the URL you need to clone the repository in the usual spot, shown in **Figure 3**.

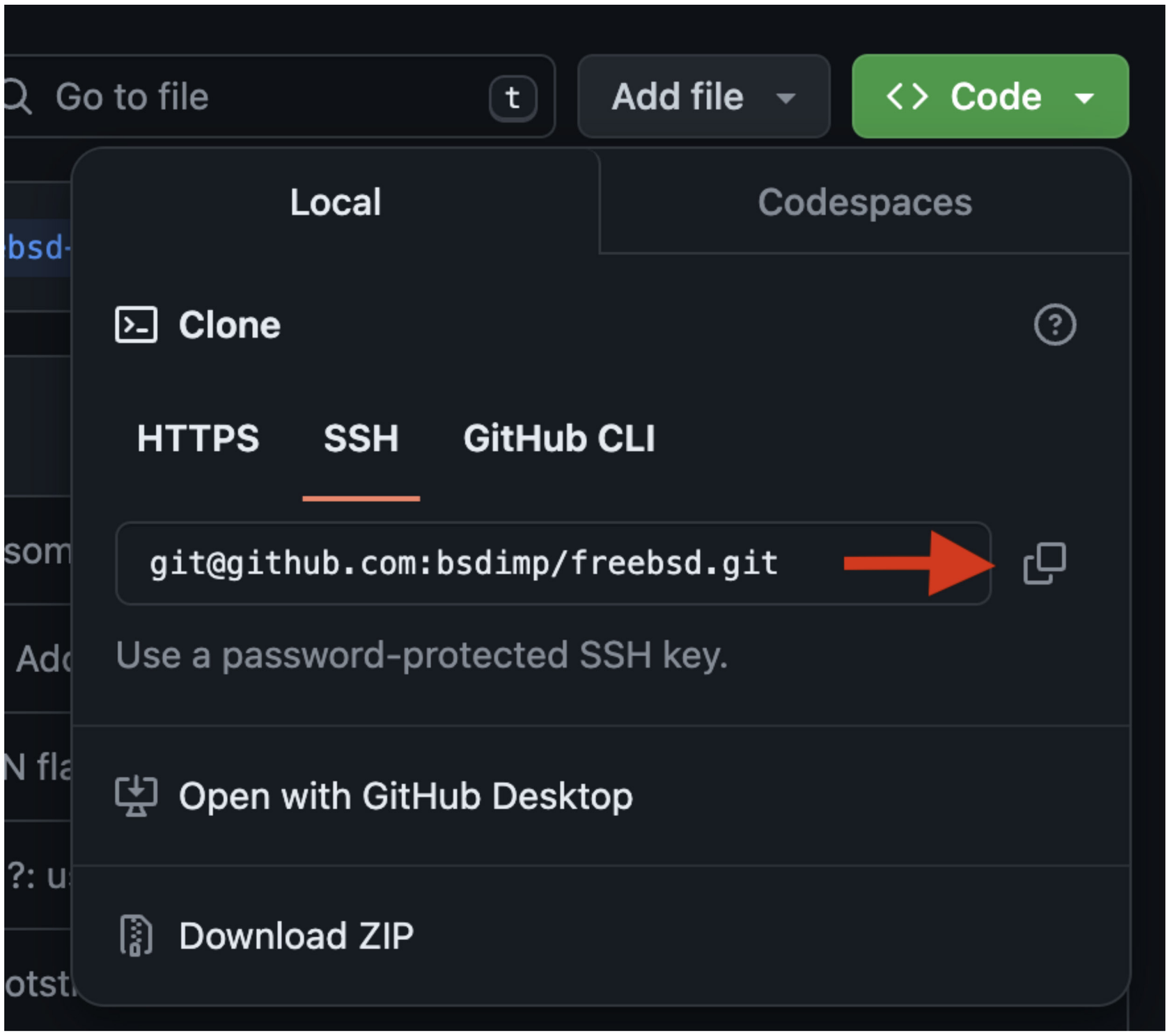



**Figure 3:** Copying the URL to clone (I forked ages ago with old repo name)

This concludes the steps you'll do with the GitHub web interface. The rest of these commands will be done in a terminal window for a host running FreeBSD. For simplicity, the screen shots have changed to the commands or the commands and the output produced by those commands.

Clone your newly created repository using the commands below.

```
% git clone
Cloning into 'freebsd-src'...
remote: Enumerating objects: 3287614, done.
remote: Counting objects: 100% (993/993), done.
remote: Compressing objects: 100% (585/585), done.
remote: Total 3287614 (delta 412), reused 815 (delta 397), pack-reused 3286621
Receiving objects: 100% (3287614/3287614), 2.44 GiB | 22.06 MiB/s, done.
Resolving deltas: 100% (2414925/2414925), done.
Updating files: 100% (100972/100972), done.
% cd freebsd-src
```

Please note you should change "user" in the above command to your GitHub username. The "-o github" will name this remote "github," which will be used in the examples below.

The PR workflow generally requires a branch. We'll assume you've followed something like the following commands, though there are many ways to use a pre-existing branch that are beyond the scope of this article.

```
% git checkout -b journal-demo
% # make changes, test them etc
% git commit
```

It is important that all the commits you make have your real name and email address as the "Author" of the commit. Git has two configuration fields for this. user.name contains your real name. And user.email has your email address. You can set them like so:

```
% git config --global user.name "Pat Bell"
% git config -global user.email "pbell@example.com"
```

In addition, please read our advice on Commit Log Messages and follow it when creating commits.

Most changes we get via PRs are small, so we'll move on to submitting them. However, if you have large changes, please read the **Evaluation Criteria** below before submitting for a smoother process.

## Submitting Your Pull Request

The next step is to push the **journal-demo** branch to GitHub (as with the above, substitute your GitHub username for "user" below:

```
% git push github
Enumerating objects: 24, done.
Counting objects: 100% (24/24), done.
Delta compression using up to 8 threads
Compressing objects: 100% (16/16), done.
Writing objects: 100% (16/16), 5.21 KiB | 1.74 MiB/s, done.
Total 16 (delta 13), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (13/13), completed with 8 local objects.
remote:
remote: Create a pull request for ‘journal-demo’ on GitHub by visiting:
remote:   https://github.com/user/freebsd-src/pull/new/journal-demo
remote:
To github.com:user/freebsd-src.git
 * [new branch]              journal-demo -> journal-demo
```

You'll notice that GitHub helpfully tells you how to create a pull request. When you visit the above URL, you're presented with a blank form, as shown in **Figure 4**.

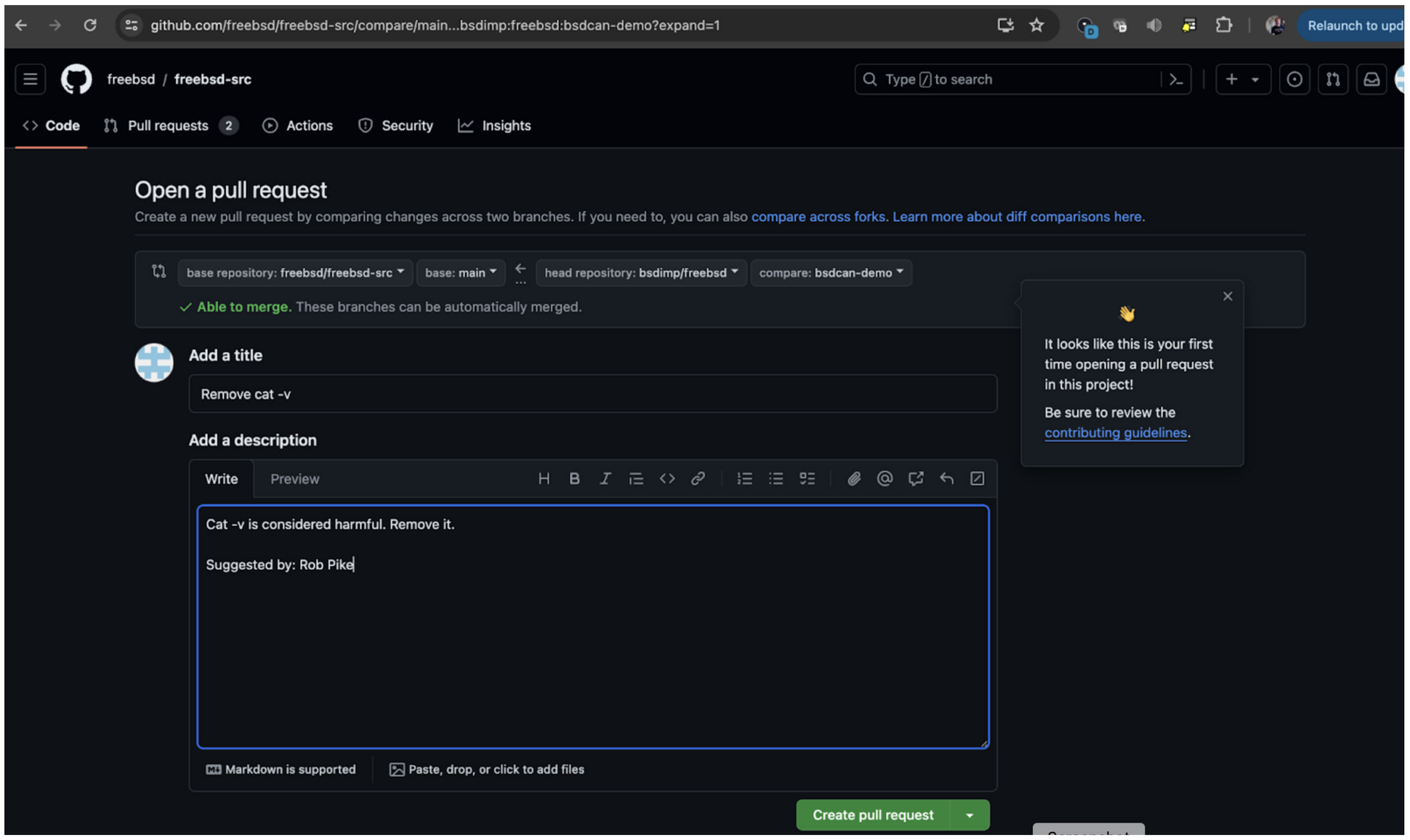



**Figure 4:** Pull request submission form.

In the "Add a Title" field, add a brief description of your work that conveys the essence of the changes. Keep this to about a dozen words, so it's easy to read. If the branch has only one commit, use the first line of the commit message for that change here. If it's multiple commits, you'll need to summarize them into one short title.

In the "Add a Description" field, write a summary of your changes. If this is a branch with just one commit, use the body of the commit message here. If there are multiple commits, then create a brief summary which briefly describes the problem solved. Explain what you changed and why, if it isn't obvious.

The example in **Figure 4** attempts to address a famous historical dispute between Bell Labs and Berkeley. It is a good example of a contentious commit outlined below. It is a good example of a contentious commit that should be socialized.

## What to Expect

After your submission, the evaluation process begins. Several automated checkers will run. These make sure that the format and style of your submission conform to our guidelines. They ensure that the proposed changes compile. They will provide feedback for changes you should make before someone looks at it. Some of these tests take time, so checking back a few hours after your submission is a good idea. Items flagged by the automated testing will be among the first things our volunteers will ask you to correct, so it saves everybody time to proactively address them.

## Replying to Feedback

Once you've received feedback, oftentimes code changes are required. Please make the changes that were suggested. Usually this means that you'll have to edit some subset of your changes (either commit messages, or the commits themselves). GitLab has a good tutorial on the mechanics of using **git rebase**.

Once you're made your changes, you'll need to push the changes back to your branch so the PR updates and the feedback loop starts over:

```
% git push github --force-with-lease
```

## Supply Chain Attacks

Recently, a bad actor attacked the xz source base to insert code that compromised sshd on certain Linux systems. FreeBSD was unaffected by this attack due to a combination of luck and process. Our process is designed to resist such attacks by having multiple layers of protection. We review code before we allow it to be tested. We only run automated testing when it's clear there's no obvious mischief in the submissions. Questions that might seem unnecessary are often motivated by the increasingly hostile work environment with which open source projects must cope.

## Wrapping up

Whether you are a casual user that has an occasional tweak to make FreeBSD better, or a more intense developer who submits so many changes that you'll earn a commit bit, the Project welcomes your submissions. This article tries to cover the basics of doing this, but is more geared to the causal user. The online resources will help for situations beyond the basics.

---

**WARNER LOSH** has been contributing to open source since before the FreeBSD project existed or the term "open source" was formally defined. He's recently been delving into the early history of Unix to discover its rich, hidden legacy. He lives in Colorado with his wife and daughter in a strawbale house heated by the sun, a small boiler, and the occasional antique computer.

Write For Us!

Contact Jim Maurer with your article ideas.

(**maurer.jim@gmail.com**)



# 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!

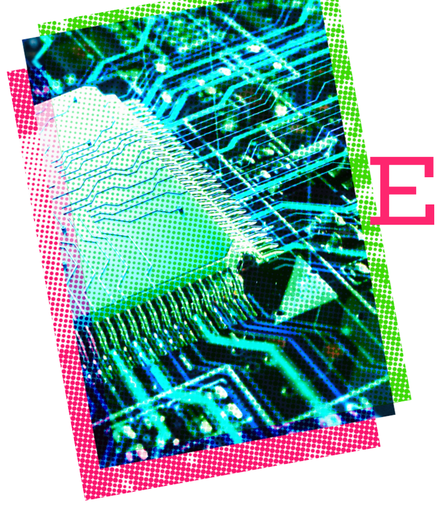
Your investment will help:

- 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



Embedded
FreeBSD

# Breadcrumbs

BY CHRISTOPHER R. BOWMAN

I've been using FreeBSD for almost 3 decades now. I first installed FreeBSD in the early 1990s because the package system made it really easy to install versions of the free CAD software I was using at the time to build my first silicon chips in 2 micron (2000nm, that's not a typo). Not having to figure out how to configure and compile 3 or 4 packages myself meant I could install the system in an evening and literally do chip design in my basement. Before that I would have to drive all the way to the university and spend hours each day working late into the night on the expensive Sun workstations. Not only could I do everything at home, but the tools also ran faster to boot! While I can program, I've always used FreeBSD as a base on which to do computing. I never developed for the community. Now, I want to build something with FreeBSD, not simply use FreeBSD to get my work done.

There is a plethora of small, embedded boards on the market — some with great mindshare. The raspberry pi in all its incarnations is a wonderful example. For me the most interesting thing about these small, embedded boards is their ability to interface with the outside world. Many of these small boards have GPIO pins that you can toggle from the CPU and thus interface to all sorts of real-world things. But I'm a hardware engineer at heart; I really want to build hardware. While I've done well in my career, I still don't have the spare $10 Billion dollars I'd need to build my own fab or the millions of dollars I'd need to buy EDA (Electronic Design Automation) software to design my own chips. There are some interesting projects out there now if you want to build your own silicon, but I started this journey before I found those. I always thought to myself, yes, I could buy an rpi or any of the other great boards out there like the Arduino, but what would I do with them? So, I continued to read about these board but never dipped my toe in. Finally, I found the board for me.

There is a plethora of small, embedded boards on the market — some with great mindshare.

Xilinx, now AMD, produces an inexpensive set of chips they call Zynq. These chips have single or dual AMD Coretex A9 CPUs complete with MMUs and a host of peripherals built into them. These chips, while not open source, are well documented from a hardware perspective. Most importantly someone (Thomas Skibo) had already done all the hard work to port FreeBSD to them. Like I said, I'm a hardware engineer at heart, and while I like writing software, porting FreeBSD from scratch was just a bigger project than I wanted to chew on at that point. There are a variety of boards with this chip (ZYBO, ZEDBOARD, ARTYZ7) at

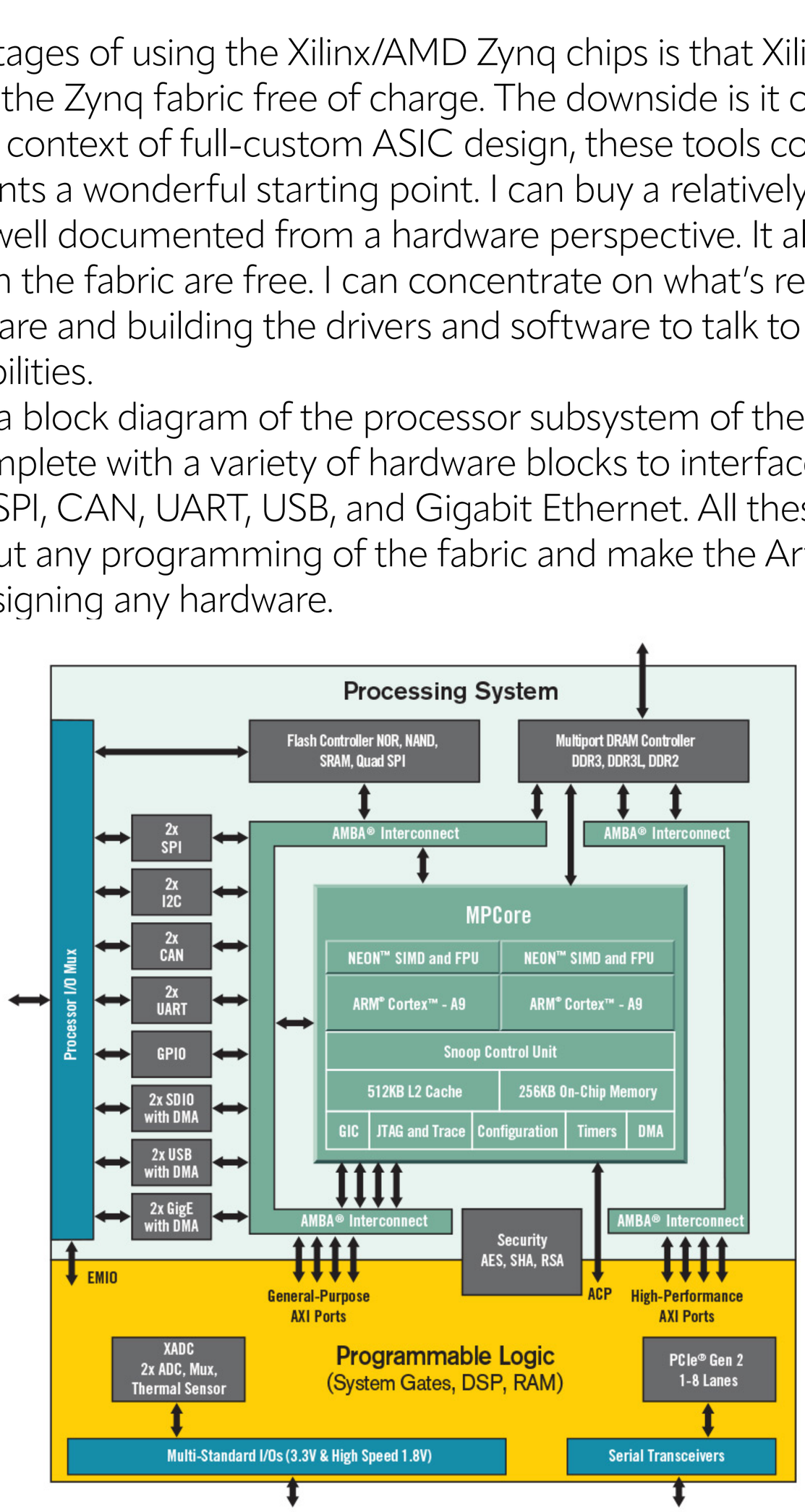
varying price points — some as low as around $200. But the most import thing for me as a hardware engineer was that these chips have an FPGA fabric built into them and connected to the CPU.

For those of you who don't know what FPGAs are, you can think of them as the halfway house between CPUs and full custom chips (ASICs). FPGA is an acronym for Field Programmable Gate Arrays. In their basic incarnation they are a large array of gates that can be interconnected to form a circuit. You'll often hear this array of gates and their interconnect network referred to as a fabric. FPGA circuits are usually designed in a language called Verilog (or VHDL) which is the same language that is used in the design of full-custom silicon ASICs. The flow of tools used to build FPGA designs is very similar to ASIC design. It's very flexible but can also be very complex. And while Verilog looks much like C, it's really a different mindset altogether.

One of the advantages of using the Xilinx/AMD Zynq chips is that Xilinx provides a basic set of tools to target the Zynq fabric free of charge. The downside is it only runs under Windows or Linux. In the context of full-custom ASIC design, these tools could cost millions.

For me, this presents a wonderful starting point. I can buy a relatively cheap board with a Zynq chip. It is fairly well documented from a hardware perspective. It already runs FreeBSD. Tools to do designs in the fabric are free. I can concentrate on what's really interesting to me: designing hardware and building the drivers and software to talk to it. There is an amazing number of possibilities.

The figure shows a block diagram of the processor subsystem of the Zynq chip. As you can see it comes complete with a variety of hardware blocks to interface to the outside world including i2C, SPI, CAN, UART, USB, and Gigabit Ethernet. All these blocks are there out of the box without any programming of the fabric and make the Arty Z7 a fine board to use even without designing any hardware.




**Zynq Z7000-20 processor subsystem block diagram**

While there are many Zynq boards the one I chose is the Digilent Arty Z7-20, not to be confused with the Digilent Arty A7 which uses a different chip that is all fabric and no processor subsystem. The Arty Z7-20 has dual ARM Cortex A9 processors (the Z7-10 has only one core) which I would guess are about as powerful as the Pentium Pros I ran decades ago, but hey what do you want in an embedded board? These cores are fully supported under LLVM running on FreeBSD. Also included is 512MB of DDR3 memory running over a 16-bit bus at 1050 MBps. The board has an Arduino/chipKIT Shield connector which allows you to easily connect any Arduino shield. It also has a couple of PMOD ports which like the Arduino shield connector is a standardized connector for external peripherals. There is a wide variety of PMOD devices listed on the Digilent site that you can buy cheaply and easily. The board includes a pair of HDMI ports, one in and one out, connected to the fabric. It also has a gigabit ethernet port that functions under FreeBSD. There are USB ports (that I've never tried) and a variety of LEDS, switches and buttons all connected to the fabric. The Zynq chip itself also contains dual ADCs (Analog to Digital Converters) allowing you to sample external signals. System storage is a standard MicroSD card up to 32 Gigabytes in size. If you never touched the fabric, you'd have a fairly complete and capable embedded board. Heck it's better than what I was running FreeBSD on when I started in the 1990s!

While there are many Zynq boards the one I chose is the Digilent Arty Z7-20.

Booting the Arty Z7 board is simple. I use **dd** to copy a prebuilt image (you can find my 14.1 RELEASE one HERE) to a MicroSD card using a cheap USB to SD card adaptor. Note that cards larger than 32 gigabytes aren't supported. When a card is inserted into my system, a device **/dev/da0** shows up. This may be slightly different on your system if you already have a **/dev/da0** device. You can easily see which device to use by listing the da devices in **/dev** before and after inserting a card. The following copies over an image:

```
# dd if=FreeBSD-armv7-14.1R-ARTY_Z7-49874af3.img \
of=/dev/da0 bs=1m status=progress
```

Meanwhile, I plug one end of a USB cable into the Arty micro-B USB connector and the other end into my FreeBSD machine. Then fire up a serial terminal program, connect to the appropriate device and set it for 115kpbs 8-N-1.

```
# cu -s 115200 -l /dev/ttyU1
```

When the image copy is complete, I insert the SD card into the Arty board and press the reset button. Make sure you setup the serial terminal before you hit reset so that you get to enjoy the entire FreeBSD boot sequence. Within seconds, I have a tiny but fully functional Unix host sitting on the network ready to start my quest for world domination!

Since the image I use comes preconfigured with DHCP on the ethernet port and with a preconfigured user account and ssh keys, I can simply connect the board to my ethernet switch, add the board's MAC to my DHCP, create a DNS entry and SSH into the board using its DNS name.

Right there it's a small, relatively cheap, full-fledged Unix host. You could host services like DHCP/DNS/NTP. You could use it for network intrusion. Possibilities are endless, but we haven't even scratched the surface yet, as we haven't even talked about using the external pins or the fabric. And that will be the focus of a future column.

Are you using these boards? Which one? What are you using it for. I'd love to hear your comments and feedback.

---

**CHRISTOPHER R. BOWMAN** first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.

FreeBSD®

## Help Create the Future. Join the FreeBSD Project!

**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**

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

FreeBSD FOUNDATION

Adventures in TCP/IP

# TCP Black Box Logging

BY RANDALL STEWART AND MICHAEL TÜXEN

## Evolution of TCP Logging in FreeBSD

4.2 BSD was released in 1983 and included the first TCP implementation in BSD. This version also added support for a facility to debug the TCP implementation. The kernel part, controlled by the kernel option `TCP_DEBUG` (disabled by default), provides a global ring buffer of `TCP_NDEBUG` (default 100) elements and routines to add an entry to the ring buffer whenever a TCP segment is sent or received, a TCP timer expires, or a TCP related protocol user request is processed. These events are only added for sockets, for which the `SOL_SOCKET`-level socket option `SO_DEBUG` was enabled. 4.2 BSD also provided the command line utility `trpt` (transliterate protocol trace), which can read the ring buffer from a live system or core file and print it. It not only prints the TCP header of the sent and received TCP segments, but also the most important parameters of the TCP endpoint when TCP segments are sent or received, TCP timers expire or a TCP related protocol user request is processed. It is important to note, that in case of a panic, the contents of the ring buffer might provide enough information to figure out why the system ended up in the bad state. However, since this facility does not match today's usage of TCP anymore, it was removed in FreeBSD 14. In earlier versions of FreeBSD, building a kernel with a non-default configuration was required.

4.2 BSD was released in 1983 and included the first TCP implementation in BSD.

In 2010, the `siftr` (statistical information for TCP research) kernel module was added to FreeBSD. No changes to the FreeBSD kernel are required, just loading the module to use it. `siftr` is only controlled via `sysctl`-variables. When enabled, controlled by the `sysctl`-variable `net.inet.siftr.enabled`, `siftr` writes its output to a file, controlled by the `sysctl` variable `net.inet.siftr.logfile` (default `/var/log/siftr.log`). The entries, except for the first and last, correspond to a sent or received TCP segment and provide information about the direction, IP addresses and TCP port numbers and internal TCP state. Since it is envisioned to be used in combination with a packet capturing tool like `tcpdump`, no additional information about the TCP segments (for example the TCP header) is stored. Every *n*-th TCP segment will be logged for each TCP connection, seperately for the sent and receive direction. n is controlled by the `sysctl`-variable `net.inet.siftr.ppl`. A TCP port filter controlled by the `sysctl-variable net.inet.siftr.port_filter` can be applied to focus on specific TCP connections. All information is stored in ASCII, therefore no additional userland tool is re-

quired to access the information. In the default configuration, only TCP/IPv4 is supported. Adding support for TCP/IPv6 requires a re-compilation of the `siftr` kernel module.

In 2015, a facility was added to the kernel, which is controlled by the kernel option `TCP_PCAP` (disabled by default). If enabled on a non-default kernel, each TCP endpoint contains two ring buffers: one for sent and one for received TCP segments. It should be noted that no additional information, not even the time when a TCP segment was sent or received, is stored. The maximum number of TCP segments in each ring buffer is controlled by the `IPPROTO_TCP`-level socket options `TCP_PCAP_OUT` and `TCP_PCAP_IN`. The default value is controlled by the `sysctl`-variable `net.inet.tcp.tcp_pcap_packets`. Since there is no userland utility to extract the contents of the ring buffers, the use of this feature is limited to analyzing core files. It should be noted that also the TCP payload is logged, which might make it hard to share core files containing such information due to privacy aspects. Support of this facility is planned to be removed in the upcoming version FreeBSD 15.

The latest TCP logging facility, the TCP BBLog (TCP black box logging) was added in 2018. It was initially called TCP BBR (black box recorder), but to avoid confusion with the TCP congestion control called BBR (bottleneck bandwidth and round trip propagation time), it is now called BBLog. BBLog is enabled on all 64-bit platforms of all production releases of FreeBSD. It combines the advantages of `TCP_DEBUG` and `TCP_PCAP` without their disadvantages. Therefore, it is intended to replace both of them. BBLog can be controlled via the `sysctl`-interface and the socket API as described later in this column.

BBLog is enabled on all 64-bit platforms of all production releases of FreeBSD.

## Introduction to BBLog

BBLog is controlled by the kernel option `TCP_BLACKBOX` (enabled by default on all 64-bit platforms) and the kernel source code is in `sys/netinet/tcp_log_buf.c` and its corresponding header file `sys/netinet/tcp_log_buf.h`. On a BBLog enabled kernel, there is a device (`/dev/tcp_log`) for providing BBLog information to userland tools, and each TCP endpoint contains a list of BBLog events.

Each event contains a standard set of important TCP state information as well as (optionally) a block of event-specific data. These events are collected to a set limit and when the limit is reached these events may be sent over to a `/dev/tcp_log` which, if open, relays the information to the reading process(s) for recording. Note that if no process has the device open then the data is discarded.

`tcplog_dumper`, from the FreeBSD ports collection, can be used to read from `/dev/tcp_log` as described below.

All FreeBSD TCP stacks have been instrumented with a minimum of the following event types:

- `TCP_LOG_IN` — Generated when a TCP segment arrives.
- `TCP_LOG_OUT` — Generated when a TCP segment is sent.
- `TCP_RTO` — Generated when a timer expires.
- `TCP_LOG_PRU` — Generated when a PRU event is called into the stack.

The TCP RACK and BBR stack generate many other logs; there are currently 72 event

types defined in `netinet/tcp_log_buf.h`. These logs instrument a wide variety of conditions and both the TCP BBR and RACK stack even have a verbose mode that can be used when debugging the stack. These verbose options are set through stack specific `sysctl`-variables `net.inet.tcp.rack.misc.verbose` and `net.inet.tcp.bbr.bb_verbose`.

Each TCP endpoint can be in one of the following BBLog states:

- `TCP_LOG_STATE_OFF (0)` — BBLog is disabled.
- `TCP_LOG_STATE_TAIL (1)` — Log only the last events on the connection. Each connection is allotted a finite number (default 5000) of log entries. When the last entry is hit, reuse the first entry overwriting it.
- `TCP_LOG_STATE_HEAD (2)` — Log only the first events processed on the connection up to the limit.
- `TCP_LOG_STATE_HEAD_AUTO (3)` — Log the first events processed on a connection and when you reach the limit dump the data out to the log dumping system for collection.
- `TCP_LOG_STATE_CONTINUAL (4)` — Log all events and when you hit the maximum collected number of events send the data out the log dumping system and start allocating new events.
- `TCP_LOG_STATE_TAIL_AUTO (5)` — Log all events at the tail of a connection and when you hit the limit send the data out to the log dumping system.

Note for general debugging the BBLog state `TCP_LOG_STATE_CONTINUAL` is often used. However in some specific instances (debugging a panic) it is preferable to use the BBLog state `TCP_LOG_STATE_TAIL` such that the last BBLog events are recorded inside the panic dump.

BBLog states can be set when the TCP connection is established or via the socket API.

BBLog states can be set when the TCP connection is established or via the socket API. In addition to that, they can be set when a TCP connection fulfills a particular condition. This is called a trace point and they are specified for particular TCP stacks and are identified by a number. One example of a tracepoint is getting `ENOBUF` when the TCP stack calls the IP output routine.

The contents of each event consists of three parts:

1. A BBLog header containing the IP addresses and TCP port numbers of the TCP connection, the time of the event, an identifier, a reason, and a tag.
2. A set of mandatory state variables of the TCP connection including the TCP connection state and various sequence number variables.
3. A set of optional data like information about send and receive buffer occupancy, TCP header information and further event specific information.

Note that TCP payload information is not contained in any BBLog event, but information about IP addresses and TCP port numbers is included in every BBLog event.

## Configuration of BBLog

There are basically two ways of configuring BBLog. The general configuration is done via the `sysctl`-interface and the TCP connection specific configuration is done via the socket API.

### Generic Configuration via the `sysctl`-Interface

This is the list of BBog related `sysctl`-variables, which are all under `net.inet.tcp.bb`:

```
[rrs]$ sysctl net.inet.tcp.bb
net.inet.tcp.bb.pcb_ids_tot: 0
net.inet.tcp.bb.pcb_ids_cur: 0
net.inet.tcp.bb.log_auto_all: 1
net.inet.tcp.bb.log_auto_mode: 4
net.inet.tcp.bb.log_auto_ratio: 1
net.inet.tcp.bb.disable_all: 0
net.inet.tcp.bb.log_version: 9
net.inet.tcp.bb.log_id_tcpcb_entries: 0
net.inet.tcp.bb.log_id_tcpcb_limit: 0
net.inet.tcp.bb.log_id_entries: 0
net.inet.tcp.bb.log_id_limit: 0
net.inet.tcp.bb.log_global_entries: 5016
net.inet.tcp.bb.log_global_limit: 5000000
net.inet.tcp.bb.log_session_limit: 5000
net.inet.tcp.bb.log_verbose: 0
net.inet.tcp.bb.tp.count: 0
net.inet.tcp.bb.tp.bbmode: 4
net.inet.tcp.bb.tp.number: 0
```

Using the **sysctl**-interface, BBLog can be enabled for TCP connections. The key **sysctl**-variables for this are **net.inet.tcp.bb.log_auto_all**, **net.inet.tcp.bb.log_auto_mode** and **net.inet.tcp.bb.log_auto_ratio**.

The first **sysctl**-variable to consider is **net.inet.tcp.bb.log_auto_all**. If this variable is set to 1 all connections will be considered for the BBLog ratio. If this value is set to zero, then only connections that have had a **TCP_LOGID** set (see below) will get the BBLog ratio applied to them. In most cases, where the **sysctl**-method is used to enable BBLog, the application probably does not have set a **TCP_LOGID**, so setting **net.inet.tcp.bb.log_auto_all** to 1 assures that every connection will be considered.

The next **sysctl**-variable to set is the **net.inet.tcp.bb.log_auto_ratio**. This value determines 1 in *n* (where *n* is the value provided by setting **net.inet.tcp.bb.log_auto_ratio**) connections will have BBLog enablement applied to them. So, for example, if **net.inet.tcp.bb.log_auto_ratio** is set to 100 then 1 in every 100 connections will have BBLog enabled upon them. If BBLog needs to be enabled for every connection **net.inet.tcp.bb.log_auto_ratio** needs to be set to 1.

The final **sysctl**-variablel to consider is **net.inet.tcp.bb.log_auto_mode**. The value is the numeric constant for the BBLog state. For TCP development, the default could be set to 4 for **TCP_LOG_STATE_CONTINUAL** to log every event that is generated by any connection for debugging purposes.

Some of the other items in the **sysctl**-variable can also be useful, the **net.inet.tcp.bb.log_session_limit** controls how many BBLog events a connection can collect before it has to do something with the data, i.e., either send it off to the collection system or recycle (overwrite) the events. The **net.inet.tcp.bb.log_global_limit** enforces a global system limit on how many total BBLog events the operating system will allow to be allocated.

The last three **sysctl**-variables are related to trace points. **net.inet.tcp.bb.tp.bbmode** specifies the BBLog state to be used if the trace point is triggered. **net.inet.tcp.bb.tp.**

`count` is the number of connections that are allowed to have the specified trace point triggered. For example, if set to 4, then 4 connections can trigger the trace point and after that no others will trigger that specific point (this is to limit the amount of BBLog events generated). `net.inet.tcp.bb.tp.number` specifies the trace point to be enabled.

**TCP Connection Specific Configuration via the Socket API**

The following `IPPROTO_TCP`-level socket options that can be used to control BBLog on an individual connection:

- `TCP_LOG` — This option sets the BBLog state on a connection. Any use of this socket option overrides any previous setting.
- `TCP_LOGID` — This option is passed a string that when set will be used to name the files generated by the `tcplog_dumper`. It associates the string as an "ID" to be associated with the connection. Note that multiple connections may use the same "ID" string. This is possible because the `tcplog_dumper` also incorporates the IP address and ports in the filename generated.
- `TCP_LOGBUF` — This socket option can be used to read data from the current connections logging buffer. Normally this is not used and instead `/dev/tcp_log` is read from by a general purpose tool such as the `tcplog_dumper` (which reads and stores the BBLogs). But this, as an alternative, allows a user process to collect a number of logs.
- `TCP_LOGDUMP` — This socket option directs the BBLog system to dump any records that are in queue on the connection to `/dev/tcp_log`. If no dump reason or ID has been given then the system default for the type of logging underway is used in any "reason" field inside the dump file.
- `TCP_LOGDUMPID` — This socket option, like `TCP_LOGDUMP`, directs the BBLog system to dump out any records to `/dev/tcp_log`, but in addition it specifies a specific user given "reason" for the output which will be included in the BBlog "reason" field.
- `TCP_LOG_TAG` — This option associates an additional "tag" in the form of a string with all BBLog records for this connection.

For example, if access to the source code of the program using a TCP connection is available, the BBLog state of the connection can be set to `TCP_LOG_STATE_CONTINUAL` using the `TCP_LOG` socket option:

```
#include <netinet/tcp_log_buf.h>

int err;
int log_state = TCP_LOG_STATE_CONTINUAL;
err = setsockopt(sd, IPPROTO_TCP, TCP_LOG, &log_state, sizeof(int));
```

This code can also be used for any other BBLog state mentioned earlier.
If no access to the source code is available, one can use with root privileges

```
tcpsso -i id TCP_LOG 4
```

where `id` is the `inp_gencnt`, which can be determined by running `sockstat -iPtcp. 4` is the numeric value of `TCP_LOG_STATE_CONTINUAL`.

## Generating BBLog Files

Before enabling BBLog on a specific TCP connection one needs to first make sure that the collection of BBLogs is taking place. FreeBSD has a tool designed for that called

**tcplog_dumper** which is available in the ports tree (**net/tcplog_dumper**). It can be installed by running with root privileges:

```
pkg install tcplog_dumper
```

Adding

```
tcplog_dumper_enable="YES"
```

to the file **/etc/rc.conf** will start the daemon automatically after the next reboot. It can also be started a daemon by manually running with root privileges:

```
tcplog_dumper -d
```

By default **tcplog_dumper** will collect BBLog's in the directory **/var/log/tcplog_dumps**. There are several other options which are supported including:

- **-J** — This option will cause the **tcplog_dumper** to output compressed files with **xz**.
- **-D directory path** — Store the files collected in the directory path specified, not the default. This can also be controlled by the **rc.conf** variable **tcplog_dumper_basedir**.

The **tcplog_dumper** will output pcapng (pcap next generation) files. pcapng supports storing meta information in addition to packet information. For **TCP_LOG_IN** and **TCP_LOG_OUT** events, **tcplog_dumper** generates an IP header from the event (so except for the source and destination IP address, the fields in the IP header might not be as they have been on the wire), uses the TCP header from the event (which means it is as the segment was on the wire) and adds a dummy payload of the correct length. For each TCP connection, **tcplog_dumper** creates a series of files and will put roughly 5000 BBLog events in each file numbered in sequence .0, .1, .2 etc. The following is an example of a series of 7 files for a single TCP connection::

```
[rrs]$ ls /var/log/tcplog_dumps/
UNKNOWN_18262_10.1.1.1_9999.0.pcapng  UNKNOWN_18262_10.1.1.1_9999.4.pcapng
UNKNOWN_18262_10.1.1.1_9999.1.pcapng  UNKNOWN_18262_10.1.1.1_9999.5.pcapng
UNKNOWN_18262_10.1.1.1_9999.2.pcapng  UNKNOWN_18262_10.1.1.1_9999.6.pcapng
UNKNOWN_18262_10.1.1.1_9999.3.pcapng records
```

So the **TCP_LOGID** was not set on the connection, one of the TCP ports was 18262, the other TCP port 9999 and the remote IPv4 address is 10.1.1.1.

Generating BBLog files from a core dump is currently being worked on. A debugger will be used to extract the information and provide it to t**cplog_dumper** for actually writing the BBLog files.

## Reading BBLog Files

There are two easily accessible tools that can read BBLog files.These are **read_bbrlog** and **wireshark**, both available as ports or packages.

### read_bbrlog

**read_bbrlog** is a small program that will read a series of BBLog files and display each log entry in text form. It needs to be given the prefix of the BBLog files as the input source and it finds all of the files associated with that tcp connection and prints out to **stdout** each event in text form. Note that there is also an option to redirect the output to a file (highly recommended since lots of data will be displayed). Here is an example on how to run **read_bbrlog**:

```
[rrs]$ read_bbrlog -i UNKNOWN_18262_10.1.1.1_9999 -o
my_output_file.txt -e Files:7 Processed 30964 records Saw
30964 records from stackid:3 total_missed:0 dups:0
```

In this case three options are used: **`-i input`** where the input argument is the base connection id, i.e., the text displayed by **`ls`** minus the **`.X.pcapng`**. The **`-o outfile`** to redirect output to the output file **`my_output_file.txt`** and finally the **`-e`** option which is typically used to put out "extended" output which is more verbose.

Here is a small clip from the file **`my_output_file.txt`** to give a flavor of the data presented. Note due to the large line length some of the data for display has been truncated off:

```
106565924 0 rack [50] PKT_OUT    Sent(0) 763046978:5  (PUS|ACK fas:0 bas:1) bw:208.00 bps(26)
                       avail:5 cw:14480 scw:14480 rw:65535 flt:0 (spo:64 ip:0)
106565979 0 rack [55] TCP_HYSTART  -- New round begins round:1 ends:763046983 cwnd:14480
106565982 0 rack [3] BBR_PACING_CALC Old rack burst mitigation len:5 slot:0 trperms:369
106565985 0 rack [3] TIMERSTAR  type:TLP(timer:4) srtt:39001 (rttvar:17063 * 4) rttmin:30000
106565986 0 rack [1] USERSEND   avail:5 pending:5 snd_una:763046978 snd_max:763046983 out:5
106565986 0 rack [0] TCP_LOG_PRU  pru_method:SEND (9) err:0
106607480 0 rack [2] IN         Ack:Normal 5 (PUS|ACK) off:32 out:5 lenin:5 avail:5 cw:14480
                                rw:4000512 una:763046978 ack:763046983
```

This shows that a 5 byte packet was sent at timemark 106565924 and sequence number 763046978. The congestion window at the time was 14480 bytes and the flight size (flt) was 0. No pacing was engaged. About 41 milliseconds (41,494 i.e. 106604046 - 106607480) an acknowledgement was received for those bytes.

## Wireshark

**`wireshark`** and **`tshark`** can also be used to display BBLog files. They only operate on individual files, not on a file series as **`read_bbrlog`** does. Currently no event specific information will be displayed. For **`TCP_LOG_IN`** and **`TCP_LOG_OUT`** events the BBLog information is shown in the Frame Information. For all other events, the BBLog information is directly shown.

**RANDALL STEWART** (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

**MICHAEL TÜXEN** (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.

PRACTICAL PORTS

# Developing Custom Ansible Modules

BY BENEDICT REUSCHLING

Ansible offers a lot of different modules and a typical user makes use of them without the need to ever write their own due to the sheer size of available modules. Even if the necessary functionality is not available in the **`ansible.builtin`** modules, the Ansible Galaxy offers plenty of third party modules from enthusiasts that extend the module count even more.

When the desired functionality is not covered by a single module or a combination of them, then you have to develop your own. Developers can chose to keep custom models local without having to publish them on the Internet or without Ansible Galaxy using them. Modules are commonly developed in Python, but other programming languages are possible when the module is not planned for submission into the official Ansible ecosystem.

Developers can chose to keep custom models local without having to publish them on the Internet or without Ansible Galaxy using them.

To test the module, install the **`ansible-core`** package, which helps by providing common code that Ansible uses internally. The custom module can then piggy-back onto much of the core Ansible functionality that existing modules use and is both reliable and stable.

## Example Module Using Shell Programming

We'll start with a simple example to understand the basics. Later, we will extend it to use Python for more functionality.

Description of our custom module: Our custom module called **`touch`** checks for a file in **`/tmp`** called **`BSD.txt`**. If it exists, the module returns **`true`** (state unchanged). If it does not exist, it creates that (empty) file and returns **`state: changed`**.

Custom modules are in a library directory next to the playbook that uses the module. Create that directory using mkdir:

```
mkdir library
```

Create a shell script in library that holds the module code:

```
touch library/touch
```

Enter the following code in library/touch as the module logic:

```
1  FILENAME=/tmp/BSD.txt
2  changed=false
3  msg=''
4  if [ ! -f ${FILENAME} ]; then
5      touch ${FILENAME}
6      msg="${FILENAME} created"
7      changed=true
8  fi
9  printf '{"changed": "%s", "msg": "%s"}' "$changed" "$msg"
```

First, we define some variables and set some default values. Line 4 checks if the file does not exist. If that is the case, we let the module create the file and update the **msg** variable. We need to notify Ansible about the changed state, so we return a variable called **changed** along with the updated message in line.

Create a playbook called **touch.yml** at the same location as the **library** directory. It looks like this:

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Run our custom touch module
      touch:
      register: result

    - debug: var=result
```

Note: We could execute the custom module against any remote nodes, not **localhost** alone. It's easier to test against **localhost** first during development.

Run the playbook like any other we've written before:

```
ansible-playbook touch.yml
```

## Running the Example Module

When the file **/tmp/BSD.txt** does not exist, the playbook output is:

```
PLAY [localhost] ***************************************

TASK [Run our custom touch module] *********************
changed: [localhost]

TASK [debug] *******************************************
ok: [localhost] => {
    "changed": true,
    "result": {
        "failed": false,
```

```
        "msg": "/tmp/BSD.txt created"
    }
}
```

When the file **/tmp/BSD.txt** exists (from a previous run), the output is:

```
PLAY [localhost] ***************************************

TASK [Run our custom touch module] ************************
ok: [localhost]

TASK [debug] ***************************************************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "msg": ""
    }
}
```

## Custom Modules in Python

What are the benefits of writing a module in Python, like the rest of the **ansible.builtin** modules? One benefit is that we can use the existing parsing library for the module parameters without having to reinvent our own. It's difficult in shell to define the name of each parameter in our own module. In Python, we can teach the module to accept some parameters as optional and require others as mandatory. Data types define what kind of inputs the module user must provide for each parameter. For example, a **dest**: parameter should be a path data type rather than an integer. Ansible provides some handy functionality to include in our script so that we can focus on our module's core functionality.

## The Ansiballz Framework

Modern Ansible modules use the Ansiballz framework. Unlike the Module Replacer, which were used by Ansible versions before 2.1, it uses real Python imports from **ansible/module_utils** instead of preprocessing the module.

Module Functionality: Ansiballz constructs a zip file. Contents:

- the module file
- **ansible/module_utils** files imported by the module
- boilerplate for the module parameters

The zip file is Base64 encoded and wrapped into a small Python script for decoding it. Next, Ansible copies it into the temp directory of the target node. When executing, the Ansible module script extracts the zip file and places itself in the temp dir, too. It then sets the **PTHONPATH** to find Python modules inside the zip and imports the Ansible module under the special **name**. Python then thinks it executes a regular script rather than importing a

> If I use a feature
> only twice per year,
> I may need to look
> it up again.

module. This allows Ansible to run both the wrapper script and the module's code in a single Python copy on the target host.

## Creating the Python Module

To create a module, use a **`venv`** or **`virtualenv`** for the development part. We start like before with a **`library`** directory where we create a new **`hello.py`** module with this content:

```
#!/usr/bin/env python3

from ansible.module_utils.basic import *

def main():
    module = AnsibleModule(argument_spec={})
    response = {"hello": "world!"}
    module.exit_json(changed=False, meta=response)

if name == "__main__":
    main()
```

**`import`** imports the Ansiballz framework to construct modules. It includes code constructs like argument parsing, file operations, and formatting return values as JSON.

## Executing the Python Module from a Playbook

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the Python module
      hello:
      register: result

    - debug: var=result
```

Again, we run the playbook like this: **`ansible-playbook hello.yml`**

```
PLAY [localhost] ***************************************

TASK [Testing the Python module] ***********************
ok: [localhost]

TASK [debug] *******************************************
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "meta": {
            "hello": "world!"
        }
    }
}
```

## Defining Module Parameters

The modules we used had taken parameters like **`path:`**, **`src:`**, or **`dest:`** to control the behavior of the module. Some of these parameters are essential for the module to function properly, while others were optional. In our own module, we want to control what parameters we take overall and which are required. Defining the data type makes our module robust against incorrect inputs.

The **`argument_spec`** provided to **`AnsibleModule`** defines the supported module arguments, as well as their type, defaults, and more.

Example parameter definition:

```
parameters = {
    'name': {"required": True, "type": 'str'},
    'age': {"required": False, "type": 'int', "default": 0},
    'homedir': {"required": False, "type": 'path'}
}
```

The required parameter **`name`** is of type string. Both **`age`** (an integer) and **`homedir`** (a path) are optional and if not defined, sets **`age`** to 0 by default. A new module that uses these parameter definitions calculates the result from passing two numbers and an optional math operator. When not provided, we assume an addition as default parameter. Create a new python file in **`library`** called **`calc.py`**:

```
#!/usr/bin/env python3
from ansible.module_utils.basic import AnsibleModule

def main():
    parameters = {
        "number1": {"required": True, "type": "int"},
        "number2": {"required": True, "type": "int"},
        "math_op": {"required": False, "type": "str", "default": "+"},
    }

    module = AnsibleModule(argument_spec=parameters)

    number1 = module.params["number1"]
    number2 = module.params["number2"]
    math_op = module.params["math_op"]

    if math_op == "+":
        result = number1 + number2

    output = {
        "result": result,
    }

    module.exit_json(changed=False, **output)

if __name__ == "__main__":
    main()
```

## The Playbook for the Module

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the calc module
      calc:
        number1: 4
        number2: 3
      register: result

    - debug: var=result
```

The **calc** module optionally takes a parameter **math_op**, but since we defined a default action (**+**) for it, the user can omit it in the playbook or on the commandline. The task that runs the module must specify the required parameters or the playbook will fail to execute.

## Running the calc Module

The relevant output of the playbook execution is below:

```
ok: [localhost] => {
    "result": {
        "changed": false,
        "failed": false,
        "result": 7
    }
}
```

We extend the example to properly handle +, -, *, /. The module returns **false** when it gets a **math_op** that is is different from the ones defined. Also, handling division by zero by returning "Invalid Operation" is a classic assignment for students since the dawn of time. I need to properly learn Python one day, but until then, my solution looks like this:

```
#!/usr/bin/env python3
    from ansible.module_utils.basic import AnsibleModule

def main():
    parameters = {
        "number1": {"required": True, "type": "int"},
        "number2": {"required": True, "type": "int"},
        "operation": {"required": False, "type": "str", "default": "+"},
}

    module = AnsibleModule(argument_spec=parameters)

    number1 = module.params["number1"]
    number2 = module.params["number2"]
    operation = module.params["operation"]
    result = ""
```

```
if operation == "+":
    result = number1 + number2
elif operation == "-":
    result = number1 - number2
elif operation == "*":
    result = number1 * number2
elif operation == "/":
    if number2 == 0:
        module.fail_json(msg="Invalid Operation")
    else:
        result = number1 / number2
    else:
        result = False

    output = {
        "result": result,
    }

    module.exit_json(changed=False, **output)

if __name__ == "__main__":
    main()
```

Testing our extended module is straightforward. Here is the test for division by zero:

```
---
- hosts: localhost
  gather_facts: false
  tasks:
    - name: Testing the calc module
      calc:
        number1: 4
        number2: 0
        map_op: '/'
      register: result

    - debug: var=result
```

Which results in the following expected output:

```
TASK [Testing the calc module] *************************************
fatal: [localhost]: FAILED! => {"changed": false, "msg": "Invalid Operation"}
```

## Conclusion

With these basics, its easy to get started on a custom module. Bear in mind that these modules need to run on different operating systems. Add extra checks to find out the availability of certain commands or let your module outright refuse to run in certain environments. Be as compatible as possible to increase the module's popularity and usefulness. There are not a lot of BSD-specific modules available. How about adding a bhyve module, or one that manages boot environments, the pf firewall or `rc.conf` entries? Plenty of options await the intrepid developer with a background in both Ansible and Python.

**References:**

- Ansible module architecture

---

**BENEDICT REUSCHLING** is a documentation committer in the FreeBSD project and member of the documentation engineering team. 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. Benedict is one of the hosts of the weekly bsdnow.tv podcast.

FreeBSD®

**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

FreeBSD FOUNDATION

# 2024 Events Calendar

## BSD Events taking place through October 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.

---



### EuroBSDCon 2024

September 19-22, 2024
Dublin, Ireland
https://2024.eurobsdcon.org/

EuroBSDCon is the International annual technical conference held in a different European country each year. It focuses on gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects. The FreeBSD Foundation is pleased to again be a Silver Sponsor.

---

### EuroBSDCon FreeBSD Developer Summit

September 19-20, 2024
Dublin, Ireland
https://wiki.freebsd.org/DevSummit/202409

Join us for talks and discussion groups on day 1, followed by a hackathon on day 2.

---

### All Things Open

October 27-29, 2024
Raleigh, NC
https://2024.allthingsopen.org/

All Things Open is the largest open source/open tech/open web conference on the East Coast and one of the largest in the United States. It regularly hosts some of the most well-known experts in the world, as well as nearly every major technology company. FreeBSD is proud to be a non-profit partner for this year's All Things Open.

---