BSD MAGAZINE

FOR NOVICE AND ADVANCED USERS

Creating and Scaling Your Applications Using ContainerPilot Pattern

Build FreeBSD 12 For RaspberryPi 3 With Crochet

Fluentd For Centralizing Log

MINIX 3: A Promising Diamond in the Rough

Interview with Prof. Andrew Tanenbaum, The Creator of MINIX3

Introducing The TrueNAS Unified Storage X10

Implementing an Enigma Machine Simulator as a Character Device

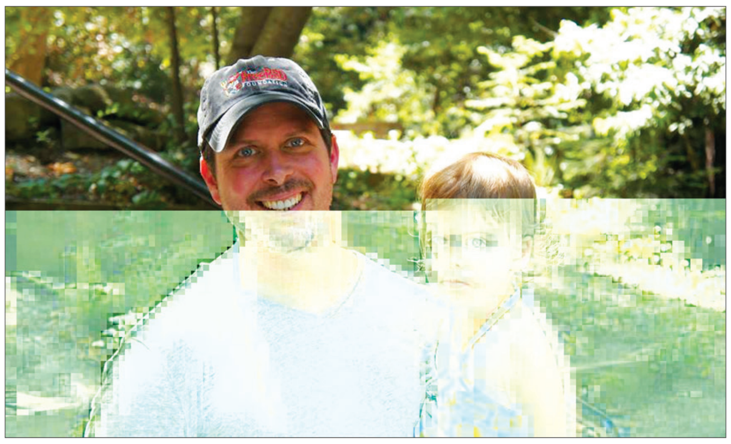
VOL 11 NO 06
ISSUE 06/2017 (94)
ISSN 1898-9144

FREENAS MINI
STORAGE APPLIANCE

IT *SAVES* YOUR LIFE.

HOW IMPORTANT IS YOUR DATA?

Years of family photos. Your entire music and movie collection. Office documents you've put hours of work into. Backups for every computer you own. We ask again, *how important is your data?*



Example of one-bit corruption

NOW IMAGINE LOSING IT ALL

Losing one bit - that's all it takes. One single bit, and your file is gone.

The worst part? **You won't know until you absolutely need that file again.**

THE SOLUTION

The FreeNAS Mini has emerged as the clear choice to save your digital life. **No other NAS in its class offers ECC (error correcting code) memory and ZFS bitrot protection to ensure data always reaches disk without corruption and *never degrades over time*.**

No other NAS combines the inherent data integrity and security of the ZFS filesystem with fast on-disk encryption. No other NAS provides comparable power and flexibility. The FreeNAS Mini is, hands-down, the best home and small office storage appliance you can buy on the market. **When it comes to saving your important data, there simply is no other solution.**

The Mini boasts these state-of-the-art features:

- 8-core 2.4GHz Intel® Atom™ processor
- Up to 16TB of storage capacity
- 16GB of ECC memory (with the option to upgrade to 32GB)
- 2 x 1 Gigabit network controllers
- Remote management port (IPMI)
- Tool-less design; hot swappable drive trays
- FreeNAS installed and configured

iXsystems™

http://www.iXsystems.com/mini

intel® inside™ ATOM™

FREENAS CERTIFIED STORAGE

With over six million downloads, FreeNAS is undisputedly *the* most popular storage operating system in the world.

Sure, you could build your own FreeNAS system: research every hardware option, order all the parts, wait for everything to ship and arrive, vent at customer service because it *hasn't*, and finally build it yourself while hoping everything fits - only to install the software and discover that the system you spent *days* agonizing over **isn't even compatible**. Or...

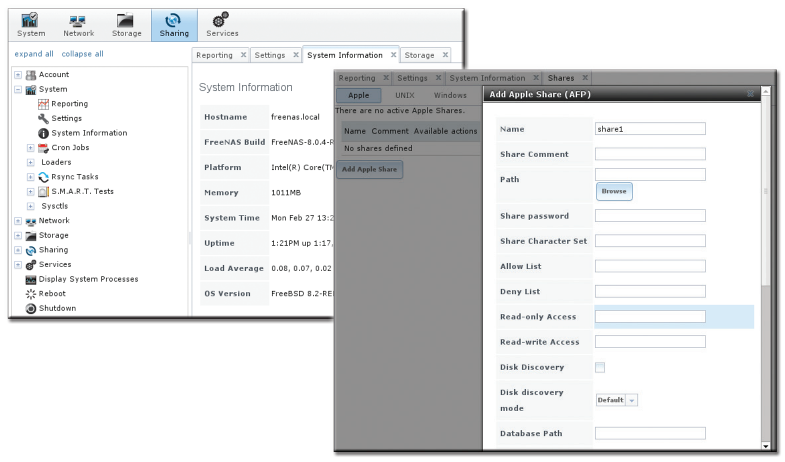
MAKE IT EASY ON YOURSELF

As the sponsors and lead developers of the FreeNAS project, iXsystems has combined over 20 years of hardware experience with our FreeNAS expertise to bring you FreeNAS Certified Storage. **We make it easy to enjoy all the benefits of FreeNAS without the headache of building, setting up, configuring, and supporting it yourself.** As one of the leaders in the storage industry, you know that you're getting the best combination of hardware designed for optimal performance with FreeNAS.

Every FreeNAS server we ship is...

» Custom built and optimized for your use case
» Installed, configured, tested, and guaranteed to work out of the box
» Supported by the Silicon Valley team that designed and built it
» Backed by a 3 years parts and labor limited warranty

As one of the leaders in the storage industry, you know that you're getting the best combination of hardware designed for optimal performance with FreeNAS. **Contact us today for a FREE Risk Elimination Consultation with one of our FreeNAS experts.** Remember, every purchase directly supports the FreeNAS project so we can continue adding features and improvements to the software for years to come. **And really - why would you buy a FreeNAS server from *anyone* else?**



System Information

Hostname	freenas.local
FreeNAS Build	FreeNAS-8.0.4-R
Platform	Intel(R) Core(TM
Memory	1011MB
System Time	Mon Feb 27 13:2
Uptime	1:21PM up 1:17,
Load Average	0.08, 0.07, 0.02
OS Version	FreeBSD 8.2-RE

There are no active Apple Shares.

Name Comment Available actions
No shares defined

Add Apple Share (AFP)

Name	share1
Share Comment	
Path	
Share password	
Share Character Set	
Allow List	
Deny List	
Read-only Access	
Read-write Access	
Disk Discovery	☐
Disk discovery mode	Default
Database Path	

FreeNAS 1U

- Intel® Xeon® Processor E3-1200v2 Family
- Up to 16TB of storage capacity
- 16GB ECC memory (upgradable to 32GB)
- 2 x 10/100/1000 Gigabit Ethernet controllers
- Redundant power supply

FreeNAS 2U

- 2x Intel® Xeon® Processors E5-2600v2 Family
- Up to 48TB of storage capacity
- 32GB ECC memory (upgradable to 128GB)
- 4 x 1GbE Network interface (Onboard) - (Upgradable to 2 x 10 Gigabit Interface)
- Redundant Power Supply

intel® inside™ XEON®

http://www.iXsystems.com/storage/freenas-certified-storage/

EDITOR'S WORD

MAGAZINE BSD

Dear Readers,

I hope this finds you well. When I started preparing this issue, it didn't occur to me that we would collect so many fantastic and unique articles written by great authors. The most amazing thing when you deal with such nice people, who are always on time and always there ready to help is the joy and satisfaction felt from their hard work. I am proud of them and happy that we can continually impart knowledge to you, our readers. I think that the joy of doing anything is this feeling which is needed to create a good thing in your life, not only in your professional life, but also in your everyday existence. I think that creates a special moment when you can offer a smile to other people and encourage them to see the world around them in brighter colours.

And now, let's look at what is featured in this month's issue. I would recommend that you read all the articles. We hope you find those which are most presently useful for you and can help you in your daily tasks. You will find more details in the next page, the Table of Contents. However, I will quickly review them all. In June, we had great new events. The TrueNAS X10 was released and Steve Wong wrote about it. We decided to publish a short article which presents new features of the TrueNAS X 10. Also, it is worth mentioning that the FreeNAS 11.0 was released as well. We all enjoy these new releases. Carlos Antonio Neira Bustos's article presents a ContainerPilot. You will learn how to create and scale your application using it. This time, Abdorrahman Homaei will show you how to build FreeBSD 12 for RaspberryPi 3 with Crochet. Rafael Santiago will focus on the source code related to the FreeBSD. He wants to introduce you to the main aspects of device driver programming for FreeBSD by using more than a simple Hello World sample. Therefore, at the end of the article you should be able to have a historic cryptographic device within their /dev. Tips on how to produce a multi-platform code base for device drivers will also be presented. That is not all you will read in his article. Moreover, the article on MINIX3 written by Jean-Baptiste Boric is a must read. When reading it, do not miss the interview with Professor Andrew Tanenbaum, the creator of MINIX3. I enjoyed reading it. Andrey Ferriyan sent an article on Fluentd For Centralizing Log. In his article, Andrey talks about how to manage diverse and disparate logson FreeBSD servers. Also, I cannot forget about my dearest Rob Somerville who every month publishes a wonderful column. This time, he will share his thoughts on: Amid the fever of "fake news" and multiple governments' desire to limit encryption in light of more terrorist atrocities, is the core principle of social media and the World Wide Web – that of freedom of expression – coming to an end?

What can I say? I just can't thank you enough. You make my day so much better every time you write to me. And I know that many readers not only enjoy the read but also look forward to reading more of your articles.

I would also like to thank the BSD team members for reviewing and proofreading , and iXsystems for their constant support and time to make this edition a success.

And now let's read the articles.
Enjoy!

Best regards
Ewa & The BSD Team

IN BRIEF

iXsystems' TrueNAS X10 Breaks New Ground With Entry Level Enterprise-Class Unified Storage Solution

iXsystems, the industry leader in storage and servers driven by Open-Source, announced the release of the TrueNAS X10. The TrueNAS X10 is a cost-effective enterprise storage solution that is designed to help small and midsized businesses modernize their storage infrastructures. For years, customers have struggled with their storage infrastructures. They have bought costly Enterprise-Class storages or do away with the idea of purchasing an enterprise-class storage due to its high cost. Companies unable to invest in enterprise-class storage are often forced to use legacy SAN/NAS systems, deploy consumer NAS systems, use direct access storage (DAS), or build their own software-defined storage (SDS) systems. The TrueNAS X10 clears this barrier to entry by providing Enterprise-Class storage for SMBs and others that are challenging the Dell™ EMC™ VNXe, Dell™ EMC™ Unity, HPE™ MSA 2040, and NetApp™ FAS2600 series of products with its functionality and price point. The TrueNAS X10 comes in a dense 2U form factor accommodating up to 12 disk drives. It enables you to reduce space, power and cooling costs and respond to the ever-changing business requirements. It is optimized for SMBs, remote offices, and enterprises of all sizes. It lets you start small and grow to nearly 400 TB as your needs change. The TrueNAS X10 can be ordered and will begin shipping in mid-July 2017. The standard lead time is three weeks for all TrueNAS systems. However, pre-built 20 TB, 60 TB, and 100 TB configurations of the TrueNAS X10 are available with a one week lead time.
Source: https://www.ixsystems.com/blog/truenas-x10/

FreeNAS 11.0 is Now Here

After several FreeNAS release candidates, FreeNAS 11.0 was released today. This version brings new virtualization and object storage features to the World's Most Popular Open-Source Storage Operating System. FreeNAS 11.0 adds bhyve virtual machines to its popular SAN/NAS, jails, and plugins, letting you use host web-scale VMs on your FreeNAS box. It also gives users S3-compatible object storage services, which turns your FreeNAS box into an S3-compatible server, letting you avoid reliance on the cloud.
FreeNAS 11.0 is based on FreeBSD 11-STABLE, which adds the latest drivers and performance improvements. Users will benefit from the overall systematic, architectural, and performance improvements. Testing indicates that the kernel of FreeNAS 11.0 is 20% faster than the kernel of FreeNAS 9.10.
FreeNAS 11.0 also introduces the beta version of a new administration GUI. The new GUI is based on the popular Angular framework and the FreeNAS team expects the GUI to be themeable and feature complete by 11.1. The new GUI follows the same flow as the existing GUI, but looks better. For now, the FreeNAS team has released it in beta form to get input from the FreeNAS community. The new GUI, as well as the classic GUI, are selectable from the login screen.
Also new in FreeNAS 11 is an Alert Service page which configures the system to send critical alerts from FreeNAS to other applications and services such as Slack, PagerDuty, AWS, Hipchat, InfluxDB, Mattermost, OpsGenie, and VictorOps. FreeNAS 11.0 has an improved Services menu that adds the ability to manage which services and applications are started at boot.
To download FreeNAS and sign-up for the FreeNAS Newsletter, visit freenas.org/download.
Source: http://www.freenas.org/blog/freenas-11-0/

The Second BETA Build for the FreeBSD 11.1 Release

The second BETA build for the FreeBSD 11.1 release cycle is now available. ISO images for the amd64, armv6, i386, aarch64, powerpc, powerpc64 and sparc64 architectures are available on most of our FreeBSD mirror sites.
Source: https://www.freebsd.org/news/newsflash.html

DigitalOcean Releases Free Cloud Firewalls Service to Strengthen Security for Large-Scale Applications

DigitalOcean, the cloud for developers, launched Cloud Firewalls, a free service that secures Droplets (cloud servers) by reducing the surface area of a potential attack. Developers can deploy the service in seconds without installing or configuring any software, and define what ports are visible on their Droplets to minimize risk. Along with DigitalOcean's free Monitoring Service launched in April 2017, Cloud Firewalls are key parts of DigitalOcean's continuous effort to add value back to developers by allowing them to deploy and scale applications of any size.

Developers with a large number of Droplets will find it much easier to secure their applications with Cloud Firewalls. It scales automatically from one Droplet to thousands, and provides a central location for defining and applying access rules to prevent unauthorized traffic from reaching them. Users can leverage tagging to a group and organize any number of Droplets, and use them to define how each group of Droplets is secured.

Cloud Firewalls give users the ability to whitelist which ports are open and which IP ranges, tags, Droplets or load balancers can access them. Users can easily configure the service and quickly through the dashboard or on the command line with doctl. They can also leverage DigitalOcean's API to automate tasks and build integrations. Official client libraries are available in Go and Ruby. Rules can be changed in one place and instantly applied to every Droplet that is tagged. The service is available in every region to all Droplet customers at no additional cost.

Source: *https://www.digitalocean.com/company/press/releases/digitalocean-releases-free-cloud-firewalls-service/*

KomodoSec Offers Free Reports on Corporate Cyberattack Surface

Komodo Consulting, a leading cyber-security consulting company, recently launched its newest service from its research arm, Peta.AI. Companies can receive free and customized reports that illustrate how individual organizations' cyber-attack surfaces appear to hackers. The corporate cyber-attack surface report reveals the client's exposure to cyber threats based on several indicators - geolocation, IP addresses, open and sensitive ports, vulnerable apps, compromised hosts, and leaked accounts - compiled from Peta.AI's Open Internet, Deep Web and Dark Net research. *Companies can visit https://www.peta.ai/compare-yourself/ to request a copy of their report.*

NetBSD 8.0 Release Process Underway

If you've been reading source-changes@, you likely noticed the recent creation of the netbsd-8 branch. If you haven't been reading source-changes@, here's some news: the netbsd-8 branch has been created, signaling the beginning of the release process for NetBSD 8.0.

We don't have a strict timeline for the 8.0 release, but things are looking pretty good at the moment, and we expect this release to happen in a shorter amount of time than the last couple major releases did.

At this point, we would love for folks to test out netbsd-8 and let us know how it goes. A couple of major improvements since 7.0 are the addition of USB 3 support and an overhaul of the audio subsystem, including an in-kernel mixer. Feedback about these areas is particularly desired.

To download the latest binaries built from the netbsd-8 branch, head to
http://daily-builds.NetBSD.org/pub/NetBSD-daily/netbsd-8/

Source: *https://blog.netbsd.org/tnf/entry/netbsd_8_0_release_process*

Phoronix Test Suite 7.2.0 Officially Released

Just days after celebrating nine years since the Phoronix Test Suite 1.0 release, Phoronix Test Suite 7.2-Trysil is now officially available. Phoronix Test Suite 7.2.0 represents the latest quarterly feature update to our cross-platform, open-source benchmarking software. Phoronix Test Suite 7.2 features a number of result parser improvements, various minor enhancements to the pts_Graph subsystem, Phoromatic support for setting a run-priority on test schedules, new sub-command helpers, improvements for running the Phoromatic client on macOS, improvements to the perf-per-dollar module, Phodevi software/hardware detection improvements, and a variety of other improvements. Phoronix Test Suite 7.2 is available via GitHub and Phoronix-Test-Suite.com.

Source: *http://www.phoronix.com/scan.php?page=news_item&px=Phoronix-Test-Suite-7.2.0*

Introducing The TrueNAS Unified Storage X10. Part 1

Steve Wong, Director of Product Management

Steve Wong is the director of product management at iXsystems. He is a senior level professional with over 20 years of experience in the fields of data communications, enterprise storage, networking, telecommunications, brand marketing, publishing, e-commerce and consumer package goods. Prior to iXsystems, Wong worked at SerialTek, Hitachi Data Systems, ClearSight Networks, Finisar, Anritsu and Mattel. He began his career in investment banking at Bear Stearns and then served as a member of technical staff at AT&T Bell Laboratories. Wong holds a BA from New York University and a MBA from the Kellogg Graduate School of Management, Northwestern University.

It was just three years ago in August 2014 when iXsystems introduced the TrueNAS Z series product line of storage appliance platforms designed for organizations needing enterprise-class storage systems. TrueNAS is based on FreeNAS, the world's #1 Open-Source software-defined storage operating system. FreeNAS has the unique benefit of tens of thousands of people around the world helping in QA and providing extensive input into each successive release of the software.

TrueNAS provided a unified storage array packed with enterprise-grade capabilities like VMware, Citrix, and Veeam certifications, integration with public clouds, capacity-efficient features like block-level inline compression, deduplication, and thin provisioning as well as other enterprise features like snapshots, replication, and data at rest encryption.

Ever since the introduction of the TrueNAS Z products, customers have asked us for an entry-class TrueNAS appliance. I'd like to announce the arrival of the most cost-effective storage available in the market, the TrueNAS Unified Storage X10. It has a street price of under $10,000 for 20TB of raw capacity, the capabilities that exists across the entire TrueNAS product portfolio are also in the TrueNAS X10. This is no TrueNAS "light" product – rather it extends the TrueNAS product line.

Technical Overview

The TrueNAS X10 is available in a 2U, 12 x 3.5-inch SAS form factor. It supports up to a maximum of 36 front-loading, hot-pluggable drives through the use of two ES12 (12-bay) expansion shelves. Its maximum raw capacity is 360TB and utilizes enterprise-class dual-ported SAS3 drives. Furthermore, the TrueNAS X10 is a hybrid-class array, meaning that it combines RAM and flash SSDs to provide performance acceleration in the form of read and write cache extensions. All TrueNAS arrays can make use of these cache extensions to increase performance and reduce latency.

The TrueNAS X10 incorporates advanced components which provide the building blocks for a modern enterprise-class solution. Each storage controller is anchored by a power-efficient Intel Xeon D-1531 processor running at 2.2Ghz. This advanced processor is a high-performance systems-on-a-chip (SOC) with 6 cores and is built on top of a 14nm lithography technology. The Thermal Design Power (TDP) value is only 45W, so it consumes less power than the lowest TrueNAS Z product. An M.2 mSATA SSD device is used to boot the storage operating system. The use of error correcting 2133MHz DDR4 ECC SODIMM modules reduces the potential for in-memory data corruption.

The native PCI Express bus is PCI Express (PCIe) Gen 3.0. The storage server connects to storage through a LSI (12 Gb/s) integrated SAS3 controller and expander. The TrueNAS X10 comes standard with dual-integrated LAN GbE ports for data access. Customers can upgrade to 10GbE connectivity if more throughput is required through a PCIe x8 slot located in each controller. Electrical and optical interfaces are both supported. Remote management is provided by a dedicated GbE port through a custom-built BMC module in each storage controller.

Like all products in the TrueNAS family, the TrueNAS X10 is available in a single-controller or a dual-controller configuration. For customers requiring high availability (HA), the dual-controller configuration is a requirement. Customers with financial constraints may opt for the single-controller version initially and then upgrade to a dual-configuration at a later point when budget permits.

The TrueNAS X10 is smaller and greener than the original entry storage array, the TrueNAS Z20. The TrueNAS X10's storage controller is 10.9" in length, 8.3" width, and 1.5" (height). This is roughly the size of a ream of paper. Power consumption is less than 40% of the Z20, yet the TrueNAS X10 is one-third smaller than a Z20. It conforms to the 80Plus Gold standard.

The TrueNAS X10 is a unified storage platform supporting many file, block and object protocols including SMB, NFS, iSCSI, AFP and WebDAV. Also supported is file syncing to the Amazon S3 cloud.

Well, that is a quick rundown of the technical merits of the new TrueNAS X10 from iXsystems. I'll be back shortly with part 2 of this blog to discuss the business value offered by the TrueNAS X10, including use cases and applications.

BSD Certification

The BSD Certification Group Inc. (BSDCG) is a non-profit organization committed to creating and maintaining a global certification standard for system administration on BSD based operating systems.

WHAT CERTIFICATIONS ARE AVAILABLE?

BSDA: Entry-level certification suited for candidates with a general Unix background and at least six months of experience with BSD systems.

BSDP: Advanced certification for senior system administrators with at least three years of experience on BSD systems. Successful BSDP candidates are able to demonstrate strong to expert skills in BSD Unix system administration.

WHERE CAN I GET CERTIFIED?

We're pleased to announce that after 7 months of negotiations and the work required to make the exam available in a computer based format, that the BSDA exam is now available at several hundred testing centers around the world. Paper based BSDA exams cost $75 USD. Computer based BSDA exams cost $150 USD. The price of the BSDP exams are yet to be determined.

Payments are made through our registration website:
https://register.bsdcertification.org//register/payment

WHERE CAN I GET MORE INFORMATION?

More information and links to our mailing lists, LinkedIn groups, and Facebook group are available at our website:
http://www.bsdcertification.org

Registration for upcoming exam events is available at our registration website:
https://register.bsdcertification.org//register/get-a-bsdcg-id

Creating and Scaling Your Applications Using ContainerPilot

What is ContainerPilot?

ContainerPilot is a micro-orchestrator that implements the autopilot design pattern for applications that are both self-operating and self-managed. The containers hosting the applications adapt themselves to the changes in their environment, and coordinate their actions with other containers through global shared state. ContainerPilot was created by Joyent, a company which specializes in cloud computing. Joyent are the creators of the SmartOS, an illumos distribution which was a long time ago opensolaris. SmartOS uses zones which are now what we called containers. Zones is a battle tested technology that has been available for several years. It is now more relevant than ever if you plan to use containers https://www.joyent.com/smartos. Here is Joyent's definition:

"ContainerPilot is an application-centric micro-orchestrator. It automates many of the operational tasks related to configuring a container as you start it, re-configuring it as you scale it or other containers around it, health-checking, and at other times during the container's lifecycle. ContainerPilot is written in Go, and runs inside the container with the main application."

Deploying containerized applications poses a challenge as the amount of ball juggling you will need to perform grows exponentially with the number of containers you need to deploy. In cases where you have dependencies in your applications, containers need to know if they can communicate with other containers that implement functionality they need. For example, your container may depend on another container that manages data persistence, or it may simply need to pull data from a service running in another container. If your containers are not self-aware, you will need to do the orchestration yourself which is not desirable.

Container autopilot uses *docker* for containerization, but you could use Triton in Joyent's public cloud or install Triton if you have the hardware and run your Triton instance. But, what is Triton? Triton is a complete cloud management solution for server and network virtualization. You can create your own cloud using containers. Triton is what powers Joyent's compute service. Triton Compute Service provides three classes of compute instances: Docker containers, infrastructure containers, and hardware virtual machines. In our case, we are only interested in Docker containers. In this article, we will use *docker-compose,* but the example is ready to start using Triton with Docker containers as well. For more information to get acquainted with the *docker*, visit https://docs.docker.com/engine/installation/. Now, let us see how both container products are defined by their creators. This is what the Docker homepage says:

"Docker is the world's leading software container platform. Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers. Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density. Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux and Windows Server apps."

Joyent's definition of Triton:

"So it gives me great pleasure to introduce Triton to the world — a piece of (open-source!) engineering brought to you by a cast of thousands, over the course of decades. In a sentence (albeit a wordy one), Triton lets you run secure Linux containers directly on bare metal via an elastic Docker host that offers tightly integrated software-defined networking. "

The Docker Engine client runs natively on Linux, macOS, and Windows. Since 2015, you can also run Docker in FreeBSD. For Triton, Joyent's implemented a docker's remote API using Node.js, an asynchronous event driven JavaScript runtime. However, why in the Node.js instead of go ? It happens that Joyent is the corporate steward of Node.js . Thus, Docker is available in SmartOS, but it is spelled Triton (it is a very simple way to say it, but a whole article is needed to do justice to Triton) you need to follow this guide https://docs.joyent.com/public-cloud/getting-started

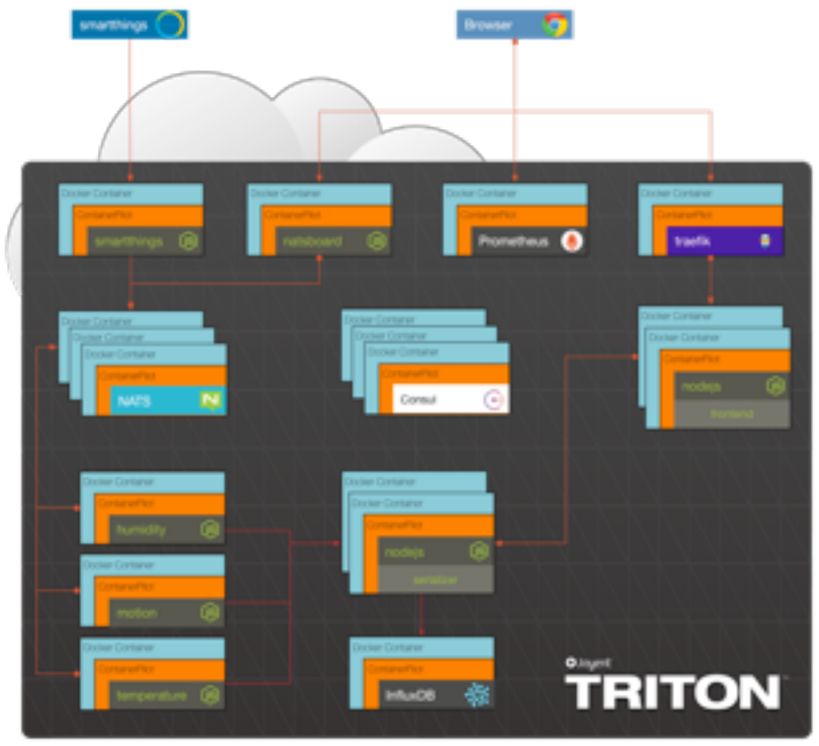
Example project

For purposes of this article, we will use Docker and an example repository which implements the container autopilot pattern. To start working with this example project, I will assume some basic command line familiarity. Also, you will need a platform that runs Docker. If you plan to use FreeBSD, installation instructions can be found in wiki https://wiki.freebsd.org/Docker . I will assume that you know docker-compose notation. If you don't, click on the following link for more information: https://docs.docker.com/compose/gettingstarted/#step-3-define-services-in-a-compose-file

The example project we are going to use to get acquainted with the container autopilot pattern is called "Node.js micro-services in Containers with Container Pilot". The author is called Wyatt Preul (https://github.com/geek). I have an old version of that repository at https://github.com/cneira/nodejs-example.git which I will use in this article.

The main idea of this project is to create micro-services that will produce and consume data that will be displayed in a frontend web application. The orchestration between these micro-services will be performed by Container Autopilot.

Here is the architecture that makes up this project.



Let's start analyzing this project on how it implements the container autopilot pattern. First, let's take a look at the docker-compose.yml or the local-compose.yml. The only difference is that local-compose.yml will spin containers in your machine but the docker-compose.yml file will use Triton and spin the containers in Joyent's Cloud. If you sign up, you will get $250 worth of compute/storage in Joyent's public cloud so that you can get the feel for the service, and test this "new" technology. Here is the URL: https://sso.joyent.com/signup.

```
consul:
       image: 
autopilotpattern/consul:0.7.2-r0.8
       restart: always
       dns:
         - 127.0.0.1
       labels:
         - triton.cns.services=consul
       ports:
         - "8500:8500"
       env_file:
         - _env
       command: >
         /usr/local/bin/containerpilot
         /bin/consul agent -server
           -config-dir=/etc/consul
           -log-level=err
           -ui-dir /ui
     nats:
         image: 
autopilotpattern/nats:0.9.6-r1.0.0
         restart: always
         env_file:
           - _env
     natsboard:
         image: d0cker/natsboard
         restart: always
         ports:
           - "3000:3000"
           - "3001:3001"
         env_file:
           - _env
     prometheus:
         image: 
autopilotpattern/prometheus:1.3.0r1.0
         mem_limit: 128m
         restart: always
         ports:
           - "9090:9090"
         env_file:
           - _env
     influxdb:
       image: 
autopilotpattern/influxdb:1.1.1
       restart: always
       env_file:
         - _env
       environment:
         - ADMIN_USER=root
         - INFLUXDB_INIT_PWD=root123
         - INFLUXDB_ADMIN_ENABLED=true
         - INFLUXDB_REPORTING_DISABLED=true
         - 
INFLUXDB_DATA_QUERY_LOG_ENABLED=false
         - INFLUXDB_HTTP_LOG_ENABLED=false
         - 
INFLUXDB_CONTINUOUS_QUERIES_LOG_ENABLED=fals
e
     traefik:
         image: d0cker/traefik:1.0.0
         labels:
           - triton.cns.services=ui
         ports:
           - "80:80"
           - "8080:8080"
         env_file:
           - _env
         restart: always
     serializer:
       image: d0cker/serializer:6.2.0
       env_file:
         - _env
       environment:
         - INFLUXDB_USER=root
         - INFLUXDB_PWD=root123
       expose:
         - "80"
       restart: always
     frontend:
       image: d0cker/frontend:6.0.4
       env_file:
         - _env
       expose:
         - "80"
       restart: always
     smartthings:
       image: d0cker/smartthings:8.0.0
       labels:
         - triton.cns.services=smartthings
       ports:
         - "80:80"
       env_file:
         - _env
       environment:
         - FAKE_MODE=true
       restart: always
     humidity:
       image: d0cker/sensor:4.0.0
       env_file:
         - _env
       environment:
         - SENSOR_TYPE=humidity
       restart: always
     motion:
       image: d0cker/sensor:4.0.0
       env_file:
         - _env
       environment:
         - SENSOR_TYPE=motion
       restart: always
     temperature:
       image: d0cker/sensor:4.0.0
```

```
        env_file:
          - _env
        environment:
          - SENSOR_TYPE=temperature
        restart: always
```

ContainerPilot needs to wrap your application so that it can pass signals to it and receive its exit code. To do that, you need to use ContainerPilot as a prefix command or entry point. In the docker-compose.yml file, we see the line that accomplished this. That means Consul will be managed by ContainerPilot. Consul is a tool for discovering and configuring services in your infrastructure. It's used as a service registry where your micro-services will advertise themselves and their health status. For more info on consul, visit its project page: https://www.consul.io/intro/index.html

```
  command: >
    /usr/local/bin/
containerpilot
    /bin/consul agent -server
      -config-dir=/etc/consul
      -log-level=err
      -ui-dir /ui
```

Your applications don't need to be aware of ContainerPilot. It adds to your application the following features right away.

- Service discovery
- Lifecycle
- Health checks
- preStart, preStop, & postStop
- Signal handling
- Periodic tasks
- Telemetry
- Coprocesses

Configuration is specified in a file called `containerpilot.json`. In there, you could define the features you want to utilize for that container. Let's check a `containerpilot.json` for the sensor docker image.

```
{
       "consul": "localhost:8500",
       "services": [
         {
           "name": "{{.SENSOR_TYPE}}",
           "health": "/usr/bin/curl -o /
dev/null --fail -s http://127.0.0.1:
{{.PORT}}/check-it-out",
           "poll": 3,
           "ttl": 10,
           "port": {{.PORT}}
         }
       ],
       "coprocesses": [
         {
           "command": ["/usr/local/bin/
consul", "agent",
                       "-data-dir=/data",
                       "-config-dir=/
config",
                       "-log-level=err",
                       "-rejoin",
                       "-retry-join",
"{{ if .CONSUL_HOST }}{{ .CONSUL_HOST }}{{
else }}consul{{ end }}",
                       "-retry-max", "10",
                       "-retry-interval",
"10s"],
           "restarts": "unlimited"
         }
       ],
       "backends": [
         {
           "name": "serializer",
           "poll": 3,
           "onChange": "pkill -SIGHUP
node"
         }
       ]
     }
```

Let's describe sections of this `containerpilot.json` config file.
"services"

- Name: This refers to the name of the service as it will appear in Consul. Each instance of the service will have a unique ID made up of the name+hostname of the container.
- Port: it is the port the service will advertise to Consul.
- health is the executable (and its arguments) used to check the health of a service.
- interfaces is an optional single or array of interface specifications. If given, the IP of the service will be obtained from the first interface specification that matches. (Default value is ["eth0:inet"]). The value that ContainerPilot uses for the IP address of the interface will be set as an environment variable with the name CONTAINERPILOT_{SERVICE_NAME}_IP. See the configuration template below.
- poll is the time in seconds between polling for health checks.
- ttl is the time-to-live of a successful health check. This should be longer than the polling rate so that the polling process and the TTL aren't racing; otherwise, Consul will mark the service as unhealthy.

"coprocesses". Coprocesses are processes that run alongside the main application, in this case:

- command is the executable (and its arguments) that will run when the coprocess executes.
- name is a friendly name given to the coprocess for logging purposes. This has no effect on the coprocess execution. This value is optional, and defaults to the command if not given.
- restarts is the number of times a coprocess will be restarted if it exits. Supports any non-negative numeric value (ex. 0, 1) or the strings "unlimited" or "never". This value is optional and defaults to "never".

"backends"

- name is the name of a backend service that this container depends on, as it will appear in Consul.
- poll is the time in seconds between polling for changes.
- onChange is the executable (and its arguments) that is called when there is a change in the list of IPs and ports for this backend.
- timeout is a value to wait before forcibly killing the onChange handler. Handlers killed in this way are terminated immediately (SIGKILL) without an opportunity to clean up their state. The minimum timeout is 1ms (see the golang ParseDuration docs for this format). This field is optional and defaults to be equal to the poll time.

But what about telemetry? This project adds telemetry to the frontend container. The container image has the following containerautopilot.json config.

```
{
       "consul": "localhost:8500",
       "services": [
         {
           "name": "frontend",
           "port": {{.PORT}},
           "health": "/usr/bin/curl -o /dev/null --fail -s http://localhost:{{.PORT}}/
check-it-out",
           "poll": 3,
           "ttl": 10
         }
       ],
       "coprocesses": [
         {
           "command": ["/usr/local/bin/consul", "agent",
                       "-data-dir=/data",
                       "-config-dir=/config",
                       "-log-level=err",
                       "-rejoin",
                       "-retry-join", "{{ if .CONSUL_HOST }}{{ .CONSUL_HOST }}{{ else }}
consul{{ end }}",
                       "-retry-max", "10",
                       "-retry-interval", "10s"],
           "restarts": "unlimited"
         }
       ],
       "backends": [
         {
           "name": "serializer",
           "poll": 3,
           "onChange": "pkill -SIGHUP node"
         }
       ],
       "telemetry": {
         "port": 9090,
         "tags": ["op"],
         "sensors": [
           {
             "namespace": "containerpilot",
             "subsystem": "frontend",
             "name": "free_memory",
             "help": "Frontend Free Memory",
             "type": "counter",
             "poll": 5,
             "check": ["/bin/memory.sh"],
             "timeout": "5s"
           }
         ]
       }
     }
```

- port is the port where the telemetry service will advertise to the discovery service. (Default value is 9090.)

- interfaces is an optional single or array of interface specifications. If given, the IP of the service will be obtained from the first interface specification that matches. (Default value is ["eth0:inet"])

- tags is an optional array of tags. If the discovery service supports it (Consul does), the service will register itself with these tags.

- sensors is an optional array of sensor configurations. If no sensors are provided, then the telemetry endpoint will still be exposed and will show only telemetry about ContainerPilot internals.

The fields for a sensor are as follows:

- namespace, subsystem, and name are what the Prometheus client library will use to construct the name for the telemetry. These three names are concatenated with underscores _ to become the final name that is recorded by Prometheus. In the above example, the metric recorded would be named my_namespace_my_subsystem_my_event_count. You can leave off the namespace and subsystem values and put everything into the name field if desired; The option to provide these other fields is simply for convenience of those who might be generating ContainerPilot configurations programmatically. Please, read the Prometheus documents on the best practices to name your telemetry.

- help is the help text that will be associated with the metric recorded by Prometheus. This is useful in debugging since it gives a more verbose description.

- type is the type of collector that Prometheus will use (one of counter, gauge, histogram or summary). See below for details.

- poll is the time in seconds between running the check.

- check is the executable (and its arguments) that is called when it is time to perform a telemetry collection.

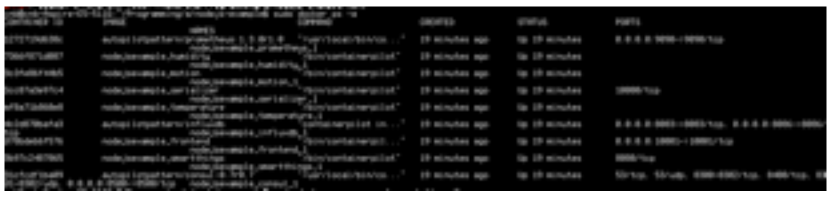
Using a sensor in telemetry is handy as you just create shell scripts that expose metrics from your service. In this case, this sensor will expose the free memory available in that container.

```
#!/bin/ash
# check free memory
echo "checked free memory sensor" 1>&2
free | awk -F' +' '/Mem/{print $3}'
```

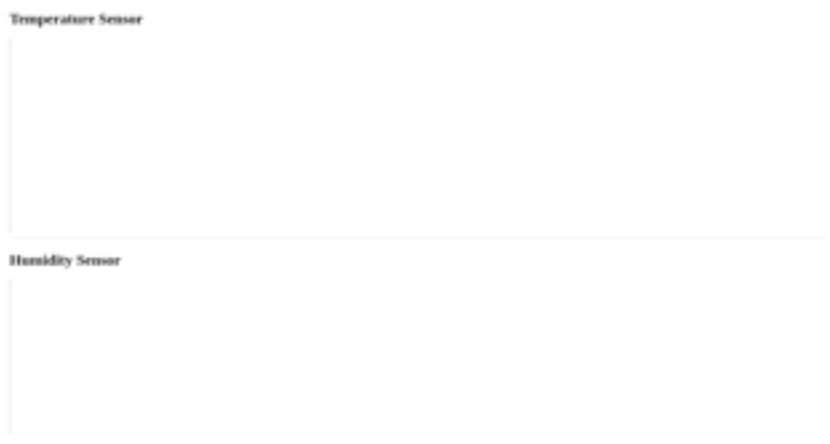
So, let’s check the actual project and start spinning containers. To get the project running, you need to follow these steps:

```
$ git clone
https://github.com/cneira/nodejs-example.g
it

$ sudo docker-compose -f local-compose.yml
up -d
```

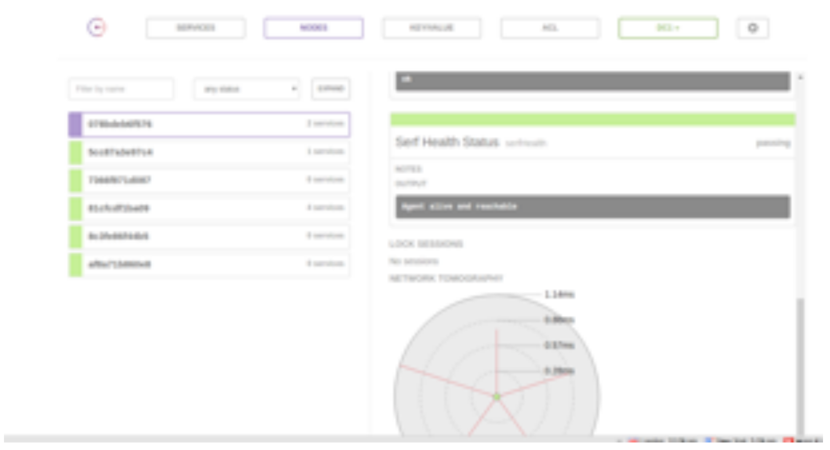
Containers images will start building. After they are all up, you will get access to the following:



http://localhost:10001 Here, you will see charts for the sensors, this is the actual application.



http://localhost:8500 Here, you will have access to consul and you’ll be able to check the health and which services have registered themselves in the catalog.



http://localhost:9090 Here, you will see Prometheus graph all the metrics exposed by your services.

Want to scale?
https://docs.docker.com/compose/reference/scale/

```
$ sudo docker-compose  scale serializer1=3
```

This will spin up to three instances of the serializer1 container. All of which will be managed by containerautopilot and will register themselves in the consul's catalog.

Conclusions

ContainerPilot saves you a great deal of effort when dealing with containers. It gives you out of the box telemetry, service discovery, etc. so that you don't need to code these services yourself. In this example, Node.js is used, but ContainerPilot could be used with any service coded in any language. Imagine leveraging ContainerPilot to add observability, telemetry, and auto-restart features to some legacy apps; that is a big win. This is just an introduction to the ContainerPilot. In another article, I'll create an application in C++ that leverages the ContainerPilot goodies and deploys them using Triton.

References:

https://www.docker.com/what-docker

https://wiki.freebsd.org/Docker

https://apidocs.joyent.com/docker/divergence

https://github.com/autopilotpattern/nodejs-example

https://www.joyent.com/blog/triton-docker-and-the-best-of-all-worlds

https://www.joyent.com/containerpilot

http://autopilotpattern.io/

https://www.youtube.com/watch?v=IwnUUJJw7UU

https://docs.docker.com/compose/gettingstarted/#step-3-define-services-in-a-compose-file

https://www.joyent.com/blog/running-node-js-in-containers-with-containerpilot

https://www.joyent.com/triton/compute

https://docs.joyent.com/private-cloud

About the Author

Carlos Antonio Neira Bustos has worked several years as a C/C++ developer and kernel porting and debugging enterprise legacy applications. He is currently employed as a C developer under Z/OS, debugging and troubleshooting legacy applications for a global financial company. Also he is engaged in independent research on affective computing . In his free time he contributes to the PC-BSD project and enjoys metal detecting.

Build FreeBSD 12 For RaspberryPi 3 With Crochet

Why FreeBSD ?

FreeBSD is one of the most stable OS of all time. Moreover, it has many bug fixes and new features for ARM SOC platform. The following are some of these features:

- CPU frequency and voltage control
- NAND device support
- SMP support
- Stable SD cards detection
- ARM AArch64 architecture support
- Initial ACPI support
- 1-Wire devices support
- GPIO support
- and many more.

Also, you can find the latest FreeBSD build for RaspberryPi 3 and you have to do that single-handedly.

What Is RaspberryPi 3 and Why RaspberryPi 3?

The Raspberry Pi 3 is the third generation of Raspberry Pi **SOC** after having replaced the Raspberry Pi 2 Model B in February 2016.

A **single-board computer** (**SBC**) is a complete computer built on a single circuit board, with microprocessor(s), memory, input/output (I/O) and other features required of a functional computer. Single-board computers were made for demonstration or development systems, for educational systems, or for use as embedded computer controllers. Many types of home computers or portable computers integrate all their functions onto a single printed circuit board.

Compared to the Raspberry Pi 2, it has:

- A 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)

Like the Pi 2, it also has:

- 1GB RAM
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot (now push-pull rather than push-push)
- VideoCore IV 3D graphics core

We choose RaspberryPi 3 because of a better CPU clock and Wi-Fi support.

What Is Crochet?

Crochet is a tool for building bootable FreeBSD images. This tool was formerly known as "freebsd-beaglebone" or "beaglebsd" since the original work was done for

BeagleBone. But it now supports more boards and should easily extend to support much more.

How To Build FreeBSD 12 For RaspberryPi 3?

You need to get FreeBSD Source-Tree. Thereafter, compile the source and write it on the SD card. So, let us do it step by step:

- You need to get the latest Source-Tree with subversion:

```
#pkg install subversion
#svn co
https://svn0.us-west.freebsd.org/base/head
/usr/src
```

Subversion is a version control system which allows you to keep old versions of files and directories (usually source code), keep a log of who, when, and why changes occurred, etc., like CVS, RCS or SCCS. Subversion keeps a single copy of the master sources. This copy is called the source **repository**, it contains all the information to permit extracting previous versions of those files at any time.

If something goes wrong and you must do it again, first, issue this command:

```
#svn cleanup /usr/src/
```

- Now, it's time to get **Crochet** to build our image:

```
#pkg install git-2.13.0

#git clone
https://github.com/freebsd/crochet.git
```

- Select which board you want to build for:

```
#cd crocket
#cp config.sh.sample config.sh.rpi3
#ee config.sh.rpi3
```

- uncomment this line (38) and exit:

```
board_setup RaspberryPi3
```

- Build the image:

```
#pkg install u-boot-rpi3-2017.01
```

- Cross-build U-Boot loader for RPi3

Das U-Boot (subtitled "the Universal Boot Loader" and often shortened to **U-Boot**) is an open-source, primary boot loader used in embedded devices to package the instructions needed to boot the device's operating system kernel. It is available for a number of computer architectures, including 68k, ARM, AVR32, Blackfin, MicroBlaze, MIPS, Nios, SuperH, PPC, RISC-V and x86.

U-boot-rpi3 is a Cross-build U-Boot loader for Rpi3.

```
#./crochet.sh -c config.sh.rpi3
```

Build process depends on your CPU speed but in general, it demands a lot of time.

How To Install FreeBSD 12 On RaspberryPi 3?

All you need is a 2GB MicroSD and a MicroSD reader. Let's suppose that after you connect your MicroSD to your PC, FreeBSD named it **da0** then:

```
#cd crochet/work/

#dd
if=FreeBSD-aarch64-12.0-GENERIC-319760.img
of=/dev/da0 bs=1m conv=sync
```

The name of your ".img" build revision can be different.

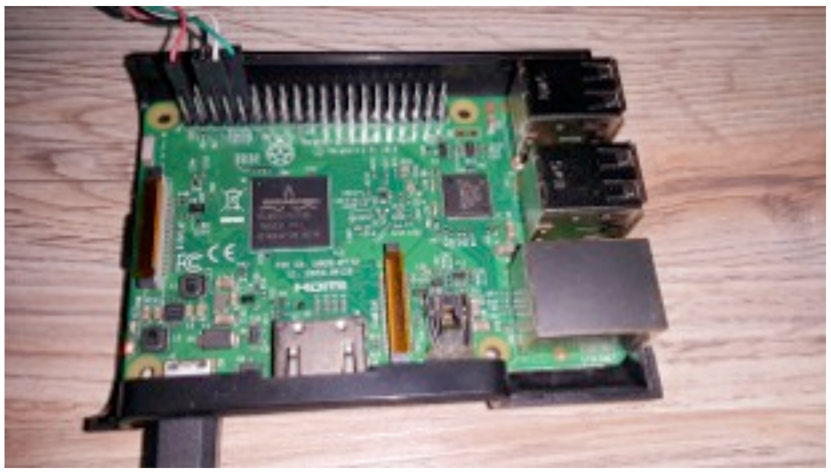
How To Access To RaspberryPi Console?

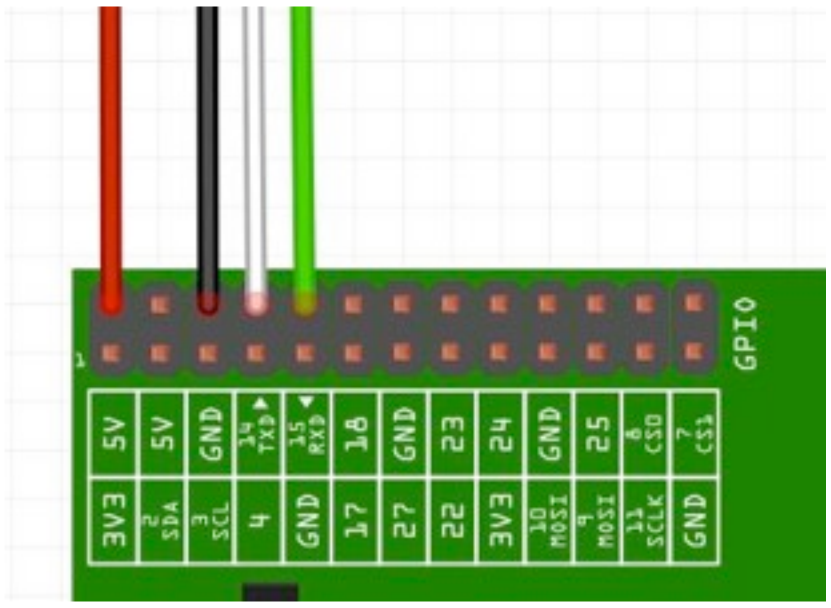
There are three ways to do that:

- HDMI cable and Keyboard
- SSH
- Console Cable

Access to RaspberryPi with Console Cable is more technical and Geek-Style. Thus, I chose to cover it. These days, most microcontrollers have a built in UARTs (universally asynchronous receiver/transmitter) that can be used to receive and transmit data serially. UARTs transmit one bit at a time at a specified data rate (i.e. 9600bps, 115200bps, etc.). This method of serial communication is sometimes referred to as **TTL serial** (transistor-transistor logic). Serial communication at a TTL level will always remain between the limits of **0V and Vcc**, which is often 5V or 3.3V. A logic high ('1') is represented by Vcc while a logic low ('0') is 0V.

Connect RaspberryPi 3 power supply and USB cable. Plug TTL Serial cable like below pictures:





Access to RaspberryPi with CU command.

The CU utility establishes a full-duplex connection to another machine, giving the appearance of being logged in directly on the remote CPU. It goes without saying that you must have a login on the machine (or equivalent) to which you wish to connect.

```
#cu -l /dev/ttyU0 -s 115200

-l Specifies the line to use

-s Sets the speed of the connection.  The
default is 9600.
```

Hit enter and you will be required to login.

user: root

Password is not required.

As you can see, command prompt shows us “rpi3”. You can issue **“uname -a”** to see more details about arch and FreeBSD version.

Conclusions

Although the build process is time consuming, FreeBSD 12 is fully supported on RaspberryPi 3.

Useful Links

http://meetbsd.ir

http://in4bsd.com

About the Author

Abdorrahman Homaei has been working as a software developer since 2000. He has used FreeBSD for more than ten years. He became involved with the meetBSD dot ir and performed serious training on FreeBSD. He is starting his own company in Feb 2017.

You can visit his website to view his CV: **http://in4bsd.com**

BORN TO DISRUPT

MODERN. UNIFIED. ENTERPRISE-READY.

INTRODUCING THE TRUENAS® X10, THE MOST COST-EFFECTIVE ENTERPRISE STORAGE ARRAY ON THE MARKET.

Perfectly suited for core-edge configurations and enterprise workloads such as backups, replication, and file sharing.

- ★ **Modern:** Not based on 5-10 year old technology (yes that means you legacy storage vendors)
- ★ **Unified:** Simultaneous SAN/NAS protocols that support multiple block and file workloads
- ★ **Dense:** Up to 120 TB in 2U and 360 TB in 6U
- ★ **Safe:** High Availability option ensures business continuity and avoids downtime
- ★ **Reliable:** Uses OpenZFS to keep data safe
- ★ **Trusted:** Based on FreeNAS, the world's #1 Open Source SDS
- ★ **Enterprise:** 20TB of enterprise-class storage including unlimited instant snapshots and advanced storage optimization for under $10,000

The new TrueNAS X10 marks the birth of a new entry class of enterprise storage. Get the full details at iXsystems.com/TrueNAS.

iX systems™

Implementing an Enigma Machine Simulator as a Character Device

This article presents a "curious" Enigma machine simulator implemented as a multi-platform character device. Until now, this device driver can be built on FreeBSD and also on Linux. The text will focus on the source code related to the FreeBSD. The aim of this article is to introduce the reader to the main aspects of device driver programming for FreeBSD using more than a simple Hello World sample. Therefore at the end of the article, the interested readers will be able to have a historical cryptographic device within their /dev. Tips on how to produce a multi-platform code base for device drivers will also be presented. For the sake of brevity, many details about Enigma machine were omitted. Due to the same reason, the text does not cover intermediate-level C programming.

What is the Enigma Machine?

The Enigma was an electro-mechanical cipher machine created by Arthur Scherbius and it was used by the German army during the World War II. The cipher implemented on this machine was first cracked by the Polish mathematician, Marian Rejewski. After some improvements, the produced cipher by the Enigma was strengthened, and this new Enigma version was cracked by the English mathematician, Alan Turing. He used the famous "Bombes" and his breakthrough ideas about Universal Machines. It is common to see the term "Turing Bombe", but the term "Bombe" was first used by Rejewski during his "hacks" against the first Enigma version.

Nowadays it is even being considered an outdated cipher. Although it is not used anymore for secret communications, the Enigma uses concepts widely applied on modern ciphers.

For the sake of brevity, deeper details about how Enigma encrypts data will be avoided. But in the next section, a quick description about the machine internals will be provided.

How does the Enigma machine work?

The Enigma is categorized as a rotor machine. In cryptography, rotor machines are devices composed of electrical and/or mechanical parts that encrypt an input. Commonly, these machines can produce different codes for the same input, producing what is called Polyalphabetic ciphers.

The Enigma has four main parts: "rotors", "rings", "plugboard" and "reflectors". Each of them responsible for increasing in some way the final security of the cipher. The machine looks like a typewriter, but instead of paper, there is a panel of light bulbs that depending on the input, it turns on a specific light bulb representing a letter. Once encrypted, all that the operator should do is to write down the output. During the WWII, the encrypted messages were sent using Morse codes.

Roughly, the motion of the rotors can be understood as an odometer. There are three rotors and they can be arranged in any position. They are chosen from a set of eight rotors. Each rotor can have its initial position configured from A to Z. Still, these rotors can have their internal wiring shifted. In this way, the outputs will be influenced. It is possible to shift each rotor to twenty-six different positions.

The plugboard swaps a letter pair. Ten letters can be swapped with the plugboard. When two letters are swapped, one is assumed as another (internally in the keyboard). As a result, the output also changes.

The reflectors are statical wirings and can be chosen from a set of two. They send back the electrical signal and can change the signal's path thus influencing the output. As you can see, the Enigma is a combination of smaller components that alone, cannot provide much security. However, when combined, they can make it less susceptible to a brute force attack. Remember that all of it was used in the past when there were no supercomputers. Truly, computers were built to crack this cipher and others, but not using brute force.

The idea of making it less prone to a brute force attack is still applied today. This is the security goal on the modern ciphers. Like Enigma, these modern ciphers are composites of small transformations that alone, cannot provide much security. Unlike Enigma, the modern ciphers operate in bit-level instead of letters. The Enigma is considered a reciprocal cipher. Reciprocal ciphers do not have a separated inverse function to decrypt the data. The function applied on encryption is the same applied on decryption. Once the machine is configured, if the encrypted message is typed, it will reveal the original message also known as plaintext. It can be understood as an ancient additive stream cipher. Modern stream ciphers like *RC4* and *SEAL,* for example, use a single *XOR* operation when encrypting a byte. Furthermore, the *XOR* is considered a reciprocal operation. Also, modern block ciphers in *OFB* operation mode are reciprocal. However, there are more ancient ciphers that implement this reciprocal property. The *Albam*, *Atbah* and *Atbash* present in *Torah* are reciprocal. Also, a cipher presented in *Kama Sutra*, called *Mlecchita vikalpa* and so on.

If you want more details about the Enigma, take a look at http://www.codesandciphers.org.uk/enigma/ among other sites, the internet is full of information about it. If you want to know the Math related to the brute force attack against Enigma, you can follow it at https://github.com/rafael-santiago/dev-enigma/blob/master/etc/brute-force.md. A bunch of movies about this machine was produced; the last one was "*The Imitation Game*" (2014). Maybe nowadays the Enigma could be considered a "popular" machine, contrary to the early days when it was classified. Now is time to talk about Programming!

From an electro-mechanical device to a logical device

With the reciprocal behavior, the Enigma cipher makes it easy to read and write operations if you start thinking of it as a device driver.

In general, a device driver can be understood as the logical part of a real world device, also known as *hardware*. By the way, logical is also known as *software*. If this special Software has its hardware implementation, you can think of it as an interpreter between the hardware and the Operating System (OS).

Also, it is possible to find device drivers purely implemented as software. In this case, sometimes it can act (emulate) like a physical device.

A device driver can 'talk' directly with the Operating System. Therefore, it acts as a bridge between the User and Kernel Space.

Many device drivers are deployed "from factory" into the kernel. I meant statically compiled. Opposing the "static" paradigm, many modern Operating Systems offer a way to integrate, on-the-fly, new kernel features with the *Loadable Kernel Modules* technique (LKMs). FreeBSD, Linux, Windows, OSX are some examples of Operating Systems with some LKM support. OpenBSD is an example of OS that does not provide a LKM implementation. In FreeBSD, the LKM is called *KLD*.

The nice part about LKMs is that you avoid a kernel recompilation when a new feature needs to be added to the OS. The result is a more modular kernel implementation. The nasty part is to add new random bugs (sometimes not tracked by the official Kernel Team Development) instead of new features. If you are writing some kernel code which you deem important, be sure about the correctness of your code since in this kind of environment, a segmentation fault leads to a kernel panic, in most cases.

Moreover, you can roughly assume the LKM as a dynamic library loaded by the kernel's "ld".

In UNIX, there are two main types of devices: *character devices* and *block devices*. Character devices are stream based and provide direct access to the device. On the other hand, Block devices provides buffered access to the hardware. On a character device, you can either read or write byte-by-byte whereas with block devices, you can read or write block-by-block. Also, there are *pseudo-devices*. Pseudo-devices can mimic some physical device and hence provide some access of a kernel function, etc. Usually, pseudo-devices are implemented as character devices. FreeBSD does not support block devices anymore.

In the next sections, we will discuss about pseudo and character devices. Details of other types of devices will not be provided or discussed.

The main functions implemented by a character device

A character device creates a “virtual” file within /dev directory. Through this file, it is possible from the user space to use the kernel functions implemented by this device by simply handling the “virtual” file.

Due to file handling, the necessities when you implement a character device are important to support the four main file functions: open, read, write and close.

In general, the open function should return a valid file descriptor. The read function should return some read bytes and the amount of read bytes. The write function should write the passed bytes and return the amount of written bytes. And finally, the close function should clean up the internal state associated with the file descriptor previously returned by the open function.

The programmatic way of accessing these device driver functions from the user space is by using the libc functions: open, read, write and close. Therefore, get a file descriptor with the open function and handle this descriptor with read, write and close functions.

Sometimes it is also necessary to setup or configure the device driver in some way. If the device driver needs this, it is necessary to implement the ioctl function. From libc, call the ioctl function passing the related file descriptor to control the device from the user space.

Now, let’s go back to the Enigma machine. Since Enigma machine acts like a cryptographic typewriter, it is possible to implement it as a character device. The reciprocal behavior is nice too because all that is necessary to do with the device is to configure the initial state with an ioctl call, feed the device with some data through a write call, and get the result back with a read call. The encryption or decryption depends on what was written because again, these two operations in the Enigma are the same. Now is time to dive into the /dev/enigma code.

The /dev/enigma code base layout

The /dev/enigma repository is organized in two main sub-directories: etc and src. The etc directory contains some additional support files and the src, of course, contains the source code. Within the src subdirectory, the device driver is divided into modules. Each module is a distinct sub-directory. These modules are: *dev_ctx*, *ebuf*, *eel*, *enigmactl*, *fops_impl* and *mod_traps*.

The dev_ctx module

This module creates virtual sessions of the /dev/enigma. This part of the device driver allows different users and data on the same device. The way of making it work is to representing each session as a structure, and also by using mutexes with their synchronization primitives.

When a user opens the device, a “usage line” is created. This usage is expressed in the following C structure:

```
struct dev_enigma_usage_line_ctx {
    libeel_enigma_ctx *enigma;
    ebuf_ctx *ebuf_head, *ebuf_tail;
#if defined(__linux__)
    struct mutex lock;
#elif defined(__FreeBSD__)
    struct mtx lock;
#endif
    int has_init;
};
```

Listing 1: The dev_enigma_usage_line_ctx struct

Each usage line has its mutex (the field named as "lock"). The libeel_enigma_ctx pointer is related with the Enigma machine emulator that belongs to the user’s session. The ebuf_ctx pointers are related with a linked list that stores the input andoutput data.

However, the device driver as a whole is represented by another structure called dev_enigma_ctx, detailed in Code Listing 2.

```
struct dev_enigma_ctx {
    struct dev_enigma_usage_line_ctx
ulines[DEV_USAGE_LINES_NR];
#if defined(__linux__)
    int major_nr;
    struct class *device_class;
    struct device *device;
#elif defined(__FreeBSD__)
    struct cdev *device;
#endif
    libeel_enigma_ctx *default_setting;
#if defined(__linux__)
    struct mutex lock;
#elif defined(__FreeBSD__)
    struct mtx lock;
#endif
};
```

Listing 2: The dev_enigma_ctx struct

As shown in Code Listing 2, the dev_enigma_ctx struct allows n usage lines, where n is DEV_USAGE_LINES_NR. If there is need to allow more users simultaneously handling the device, the constant DEV_USAGE_LINES_NR should be increased.

With other codes described, we will return to the dev_ctx module to understand some additional details.

The ebuf module

The ebuf is a tiny code module that implements linked list conveniences. However, when you need some convenience related with some data structure, you should try to use the data structures provided by its Kernel API. This reuse is considered a good practice. Both FreeBSD and Linux have their implementation of these well-known data structures.. But in the case of /dev/enigma, the data structure was so minimal that it was written from scratch. Code Listing 3 shows the entire ebuf header file.

```
#ifndef DEV_ENIGMA_EBUF_H
#define DEV_ENIGMA_EBUF_H 1


typedef struct _ebuf_ctx {
 char c;
 struct _ebuf_ctx *next;
}ebuf_ctx;


int add_char_to_ebuf_ctx(ebuf_ctx **ebuf,

                         const char c, ebuf_ctx
*tail);

char get_char_from_ebuf_ctx(ebuf_ctx **ebuf);

void del_ebuf_ctx(ebuf_ctx *ebuf);

#endif
```

`Listing 3: The ebuf.h header file`

The declared ebuf_ctx struct in th Code Listing 3 stores the user inputs. This storage is done by the add_char_to_ebuf_ctx(). Contrary to that, the get_char_from_ebuf_ctx() removes data from ebuf_ctx and also returns the removed data.

Remember that data here is merely a byte written or read by the user, and it should be encrypted or decrypted using the associated Enigma Machine's setting. The del_ebuf_ctx() function is a cleanup function used to free the previously allocated ebuf_ctx resources.

The eel module

The eel module is related to the Enigma Machine simulator. This was implemented according to the specifications provided by Tony Sale's website at http://www.codesandciphers.org.uk/enigma/. For brevity, it will not be discussed here.

The enigmactl module

The enigmactl module defines some C macros related to the I/O control of the device driver. Also, inside this directory is a source code of a user mode application. The user mode application is a way of controlling the device driver from the user space. The application works by reading some command line options and emitting some ioctl calls based on those.

To implement ioctl support for a device driver, it is necessary to create the commands that the user applications will pass as arguments to an ioctl call. Creating a command involves defining constants using special C macros provided by the Kernel API. These definition macros can be _IO, _IOR, _IOW or _IOWR.

To decide what to use depends on the effect intended with the ioctl call. Sometimes it is necessary to only set up the device driver with some user data. In this case, the command for doing it should be defined with the macro _IOW (W means to write). When there is need to only read some data and return it to the user, the ioctl command should be defined with _IOR (R means to read). When it is necessary to write from a user space and return data to the user space, the macro _IOWR (WR, guess what?) is the choice. Code Listing 4 follows the ioctl commands of the /dev/enigma.

```
#define ENIGMA_IOC_MAGIC 'E'


#define ENIGMA_RESET _IO(ENIGMA_IOC_MAGIC, 0)
#define ENIGMA_SET _IOW(ENIGMA_IOC_MAGIC, 1,
libeel_enigma_ctx)
#define ENIGMA_SET_DEFAULT_SETTING
_IOW(ENIGMA_IOC_MAGIC, 2, libeel_enigma_ctx)
#define ENIGMA_UNSET_DEFAULT_SETTING
_IO(ENIGMA_IOC_MAGIC, 3)
```

`Listing 4: The ioctl commands supported by /dev/enigma`

The first argument passed with the macros _IO, _IOR, _IOW and _IOWR is related to the group, and it must be a byte constant. In FreeBSD, it seems okay to pick any, contrary to Linux where you need to pick an unused one so as to follow the convention adopted (https://github.com/torvalds/linux/blob/master/Documentation/ioctl/ioctl-number.txt).

The second argument passed with the _IO* macros is the internal number. This will represent the related command.

The macros, _IOR and _IOW, also receive a third argument related to the data type that should be read or written. When it is necessary to pass more than one argument, the convention is to pass a struct which groups those multiple arguments. This is the case with the /dev/enigma. The command ENIGMA_SET is emitted for setting up the Enigma Machine simulator, and it needs to inform the chosen rotors, their initial position, their rings, the plugboard, the reflectors and so on. All machine configurations go in the libeel_enigma_ctx struct.

The ENIGMA_RESET command is used when there is a necessity to reset the machine to its initial configuration state. For decrypting a message, the machine needs to be in the same initial configuration used during the encryption process.

The ENIGMA_SET_DEFAULT_SETTING is an ioctl command for defining a default machine setting to any file descriptor acquired using the libc open() call. Thus, once the device is opened it can start using it without configuring the machine.

The ENIGMA_UNSET_DEFAULT_SETTING is used when the default setting is not required anymore. It will clear the default setting.

The fops_impl module

This module is where the functions of the device driver (the file operation functions) are implemented. Let's start with the open function. The code of this function follows the details in Code Listing 5.

```
MALLOC_DEFINE(M_DEV_OPEN,

              "DEV_ENIGMA_dev_open", "Allocations
related with dev_open");

int dev_open(struct cdev *dev,

             int flags __unused,

             int devtype __unused, struct thread
*td __unused) {
    int uline = new_uline();

    if (uline == -1) {
        return -EBUSY;
    }

    if (!lock_uline(uline)) {
        return -EBUSY;
    }

    dev->si_drv1 = malloc(sizeof(int), M_DEV_OPEN,
M_NOWAIT);

    if (dev->si_drv1 != NULL) {
        *((int *)dev->si_drv1) = uline;
        devfs_set_cdevpriv(dev->si_drv1,
dev_close_dtor);
    }

    unlock_uline(uline);

    return (dev->si_drv1 != NULL) ? 0 : -ENOMEM;
}
```

Listing 5: The /dev/enigma open function

As you can see in the prototype of dev_open(), the functions used are some internal Kernel data structures besides being flagged. The struct cdev is an important structure that represents the character device. The "__unused" keyword is a way of saying to the compiler that even when not using the related variable, that variable should be there. Otherwise, the compiler will emit warnings about the lack of usage of them. Note that it is important to follow the prototype since these file operation functions will be referenced later by some internal function pointers.

The first thing done in the dev_open() function is attempt to acquire a valid usage line (a user session with a non-busy Enigma Machine). If the new_uline() function returns -1, it means that there are no free Enigma Machines at that moment Due to this, the dev_open() warns that the device is currently busy. As a result, the user application will get this error.

Otherwise, with a valid usage line, it is necessary set up some information about a successful open operation. The information will be used later for cleaning up the usage line.

It is important to do all this work in an "atomic" way. Due to this, first you need to call the lock_uline() function. This function will try to lock the mutex associated with the current usage line. When failing, an error is returned indicating that the device is busy. If it succeeds, it indicates that any other lock attempt will fail. This guarantees that nothing besides the current instance will access the following code section or any other code section that depends on locking the related usage line mutex.

After the lock is acquired, some memory bytes are allocated that will point to the current usage line value returned by the new_uline() function. Note that in the Kernel API, the malloc call changes a little when compared with the classical libc malloc.

The FreeBSD's malloc from the Kernel API receives the amount of memory to be allocated, an identifier for this allocation call and a mode of operation. This operation flag changes the malloc behavior depending on what you pass. In the Code Listing 5, N_NOWAIT was used which simply says not to wait if there is no memory. The reason we use this flag is that it is considered a bad idea to use a code that could block or hang the Kernel mode (thus hanging the Kernel as a whole). When passed, N_NOWAIT malloc could return NULL. Therefore, it is paramount to check the nullity of the return otherwise, we can get a kernel panic by accessing a null pointer.

The way of defining these parameters requested by the malloc function is easy. You should use the macro MALLOC_DECLARE, where you simply create an identifier, for example, MALLOC_DECLARE(M_DEV_OPEN). Thereafter, you need to define a message related to this identifier by using the macro MALLOC_DEFINE, for example:

```
MALLOC_DEFINE(M_DEV_OPEN,

              "DEV_ENIGMA_dev_open", "Allocations
related with dev_open")
```

This can help to find bugs related with pointers allocated using the defined identifier because the Kernel will log the allocations using the passed identifier in the malloc call. I find it pretty handy and cool.

The usage of devfs_set_cdevpriv() in the dev_open() function (Code Listing 5) is a "trick" to clean up the usage line when closing the file descriptor. If not done, the device driver will start returning EBUSY errors even when no one is using it.

Roughly speaking about the devfs_set_cdevpriv(), we are saying, "well, when closing the file descriptor associated with this open operation, call the function dev_close_dtor thus passing the pointer si_drv1". In this case, si_drv1 points to the usage line index. Having this index, we can easily clean up the usage line in dev_close_dtor function, making it ready to handle another future request. By the way, the si_drv1 and si_drv2 are work void pointers left in the cdev structure for a general use by developers.

The last thing to be done before returning is unlocking the mutex so that other operations that depend on this mutex can go ahead.

An important remark: The dev_open function does not return the file descriptor that the user open call receives, this is handled by the Kernel. The dev_open() function should only tell the Kernel if an error occurred during its execution. If everything is fine, the Kernel will return a valid file descriptor to the user.

We have covered all the dev_open function code. Now, Code Listing 6 shows the code of dev_close_dtor(). The function code is straightforward, given a usage line number the dev_close_dtor function calls the release_uline() function passing this number. Afterwards, the previous allocated memory by dev_open() is freed.

```
void dev_close_dtor(void *data) {
    if (data == NULL) {
        return;
    }

    release_uline(*(int *)data);

    free(data, M_DEV_OPEN);

    data = NULL;
}
```

Listing 6: The dev_close_dtor function

During the dev_open() code discussion, some synchronization using mutexes was necessary but the code for this synchronization was not detailed. Code Listing 7 shows how the mutex of each usage line is handled.

```
int lock_uline(const int uline) {
    if (!(uline >= 0 && uline <=
DEV_USAGE_LINES_NR)) {
        return 0;
    }

#if defined(__linux__)
    if
(!mutex_trylock(&g_dev_ctx.ulines[uline].lock)) {
        return 0;
    }
#elif defined(__FreeBSD__)
    if
(!mtx_trylock(&g_dev_ctx.ulines[uline].lock)) {
        return 0;
    }
#endif
```

```
    return 1;
}
```

Listing 7: The lock_uline function

There are a bunch of mutex functions, and you should know all of them. The best resource for this is the man pages. In Code Listing 7, for the FreeBSD, I have used the function mtx_trylock. The nice thing about this function is that it will not hang if the mutex is already locked, instead it will simply fail. Remember: In Kernel mode code, busy waiting tends to be pretty bad. If you call lock_uline() and get a return value of 1, you can proceed because no one else holds the lock.

Synchronization and mutexes are extensive subjects. Therefore, if you want to dive into Kernel programming, you should master everything you could about it. There are other types of mutexes, for example, shared locks. This type of lock can be acquired by more than one instance depending on some rules that your code logic implements. The best practice is to find the type of lock that best fits your concurrency requirements.

The next relevant device driver function is the dev_write(). This function will be indirectly accessed when calling the libc write() function from the user space over the file descriptor previously returned by the libc open() function. In the /dev/enigma context, a write() function means to type something on the machine's "keyboard".

The dev_write() function follows the details in Code Listing 8. The first thing done by this function is to read the stored usage line reference from the field si_drv1 in the cdev struct. The dev_write() function also uses another structure called uio. The uio structure gathers data that is flowing to and from Kernel space. In the case of dev_write(), the data is flowing to the Kernel from the user space. The ioflags parameter is a set of flags related to the write operation and not relevant to the /dev/enigma.

After making sure that the usage line is initialized (Is the Enigma machine well-configured?), a buffer with "iovlen" bytes is allocated. If this allocation succeeds, the usage line is locked for exclusive use and the data from the user space is transferred to the Kernel space by calling the uiomove() function.

Personally, I find the uiomove() function awesome. Because, when it is combined with the uio struct, the function is capable of knowing our intentions about the data transferring operation. So, you do not need to explicitly use separated functions to copy data from the Kernel to user spaces and *vice versa*. Basically, that is what you do in other operating systems. In FreeBSD, when it is necessary to transfer some data between the Kernel space and user space, the uiomove() is considered the best way to achieve that.

Once the data is transferred, all that should be done is to parse the read buffer, adding each byte to the internal linked list ebuf. However, the data can be encrypted or decrypted with the Enigma by simply calling "libeel_type(read_byte)".

Finally, the usage line is unlocked and if all is okay, 0 is returned. Otherwise, a generic fail is signaled. Note that unlike other operating systems, in FreeBSD, the dev_write() function should not return the amount of written data.

The dev_read() function detailed in Code Listing 9 is rather similar to the dev_write(). The difference is that the data stored in the ebuf linked list is not post-processed since it was already pre-processed during the dev_write() execution. Thus, the stored data is just removed from the linked list and returned to the user space.

Note that even with a different context, the dev_read() also uses uiomove() to transfer data from the Kernel to user space. But in this case, it is not necessary to allocate a buffer because it is transferred to the passed user buffer. All that should be done is to avoid going beyond the buffer limit. The condition, "read_bytes != user_len", prevents exceeding the buffer limit.

Like dev_write(), the dev_read() function only tells the Kernel if the operation was successful or not.

The dev_ioctl() function is responsible for implementing the device driver control routines. Besides a struct cdev, this function receives the command index, the data passed by the user (if the ioctl command is a "write type" command) and a thread struct. The command index is related to the previously defined indexes in "enigmactl.h".

If the cmd is equaled to ENIGMA_RESET, a lock attempt over the usage line mutex is done, and once locked, the function libeel_init_machine() is called to put the Enigma simulator in its initial state.

If the cmd is equaled to ENIGMA_SET, the data passed by the user is copied from the Enigma struct to the usage line context (it represents an Enigma setting). This also calls libeel_init_machine() passing the new read settings from the data pointer.

The commands, ENIGMA_SET_DEFAULT_SETTING and ENIGMA_UNSET_DEFAULT_SETTING do not depend on locking the usage line because they only handle a default setting copy used by all acquired /dev/enigma instances. When a default is set it is loaded from the data pointer passed by the user. When the default is cleared, the function unset_default_enigma_setting() is called and the previous default setting is discarded.

The dev_ioctl() when there is no error, should return 0. The dev_ioctl() function is detailed in Code Listing 10.

```
int dev_write(struct cdev *dev, struct uio *uio,
int ioflags) {
    struct dev_enigma_usage_line_ctx *ulp;
    int uline;
    char *temp_buf = NULL;
    char *bp, *bp_end;
    ssize_t written_bytes = 0;
    size_t temp_buf_size = 0;

    uline = *(int *)dev->si_drv1;

    ulp = dev_uline_ctx(uline);

    if (ulp == NULL) {
        return -EBADF;
    }

    if (!ulp->has_init) {
        return -EINVAL;
    }

    temp_buf_size = uio->uio_iov->iov_len;
    temp_buf = (char *) malloc(temp_buf_size,
M_DEV_WRITE, M_NOWAIT);

    if (temp_buf == NULL) {
        return -ENOMEM;
    }

    if (!lock_uline(uline)) {
        return -EBUSY;
    }

    if (uiomove(temp_buf, temp_buf_size, uio) != 0)
{
        free(temp_buf, M_DEV_WRITE);
        unlock_uline(uline);
        return -EFAULT;
    }

    bp = temp_buf;
    bp_end = bp + temp_buf_size;

    while (bp != bp_end) {
        libeel_enigma_input(ulp->enigma) = *bp;
        written_bytes +=
add_char_to_ebuf_ctx(&ulp->ebuf_head,

libeel_type(ulp->enigma),

ulp->ebuf_tail);
        bp++;
    }

    free(temp_buf, M_DEV_WRITE);

    unlock_uline(uline);

    return (written_bytes == temp_buf_size) ? 0 :
EFAULT;
}
```

Listing 8: The dev_write function

```
int dev_read(struct cdev *dev, struct uio *uio, int
ioflags) {
    int uline;
    struct dev_enigma_usage_line_ctx *ulp;
    char byte;
    size_t read_bytes = 0;
    size_t user_len = 0;

    uline = *(int *)dev->si_drv1;

    ulp = dev_uline_ctx(uline);

    if (ulp == NULL) {
        return -EBADF;
    }

    if (!lock_uline(uline)) {
        return -EBUSY;
    }

    user_len = uio->uio_iov->iov_len;

    while (read_bytes != user_len && ulp->ebuf_head
!= NULL) {
        byte =
get_char_from_ebuf_ctx(&ulp->ebuf_head);
        if (uiomove(&byte, 1, uio) != 0) {
            read_bytes = 0;
            goto __dev_read_epilogue;
        }
        read_bytes++;
    }

__dev_read_epilogue:
    unlock_uline(uline);

    return (read_bytes == 0) ? EFAULT : 0;
}
```

Listing 9: The dev_read function

```
int dev_ioctl(struct cdev *dev, u_long cmd, caddr_t
data, int flag, struct thread *td) {
    long result = 0;
    libeel_enigma_ctx user_enigma;
    struct dev_enigma_usage_line_ctx *ulp;
    int uline;

    if (dev->si_drv1 == NULL) {
        return -EINVAL;
    }

    uline = *(int *)dev->si_drv1;

    ulp = dev_uline_ctx(uline);

    if (ulp == NULL) {
        return -EINVAL;
    }

    switch (cmd) {

        case ENIGMA_RESET:
            if (ulp->has_init) {

                if (!lock_uline(uline)) {
                    return -EBUSY;
                }

                ulp->has_init =
libeel_init_machine(ulp->enigma);

                if (!ulp->has_init) {
                    result = -EINVAL;
                } else if (ulp->ebuf_head != NULL)
{
                    del_ebuf_ctx(ulp->ebuf_head);
                    ulp->ebuf_head = NULL;
                }

                unlock_uline(uline);
            } else {
                result = -EINVAL;
            }
            break;

        case ENIGMA_SET:
            if (data == NULL) {
                return -EFAULT;
            }

            if (!lock_uline(uline)) {
                return -EBUSY;
            }

            memcpy(ulp->enigma, (libeel_enigma_ctx
*)data, sizeof(libeel_enigma_ctx));

            if (!(ulp->has_init =
libeel_init_machine(ulp->enigma))) {
                result = -EINVAL;
            } else if (ulp->ebuf_head != NULL) {
                del_ebuf_ctx(ulp->ebuf_head);
                ulp->ebuf_head = NULL;
            }

            unlock_uline(uline);
            break;

        case ENIGMA_SET_DEFAULT_SETTING:
            if (data == NULL) {
                return -EFAULT;
            }

            memcpy(&user_enigma, (libeel_enigma_ctx
*)data, sizeof(libeel_enigma_ctx));

            if
(!set_default_enigma_setting(&user_enigma)) {
                result = -EINVAL;
            }
            break;

        case ENIGMA_UNSET_DEFAULT_SETTING:
            if (!unset_default_enigma_setting()) {
                result = -EFAULT;
            }
            break;

        default:
            result = -ENOTTY;
            break;

    }
    return result;
}
```

Listing 10: The dev_ioctl function

The last file operation function that must be detailed is the dev_close() function. This function simply returns EBADF error if for some reason the pointer si_drv1 is null. This pointer is used to store the reference to the associated /dev/enigma usage line. If this is null, there is nothing to be done with the file descriptor because all session reference was lost. To avoid more serious errors, the current file operation is invalidated by returning EBADF. The dev_close() follows the details in Code Listing 11. It is important in normal conditions to return 0. Otherwise, the Kernel will return an error to the user.

```
int dev_close(struct cdev *dev,

                int flags __unused,

              int devtype __unused,

              struct thread *td __unused) {
    if (dev->si_drv1 == NULL) {
        return -EBADF;
    }

    return 0;
}
```

Listing 11: The dev_close() function

The file operation functions, dev_open(), dev_write(), dev_read(), dev_ioctl() and dev_close(), alone do not compose the device driver. They only specify how the device driver should act during the *open*, *write*, *read*, *ioctl* and *close* requests respectively.

Now, it is time to take a look at the *mod_traps* module.

The mod_traps module

In the mod_traps sub-directory, there are implementation files, “mod_init.c”, “mod_exit.c” and “mod_quiesce.c”. Each of these files implements an important function that will be called during the device driver's life cycle.

Let’s start with Code Listing 12 that shows the enigma_init() function.

The enigma_init() function shows us interesting things. The first is the way of initializing a mutex variable by calling mtx_init(). Note that the second parameter passed in mtx_init() could help you when debugging deadlocks, race conditions bugs, etc. This is because the parameter identifies the related mutex in the system logs.

The master mutex is directly initialized into the enigma_init(). However, each usage line has its mutex that is initialized in the init_ulines() function.

The function make_dev() registers and creates the device driver file descriptor within the /dev/ subdirectory. Thus, after calling make_dev(), if all succeeds, we will have some Enigma Machines ready in the /dev eventually. Nonetheless, the user will only see one Enigma there.

Another interesting and important thing present in Code Listing 12 is the declaration of the dev_enigma_cdevsw structure. This will register the file operation functions of the device thus creating a bridge between a user space call and the Kernel. Therefore, when some operation is done with the device driver, the Kernel will know what to call in order to handle the request. The struct initialization uses the *C99 convention*. The struct cdevsw has much more fields but they are not relevant for a character device. In this way, it is possible to initialize only what is deemed necessary.

Note that the registration of the file operation functions has occurred during the make_dev() call.

```
static struct cdevsw dev_enigma_cdevsw = {
    .d_version = D_VERSION,
    .d_open = dev_open,
    .d_close = dev_close,
    .d_read = dev_read,
    .d_write = dev_write,
    .d_ioctl = dev_ioctl,
    .d_name = DEVNAME
};


int enigma_init(void) {
    uprintf("dev/enigma: Initializing the
/dev/enigma...\n");

    dev_ctx()->default_setting = NULL;
    mtx_init(&dev_ctx()->lock,
             "DEV_ENIGMA_device_lock",
             NULL,
             MTX_DEF);

    init_ulines();

    dev_ctx()->device =
make_dev(&dev_enigma_cdevsw,
                                     0,
                                     UID_ROOT,
                                          GID_WHEEL,
                                          0666,
                                          DEVNAME);

    if (dev_ctx()->device == NULL) {
        uprintf("dev/enigma: device creation
fail.\n");
        return 1;
    }

    uprintf("dev/enigma: Done.\n");

    return 0;
}
```

Listing 12: The enigma_init function

The safe_for_unloading() function is called when the *KLD* is unloaded and it checks if the device driver is busy or not. The check logic of this function is not the best because it does not guarantee that a user just after unlocking the mutex will not lock the mutex again. However, it will work in most cases. Code Listing 13 follows the safe_for_unloading() function code. Code Listing 14 details the enigma_exit() function. This function releases the internal structures held by the device driver, and also releases the device driver. As a result, the /dev/enigma file will disappear since the *KLD* was unloaded.

```
int safe_for_unloading(void) {
    int safe = 1;
    int u;

    for (u = 0; u < DEV_USAGE_LINES_NR && safe;
u++) {
        if (!lock_uline(u)) {
            safe = 0;
        } else {
            safe = (dev_ctx()->ulines[u].enigma ==
NULL);
            unlock_uline(u);
        }
    }

    return safe;
}
```

Listing 13: The safe_for_unloading function

```
void enigma_exit(void) {
    uprintf("dev/enigma: The /dev/enigma "

            "is being unloaded...\n");

    deinit_ulines();

    unset_default_enigma_setting();

    mtx_destroy(&dev_ctx()->lock);

    destroy_dev(dev_ctx()->device);

    uprintf("dev/enigma: Done.\n");
}
```

Listing 14: The enigma_exit function

The functions enigma_init(), safe_for_unload() and enigma_exit() are triggered when the Kernel module that represents the device driver is loaded or unloaded. These functions also can be understood as the events that have occurred during the device driver life cycle.

In the src sub-directory, there is tan implementation file called "mod.c". This file has C code for FreeBSD and Linux, but Code Listing 15 shows only the FreeBSD part. According to Code Listing 15, a function with a specific prototype follows is implemented. This function has a suggestive name: enigma_modevent. Depending on the received event value, one of the three functions implemented in the mod_traps will be called. When the module is loaded, the event variable will be equaled to MOD_LOAD. When the module is being unloaded, without force, the "event" will be equaled to MOD_QUIESCE. Finally, when actually unloaded, the event variable will be equaled to MOD_UNLOAD.

The nice thing about the MOD_QUIESCE event handling is that if we return anything different from zero, the unloading will be canceled. This is because non-zero values mean an error case.

You can understand the enigma_modevent() as the "main function" of the driver. However, handling the events in the enigma_modevent() function does not have any effect when this function is isolated. It is important to inform the Kernel that the enigma_modevent() is the main entry point. In doing so, the Kernel API macro DEV_MODULE registers this function as the effective device driver event handler.

```
static int enigma_modevent(module_t mod __unused,
                           int event,
                           void *arg __unused) {
    int error = 0;

    switch (event) {
        case MOD_LOAD:
            error = enigma_init();
            break;
        case MOD_UNLOAD:
            enigma_exit();
            break;

        case MOD_QUIESCE:
            if (!safe_for_unloading()) {
                error = EBUSY;
            }
            break;

        default:
            error = EOPNOTSUPP;
            break;
    }

    return error;
}

DEV_MODULE(enigma, enigma_modevent, NULL);
```

Listing 15: The enigma_modevent function

Having covered all the device driver code, let's now get some relevant information on how to build the entire code.

Building the /dev/enigma

First, here is an important remark: to compile any device driver code discussed in this article, you must have the FreeBSD source code. Maybe you had already copied it when you installed your FreeBSD copy. Take a look in your /usr/src directory. If you do not have the source code, visit https://www.freebsd.org/developers/cvs.html for more information.

Bear in mind that only the device driver code related to FreeBSD was discussed above. The driver also compiles on Linux and in this case, of course, another Kernel API is used. The nice part about the /dev/enigma build is that the user does not need to edit anything to make the compilation possible in these two "Worlds". Actually, the user should invoke the same command in the two platforms.

I usually use my build system for my weekend projects. I call it *Hefesto* (https://github.com/rafael-santiago/hefesto). I will show you how to compile the /dev/enigma using Hefesto, but first, I will show you how to compile simpler drivers using the FreeBSD build framework, which is based on *Make*. Hence all you need to do is compose a *Makefile* and indicate your resources in it.

To compile a device driver in FreeBSD, use the framework implemented in bsd.kmod.mk. It can be done by including it into your Makefile. Also, it is important to indicate the output module name and the source files of this Kernel module. Code Listing 16 shows a Makefile sample.

```
KMOD= sample

SRC= device.c

include <bsd.kmod.mk>
```

Listing 16: A rather basic device driver Makefile sample

The problem with using a Makefile directly is the necessity of keeping track of two different Makefiles since the /dev/enigma should be compiled in FreeBSD and also Linux. Another problem is managing the Makefile. Personally, I find it boring and when the project is well-structured, divided into several sub-directories, the Makefile tends to become trickier and cryptic. As I said, in my weekend projects, I enjoy the freedom of using my stuff. Therefore, I decided to get rid of these Make complications by using one more layer of abstraction: my build system.

Using Hefesto, I just have to track one build script and it will make sense in any supported platform. Additionally, I can perform minor programmatic tasks such as installing and uninstalling the software, run tests and check if it is ok or not in a non-cryptic fashion. Hence, everything can be accomplished with only one tool, and as a result, only one programming language syntax for everything.

The way how I have been building the /dev/enigma

To build the discussed device driver using my build system, you need to clone three of my repositories using the following commands:

```
# git clone https://github.com/rafael-santiago/hefesto —recursive

# git clone https://github.com/rafael-santiago/helios —recursive

# git clone https://github.com/rafael-santiago/dev-enigma —
recursive
```

After doing this, you should move to the Hefesto's src sub-directory and call the bootstrap build (build.sh):

```
# cd hefesto/src

# /usr/local/bin/bash build.sh
```

After calling the bootstrap build, a prompt confirming where to install hefesto will be presented. All that is required of you is to confirm the location and Hefesto will be installed on your machine. However, you should do a new login in order to reload some environment variables (opening a new console window has the same effect if you are on X). The Hefesto's copy related to the downloaded repository can be removed.

At this instant, you should "teach" Hefesto how to build a FreeBSD device driver. This should be done in accordance with the Helios downloaded copy in the following way:

```
# cd helios

# hefesto --install=freebsd-module-toolset
```

The Helios can be understood as a "standard library" for Hefesto. Once the FreeBSD module toolset is installed, you can remove your Helios copy, and now Hefesto knows how to build a FreeBSD KLD.

To build the /dev/enigma , you should move to its src sub-directory and call Hefesto from there.

```
# cd dev-enigma/src

# hefesto
```

The device-driver and the support codes, such as enigmactl, will be compiled. The tests will run. Finally, the "enigma.ko" file will be created in the src sub-directory. The output should be something like:

```
(...)

*** All /dev/enigma was built.

(...)
```

If you want to do a quick loading test, use the "kldload ./enigma.ko" command. It is important to use "./" otherwise the kldload will not properly find the module. After executing the kldload command, a load message defined in the enigma_init() function, will be printed in the console as described below:

```
dev/enigma: Initializing the /dev/enigma...
dev/enigma: Done.
```

The kldstat command is a useful command that shows information about the loaded modules.

Its output shows "enigma.ko", indicating that it has been loaded. If a ls /dev is executed, it also shows the device driver's virtual file. In Figure 1, you can see *Beastie* pointing to it.

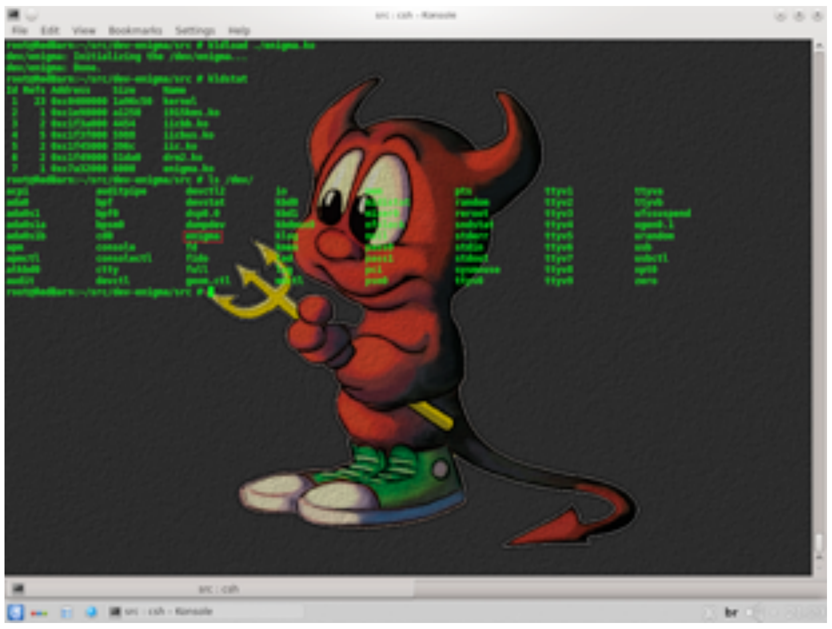


Figure 1: The ls /dev output after loading enigma.ko

To unload the enigma.ko you should use the command "kldunload enigma".

If you want to install the /dev/enigma, you should run the command "hefesto --install". To uninstall, use "hefesto --uninstall". Even when installed, you still need to load the module with kldload before using it.

Using the /dev/enigma

Due to the virtual sessions allowed by the device driver, opening the device and start writing or reading something is impossible. If you do it, you will get an error because the opened Enigma machine does not have a valid configuration yet. With the "enigmactl" application, it is possible to set a default configuration for opened sessions. This application comes in handy for a user who wants to use the device driver directly from a *shell script*.

The enigmactl expects several options related to the Enigma Machine components. If you run it without options, you can learn more about it. The enigmactl can be used through the   enigmactl.sh shell script. This shell script uses the dialog command to configure the Enigma machine in an easier and interactive way.

Also, after opening the device driver, it is essential to hold the file descriptor because it represents the acquired session. So, the "echo foobar > /dev/enigma" command does not work. Code Listing 17 shows how to handle file descriptors in a s*hell script*. It takes into consideration that /dev/enigma was previously installed by the "hefesto -–install"command.

The line “exec <> 3 /dev/enigma” present in Code Listing 17 will open the device driver file for reading and writing.

Before executing the *Shell script* in Code Listing 17, it is required to execute the command “kldload /usr/local/share/dev-enigma/mod/enigma.ko”. For an interested reader, the following is a short encrypted message:

“JTBXDUBAADYLVSWCVURPODZNVXPYSPWVSGNLV MPWURBJGGCYOCGOEUOCG”

The Enigma setting used was:

Rotors: VII, V and VIII (respectively)

Rotor position at: B, S, D (respectively)

Reflector: B

All rings at position “1”.

Plug-board: F/B,S/D (this is the input format expected by the “enigmactl.sh” script)

Would you be able to decrypt the message?

```
#!/usr/local/bin/bash

# File name: 'enigma.sh'

if [ -f /usr/local/bin/enigmactl.sh ] ; then
    dialog --yesno "Do you want setup /dev/enigma?"
0 0

    if [ $? -eq 0 ]; then
        /usr/local/bin/enigmactl.sh
        if [ $? -ne 0 ]; then
            exit $?
        fi
    fi
fi

exec 3<>/dev/enigma

if [ $? -ne 0 ]; then
    echo "error: when trying to open /dev/enigma."
    exit $0
fi

text=$1

if [ -z $text ]; then
    text=$(dialog --stdout --inputbox "Type the
text to be written to /dev/enigma" 0 0)
fi

if [ -z $text ]; then
    echo ""
    echo "INFO: aborted by the user."
    exit 1
fi

echo $text>&3

output=$(cat <&3)

dialog --title "/dev/enigma output" --infobox
$output 0 0

exec 3>&-
```

Listing 17: Using the /dev/enigma from a shell script

How to get a multi-platform code?

Having not discussed the source code part for Linux, this tiny device driver code shows how C language works pretty well on producing a multi-platform code. The C pre-processor can be a powerful tool in this case. Additionally, is it important to add more abstraction layers to your code. This explains why the code was not written as a huge single implementation file.

The layout of the repository can also contribute to the simplicity when producing the multi-platform code. For /dev/enigma, the *fops_impl* and *mod_traps* sub-directories concentrate all platform dependent codes. As a result, the codes for FreeBSD are stored in the “freebsd” sub-directory, and for Linux, in the “linux” sub-directory. These directory names help the build system easily get the right codes for the compilation. Code Listing 18 shows a part of the main build script. As you can see, most of the additional directories have been defined statically except for “fops_impl” and “mod_traps” where hefesto.sys.os_name() was used. This build system’s function will return “freebsd” when on FreeBSD and Linux when on “linux”.

```
dev-enigma.prologue() {
    cleaner();

    device_installer();

    $includes.add_item(hefesto.sys.pwd());

$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(), "eel"));
$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(), "ebuf"));

$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(), "dev_ctx"));

$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(),
        hefesto.sys.make_path("fops_impl",
hefesto.sys.os_name())));

$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(),
        hefesto.sys.make_path("mod_traps",
hefesto.sys.os_name())));

$includes.add_item(hefesto.sys.make_path(hefesto.sy
s.pwd(), "enigmactl"));
}
```

Listing 18: How the codes are scanned for the compilation

Also, it is vital to abstract the build issues when changing platforms. The user should not worry about these complications when just compiling and using their code. In /dev/enigma, the build system allows a conditional code inclusion. Based on the current platform, we can easily change the build toolset. As a result, the user can build a platform dependent code straightforwardly and transparently. Much more was done using the related build system, but it is out of the article's scope. Thus, we will stop here.

Conclusions

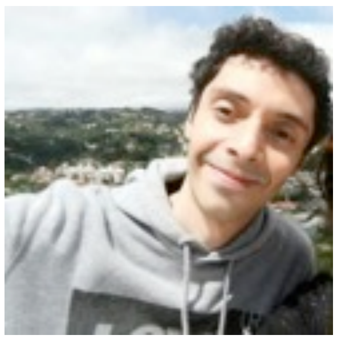
The discussed driver shows that to write a device driver is not rocket science. It also shows how powerful the C language is. The internal *DSL* implemented using C Macros guides the programmers across the device driver code development, making the entire process much easier. C pointers and especially function pointers are also powerful. You can read more about the power of the C pointers in the book *"Beautiful Code"*, more specifically in the chapter *"Another Level of Indirection"* by Diomidis Spinellis. This article also reveals the importance of abstraction when coding something and shows that a real good abstraction is much more than simply creating tons of Classes in an OO fashioned way.

If you liked this article and now interested in device driver programming, you should start reading the following books: *"FreeBSD Device Drivers: A guide for the Intrepid"* by Joseph Kong and *"Linux Device Drivers"* by Alessandro Rubini.

If you are interested in knowing more about cryptography, a nice introduction best suited for you can be found in *"The Code Book"* by Simon Singh. Are you searching for advanced topics? *"Applied Cryptography"* by Bruce Schneier, *"Handbook of Applied Cryptography"* by Menezes, Oorschot and Vanstone and *"Understanding Cryptography"* by Paar and Pelzl would be worth your time.

If you liked /dev/enigma, you can download the code at https://github.com/rafael-santiago/dev-enigma. If you are searching for a library to implement your Enigma simulator, maybe my library at https://github.com/rafael-santiago/eel could be useful to you.

Remember: the Enigma is an outdated cipher. You should not use it for secrecy anymore but just for fun.



About the Author

Rafael Santiago de Souza Netto is a Computer Scientist from Brazil. His main areas of interest are Programming, Computer Networks, Operating Systems, UNIX culture, Compilers, Cryptography, Information Security, Social Coding. He has been working as Software Developer since 2000. You can find him at GitHub (as rafael-santiago).

ServerU

Rack-mount networking server

Designed for BSD and Linux Systems

DESIGNED FOR FreeBSD | DESIGNED FOR GNU / Linux | DESIGNED FOR PRO apps FreeBSD Enterprise Appliance | DESIGNED FOR pfSense

Designed. Certified. Supported

Up to **5.5Gbit/s** routing power!

KEY FEATURES

- 6 NICs w/ Intel igb(4) driver w/ bypass
- Hand-picked server chipsets
- Netmap Ready (FreeBSD & pfSense)
- Up to 14 Gigabit expansion ports
- Up to 4x10GbE SFP+ expansion

PERFECT FOR

- BGP & OSPF routing
- Firewall & UTM Security Appliances
- Intrusion Detection & WAF
- CDN & Web Cache / Proxy
- E-mail Server & SMTP Filtering

contactus@serveru.us | www.serveru.us
8001 NW 64th St. Miami, LF 33166 | +1 (305) 421-9956

Fluentd for Centralizing Logs

In this article, I will talk about how to manage diverse and disparate logs on FreeBSD servers. As system administrators, when we want to know which services are disabled or not running, we check our logs in /var/log. The most useful commands we can use to check if the services are running in FreeBSD are “ps” and “tail”. As an example, we want to know if our web server (e.g. Nginx) is running. Using a combination with “grep” command, we can do something similar to what has been described below. Here is the result from those commands.

```
$ ps ax | grep “nginx”

  726   -  Is      0:00.00 nginx: master
process /usr/local/sbin/nginx

  728   -  S       1:18.10 nginx: worker
process (nginx)

  729   -  S       1:23.45 nginx: worker
process (nginx)

86660   0  S+      0:00.00 grep nginx
```

There are five columns in the result, and we can assure that our web server is running well. If no result after we executed the commands then it means that our web server is not running. Start the web server using this command.

```
$ sudo service nginx start

Performing sanity check on nginx
configuration:

nginx: [emerg] unknown directive "ttp" in
/usr/local/etc/nginx/nginx.conf:19

nginx: configuration file
/usr/local/etc/nginx/nginx.conf test
failed

Starting nginx.

nginx: [emerg] unknown directive "ttp" in
/usr/local/etc/nginx/nginx.conf:19

/usr/local/etc/rc.d/nginx: WARNING: failed
to start nginx
```

Oops! Our Nginx web server failed to start. Let’s check the log to see what the underlying problem is:

```
$ sudo tail -f /var/log/nginx-error.log

.....

2017/03/09 08:54:03 [emerg] 94786#0:
invalid number of arguments in "root"
directive in
/usr/local/etc/nginx/nginx.conf:311

2017/06/01 04:03:15 [emerg] 60819#0:
unknown directive "ttp" in
/usr/local/etc/nginx/nginx.conf:19

2017/06/01 04:03:15 [emerg] 60820#0:
unknown directive "ttp" in
/usr/local/etc/nginx/nginx.conf:19

     .....
```

If you are an experienced system administrator, you can handle that simple error and fix it in no time. However, imagine if we were not responsible for one service, but hundreds or thousands of services. Tracking down errors on various machines and different log files would be time consuming and frustrating. There are some solutions for managing logs at scale. There are open-source monitoring applications like Zabbix and SolarWinds. You can use them if you want, but in this article, I will show you how to integrate our logs so that in the future, we can analyze it for general purpose.

Fluentd Architecture

I will introduce you to Fluentd (http://www.fluentd.org/), an open-source data collector. Fluentd is very modular and we can integrate almost any logs and unify the logs and processing them. **Figure 1** shows you Fluentd's architecture, and gives you a sense of how you can make your logs easier to manage.

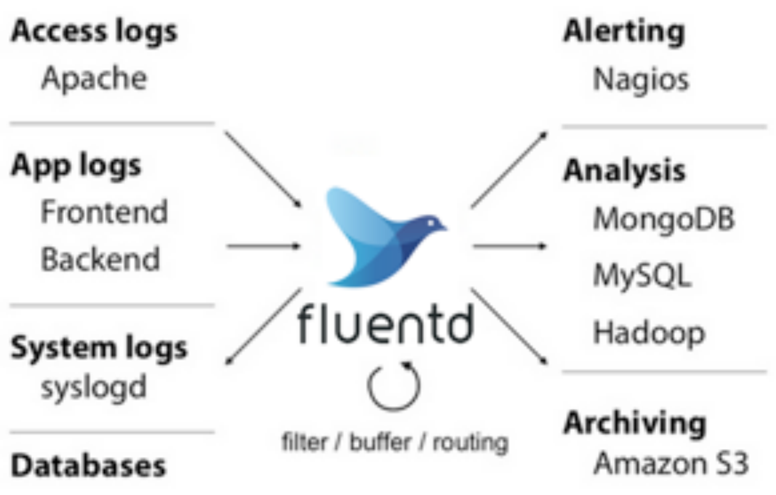


Figure 1. Fluentd architecture (source: https://www.fluentd.org)

Fluentd Installation

Let us install Fluentd on FreeBSD server. I'm using FreeBSD 10.3 with 1GB RAM. There are two types installation we can use, first by using ports collection (`/usr/ports`) or second you can compile from the source. In this case, we are going to use ports for our installation. Remember, before we try to install from the ports, we must update our ports.

```
$ cd /usr/ports

$ sudo portsnap fetch update
```

Wait for a couple minutes until our ports are fresh with new updates. After that, we can try to find if Fluentd exist in the ports collection. Use make search name="<key>". Still in the current directory of /usr/ports, execute this command.

```
$ make search name="fluentd"
```

Depending on your FreeBSD, it may display different results. Below, you can see that there are two different packages of Fluentd.

```
Port:   rubygem-fluentd-0.12.14_1

Path:
/usr/ports/sysutils/rubygem-fluentd

Info:   Fluent event collector

Maint:  kuriyama@FreeBSD.org

B-deps: indexinfo-0.2.6
libedit-3.1.20170329_2,1 libexecinfo-1.1_3
libffi-3.2.1
libyaml-0.1.6_2 ruby-2.3.4,1
ruby23-gems-2.6.12

R-deps: indexinfo-0.2.6
libedit-3.1.20170329_2,1 libexecinfo-1.1_3
libffi-3.2.1
libyaml-0.1.6_2 ruby-2.3.4,1
ruby23-gems-2.6.12 rubygem-cool.io-1.2.4_1
rubygem-
http_parser.rb-0.6.0
rubygem-iobuffer-1.1.2 rubygem-json-2.1.0
rubygem-msgpack-0
.5.12 rubygem-sigdump-0.2.4
rubygem-thread_safe-0.3.6
rubygem-tzinfo-1.2.3 rubyg
em-yajl-ruby-1.3.0

WWW:    http://fluentd.org/

Port:   rubygem-fluentd010-0.10.61

Path:
/usr/ports/sysutils/rubygem-fluentd010

Info:   Fluent event collector

Maint:  kuriyama@FreeBSD.org

B-deps: indexinfo-0.2.6
libedit-3.1.20170329_2,1 libexecinfo-1.1_3
libffi-3.2.1
libyaml-0.1.6_2 ruby-2.3.4,1
ruby23-gems-2.6.12

R-deps: indexinfo-0.2.6
libedit-3.1.20170329_2,1 libexecinfo-1.1_3
libffi-3.2.1
libyaml-0.1.6_2 ruby-2.3.4,1
ruby23-gems-2.6.12 rubygem-cool.io-1.2.4_1
rubygem-
http_parser.rb-0.6.0
rubygem-iobuffer-1.1.2 rubygem-json-2.1.0
rubygem-msgpack-0
.5.12 rubygem-sigdump-0.2.4
rubygem-yajl-ruby-1.3.0

WWW:    http://fluentd.org/
```

The newer and stable version is v0.12. Therefore, we will install using the stable version. Go to `/usr/ports/sysutils/rubygem-fluentd` folder and install using `make install` command. There is one dependency, `rubygems-tzinfo-data,` and is not included in this stable version. Let's go to /usr/ports/devel and install the dependency.

```
$ cd /usr/ports/devel/rubygems-tzinfo-data

$ sudo make install
```

After that we can start install rubygem-fluentd

```
     $ cd
/usr/ports/sysutils/rubygem-fluentd

     $ sudo make install
```

For now, accept the default installation. Fluentd will install `ruby` , the language it was   written in. If you don't have any Ruby installation, the ports will automatically download and install it. Sometimes, we could be having it in our server. For my case, I got an error. We can either uninstall the old Ruby or we can upgrade it by executing `make reinstall` command. In this article, I prefer to uninstall the old Ruby.

I don't have any issue removing my old version of Ruby (`/usr/ports/lang/ruby22`). Change our current directory to `/usr/ports/lang/ruby22` and execute this command `make deinstall` to uninstall old Ruby. Then, execute `make install` in `/usr/ports/sysutils/rubygem-fluentd.`

Configure fluentd

Every installation uses the ports located in `/usr/local,` and the configuration in `/usr/local/etc`. Fluentd's configuration is in `/usr/local/etc/fluentd`. There are six directives we can use for configuration:

- **source** for determining input sources
- **match** for output destinations
- **filter** for event processing pipelines
- **system** for system wide configuration
- **label** for group output and filter for internal routing
- **@include** for include other files

In this case, we now want to make our Nginx logs to be recorded by Fluentd. Type the following configuration in fluentd.conf.

```
<source>

  type tail

  format nginx

  path /var/log/nginx-access.log

  tag nginx.access

</source>
```

Next, we start the fluentd daemon using this command: `sudo /usr/local/bin/fluentd.`

```
     $ sudo /usr/local/bin/fluentd -c
/usr/local/etc/fluentd/fluent.conf
```

We only have one log, the nginx-access.log. You can put another log into fluentd like syslog and etc. To make our fluentd auto start in the next reboot, we should put this flag, `fluentd_enable="YES"` in `/etc/rc.conf.` To start or stop fluentd, we don't need the long command like we used above. The following start and stop commands can do it.

```
     $ sudo service fluentd stop

     $ sudo service fluentd start
```

Debugging fluentd

We know that our fluentd service is working by using `ps ax` command. However, how sure are we that our log is working and read by fluentd?. First, stop our fluentd service and then execute the command by manually using the following command.

```
     $ sudo /usr/local/bin/fluentd -c
/usr/local/etc/fluentd/fluent.conf

2017-06-09 13:30:54 +0000 [info]: reading
config file
path="/usr/local/etc/fluentd/fluent.conf"

2017-06-09 13:30:54 +0000 [info]: starting
fluentd-0.12.14
```

```
2017-06-09 13:30:54 +0000 [info]: gem
'fluentd' version '0.12.14'

2017-06-09 13:30:54 +0000 [info]: adding
match pattern="debug.**" type="stdout"

2017-06-09 13:30:54 +0000 [info]: adding
source type="forward"

2017-06-09 13:30:54 +0000 [info]: adding
source type="monitor_agent"

2017-06-09 13:30:54 +0000 [info]: adding
source type="syslog"

2017-06-09 13:30:54 +0000 [info]: using
configuration file: <ROOT>

  <source>

    @type forward

    port 24224

  </source>

  <source>

    @type monitor_agent

    bind 0.0.0.0

    port 24220

  </source>

  <source>

    @type syslog

    port 5140

    bind 0.0.0.0

    tag system

    with_priority true

  </source>

  <match debug.**>

    @type stdout

  </match>

</ROOT>

2017-06-09 13:30:54 +0000 [info]:
listening fluent socket on 0.0.0.0:24224
```

Now, open another terminal and use fluent-cat command to send our message to fluentd. Command fluent-cat is useful for debugging and testing if our configuration or fluentd is running well.

```
     $ echo '{"message":"hello"}' | sudo
/usr/local/bin/fluent-cat --host <your_ip>
--port 24224 debug
```

Remember to change <your_ip> with your ip address. After executing the command, a "hello" message will be displayed in fluentd. The rest is shown below.

```
2017-06-09 13:30:54 +0000 [info]:
listening fluent socket on 0.0.0.0:24224

2017-06-09 13:33:34 +0000 debug:
{"message":"hello"}
```

That is all. If you want to analyze the log, you can add ElasticSearch and Kibana for advanced analytics.

Conclusions

Using fluentd allows developers, data analysts or system administrators to utilize the logs as they are generated. Logs are important but in some entries, we want our logs not giving false information or bad interpretation. As logs quickly iterate, we can make our data better and easier to manage in the future.

About the Author

Andrey Ferriyan is a writer, researcher and practitioner. He is among the Python and R enthusiasts. His experience is in UNIX-like servers (GNU/Linux, FreeBSD and OpenBSD). Being a Data Scientist wannabe, his area of interests includes Information Security, Machine Learning and Data Mining. Currently, He is a student at Keio University under the LPDP (Indonesia Endowment Fund for Education). He leads a startup company in Indonesia called ATSOFT with his friends. And he speaks English and Indonesian.

MINIX3: A Promising Diamond in the Rough

From its humble beginnings as a teaching tool for Andrew Tanenbaum's Operating Systems: Design and Implementation 30 years ago, MINIX has evolved a lot since. Now in its third edition and with a focus on reliability, the raccoon-themed operating system is set to become a serious and dependable UNIX-like contender.

The current version, MINIX 3,   bears only a superficial resemblance to its older siblings. Released in 1987, MINIX 1 was an operating system running on IBM PC-class hardware with a 4.77 MHz 8086 processor, 5 1/4" floppy drives and at least 256 KB of memory. It was designed as a small UNIX V7 educational clone because of AT&T's no-teaching clause at the time. MINIX 1 enjoyed a tremendous following for a period of time until Linus Torvalds created Linux in 1991 after reading Tanenbaum's textbook. MINIX 2 followed in 1997 with networking capabilities, and the whole system was retroactively relicensed under BSD terms in 2000.

Beginning with MINIX 3 in 2005, the operating system acquired many features that UNIX programmers take for granted, like virtual memory, dynamic linking or pkgsrc support. The most radical change in recent history was the replacement of the vintage, UNIX Version 7-flavored userland with NetBSD's, giving it a much needed modern POSIX implementation. With an emphasis on reliability and embedded systems, MINIX has far outgrown its original teaching purposes.

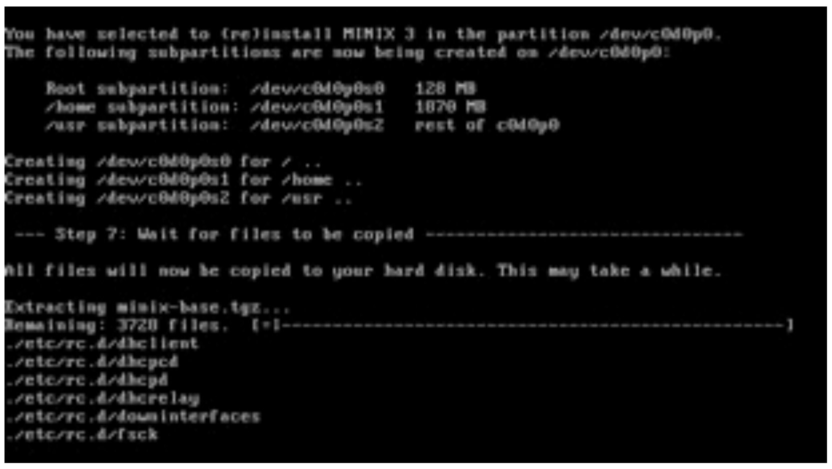
Why MINIX 3?

The most distinctive feature of MINIX 3 is its multi-server architecture, where drivers and other components are running as user-land processes on top of a micro-kernel. Compared to the monolithic kernel approach of Linux and the BSDs', this massively reduces the amount of code running with kernel privileges. While it does not reduce the number of bugs, it reduces their power: a crash in a network card driver will not bring down the system or allow a hacker to gain complete control of it. Given that most program sources have statistically at least 3 bugs per thousand lines of code, MINIX's small TCB (between 15 to 30k lines of code) is refreshingly small and reassuring when compared to the tens of millions lines inside the Linux kernel.

MINIX is not the only micro-kernel operating system around. Other open-source projects include GNU Hurd, HelenOS, Redox, Google's Fuchsia and the L4 family, the last of which has seen over a billion of deployments hidden inside mobile phone modem chips. Proprietary products include VxWorks, QNX and INTEGRITY which are all well-entrenched in embedded and highly reliable applications.

Installing MINIX 3

MINIX 3 runs on both i386 and ARM. While it can run on physical hardware, MINIX is nowhere near the hardware support level of more mainstream operating systems (the lack of USB support on x86 makes an installation on modern, legacy-free consumer hardware a challenging task). It is thus recommended for newcomers to install it on a virtual machine first. We'll be using QEMU in this article, but most other virtualization platforms should work.



```
You have selected to (re)install MINIX 3 in the partition /dev/c0d0p0.
The following subpartitions are now being created on /dev/c0d0p0:

    Root subpartition:  /dev/c0d0p0s0   128 MB
    /home subpartition: /dev/c0d0p0s1   1870 MB
    /usr subpartition:  /dev/c0d0p0s2   rest of c0d0p0

Creating /dev/c0d0p0s0 for / ..
Creating /dev/c0d0p0s1 for /home ..
Creating /dev/c0d0p0s2 for /usr ..

 --- Step 7: Wait for files to be copied -------------------------------

All files will now be copied to your hard disk. This may take a while.

Extracting minix-base.tgz...
Remaining: 3720 files.  [=]-------------------------------------------------]
./etc/rc.d/dhclient
./etc/rc.d/dhcpcd
./etc/rc.d/dhcpd
./etc/rc.d/dhcrelay
./etc/rc.d/downinterfaces
./etc/rc.d/fsck
```

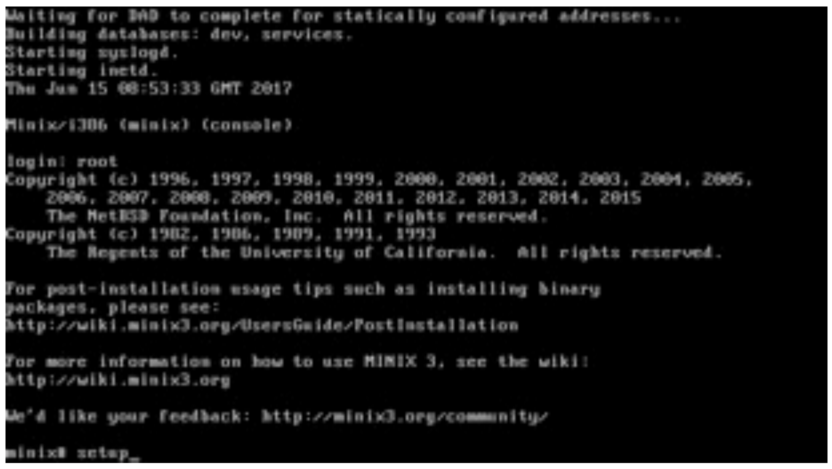
The latest stable release (3.3.0) is starting to get rather old, so instead we'll be using the latest release candidate. Head over to www.minix3.org, click on Download, scroll down to Development snapshots and download the most recent ISO (minix_R3.4.0rc6-d5e4fc0.iso.bz2 at the time this article was written), then decompress it with bunzip2. We'll need a hard disk image to install MINIX 3 onto. So, let's create one:

```
qemu-img create -f qcow2 minix3.hdd 10G
```

Now, we are ready to install MINIX 3 on the virtual machine. Start the installation CD:

```
qemu-system-i386 -enable-kvm -hda
minix3.hdd -cdrom
minix_R3.4.0rc6-d5e4fc0.iso
```

Thereafter, follow the instructions printed on the screen: select the default boot option, log in as root and run the setup command. Answer the questions (in case of doubt, the default settings are fine), wait for the installer to finish and restart the machine with shutdown (-r) now.

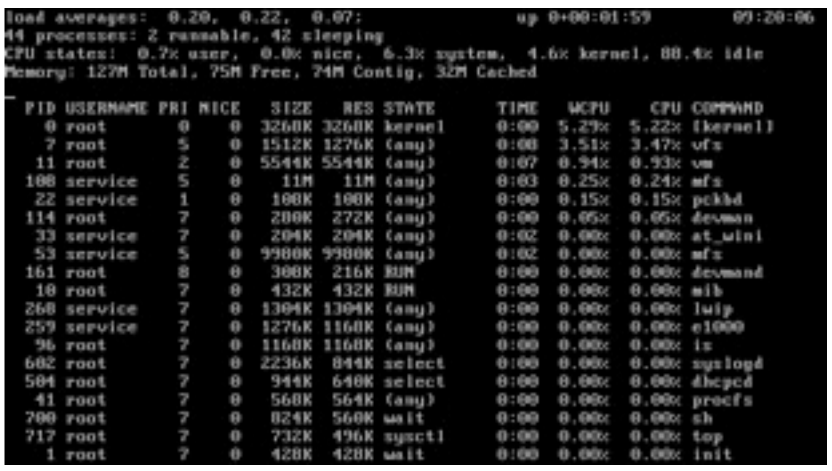


```
Waiting for DAD to complete for statically configured addresses...
Building databases: dev, services.
Starting syslogd.
Starting inetd.
Thu Jun 15 08:53:33 GMT 2017

Minix/i386 (minix) (console)

login: root
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
    2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

For post-installation usage tips such as installing binary
packages, please see:
http://wiki.minix3.org/UsersGuide/PostInstallation

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org

We'd like your feedback: http://minix3.org/community/

minix# setup_
```

Command-line interface

MINIX 3 is now up and running on your virtual machine. Log in as root and the system is yours. Thanks to its NetBSD userland, MINIX 3 feels like a modern POSIX system, at least for most non-administrative tasks. For good measure, set a password for root with passwd.

Let's explore around a bit. Run the top command to get an interactive view of the running processes:



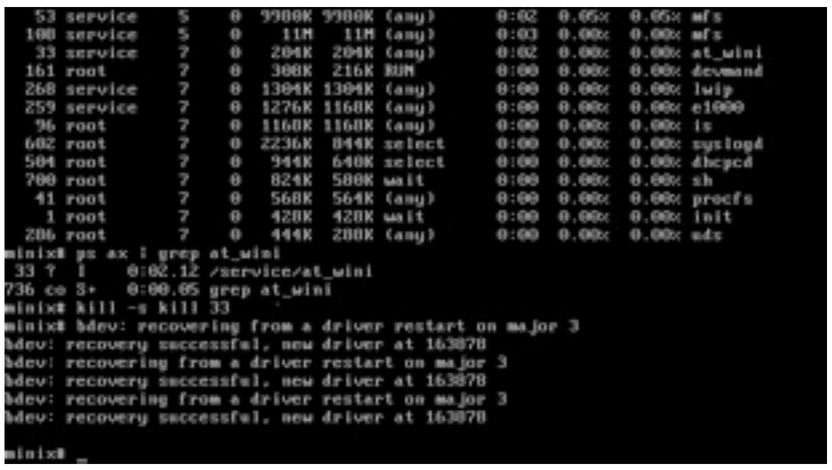
```
load averages:  0.20,  0.22,  0.07;                up 0+00:01:59        09:20:06
44 processes: 2 runnable, 42 sleeping
CPU states:  0.7% user,  0.0% nice,  6.3% system,  4.6% kernel, 88.4% idle
Memory: 127M Total, 75M Free, 74M Contig, 32M Cached
_
  PID USERNAME PRI NICE   SIZE    RES STATE       TIME   WCPU    CPU COMMAND
    0 root       0    0  3260K  3260K kernel      0:00  5.29%  5.22% [kernel]
    7 root       5    0  1512K  1276K (any)       0:00  3.51%  3.47% vfs
   11 root       2    0  5544K  5544K (any)       0:07  0.94%  0.93% vm
  108 service    5    0    11M    11M (any)       0:03  0.25%  0.24% mfs
   22 service    1    0   108K   108K (any)       0:00  0.15%  0.15% pckbd
  114 root       7    0   280K   272K (any)       0:00  0.05%  0.05% devman
   33 service    7    0   204K   204K (any)       0:02  0.00%  0.00% at_wini
   53 service    5    0  9900K  9900K (any)       0:02  0.00%  0.00% mfs
  161 root       8    0   308K   216K RUN         0:00  0.00%  0.00% devmand
   10 root       7    0   432K   432K RUN         0:00  0.00%  0.00% mib
  268 service    7    0  1304K  1304K (any)       0:00  0.00%  0.00% lwip
  259 service    7    0  1276K  1160K (any)       0:00  0.00%  0.00% e1000
   96 root       7    0  1160K  1160K (any)       0:00  0.00%  0.00% is
  602 root       7    0  2236K   844K select      0:00  0.00%  0.00% syslogd
  504 root       7    0   944K   640K select      0:00  0.00%  0.00% dhcpcd
   41 root       7    0   568K   564K (any)       0:00  0.00%  0.00% procfs
  700 root       7    0   824K   560K wait        0:00  0.00%  0.00% sh
  717 root       7    0   732K   496K sysctl      0:00  0.00%  0.00% top
    1 root       7    0   428K   428K wait        0:00  0.00%  0.00% init
```

The output is a bit cramped due to the 80x25 VGA console mode. However, we can see that most subsystems like vfs, vm and pci are running as usermode processes. Exit top with q, then search for the hard disk driver:

```
ps ax | grep at_wini
```

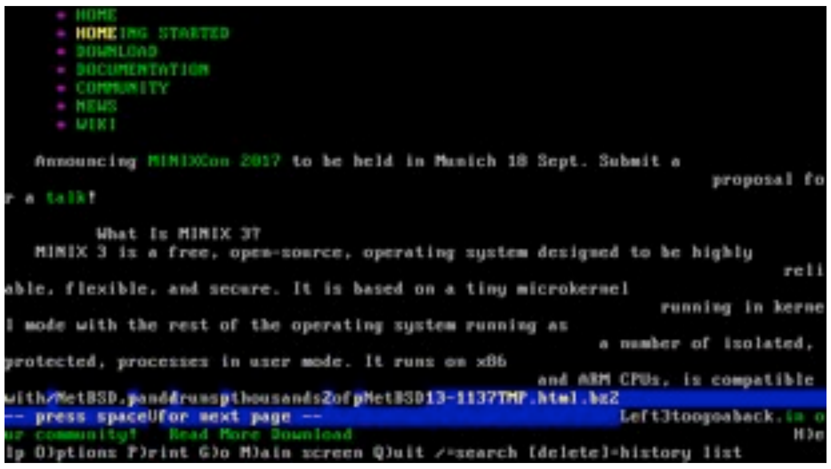
For fun, let's SIGKILL the driver:

```
kill -s kill $(PID_OF_AT_WINI)
```



```
   53 service    5    0  9900K  9900K (any)       0:02  0.05%  0.05% mfs
  108 service    5    0    11M    11M (any)       0:03  0.00%  0.00% mfs
   33 service    7    0   204K   204K (any)       0:02  0.00%  0.00% at_wini
  161 root       7    0   308K   216K RUN         0:00  0.00%  0.00% devmand
  268 service    7    0  1304K  1304K (any)       0:00  0.00%  0.00% lwip
  259 service    7    0  1276K  1160K (any)       0:00  0.00%  0.00% e1000
   96 root       7    0  1160K  1160K (any)       0:00  0.00%  0.00% is
  602 root       7    0  2236K   844K select      0:00  0.00%  0.00% syslogd
  504 root       7    0   944K   640K select      0:00  0.00%  0.00% dhcpcd
  700 root       7    0   824K   580K wait        0:00  0.00%  0.00% sh
   41 root       7    0   568K   564K (any)       0:00  0.00%  0.00% procfs
    1 root       7    0   428K   428K wait        0:00  0.00%  0.00% init
  286 root       7    0   444K   288K (any)       0:00  0.00%  0.00% mds
minix# ps ax | grep at_wini
 33 ?  I     0:02.12 /service/at_wini
736 co S+    0:00.05 grep at_wini
minix# kill -s kill 33
minix# bdev: recovering from a driver restart on major 3
bdev: recovery successful, new driver at 163878
bdev: recovering from a driver restart on major 3
bdev: recovery successful, new driver at 163878
bdev: recovering from a driver restart on major 3
bdev: recovery successful, new driver at 163878

minix# _
```

The reincarnation server will automatically restart the deceased driver, and the system will carry on as if nothing happened. This demonstrates the self-healing capabilities of MINIX 3 where the operating system can survive driver failures. That being said, not all server crashes are survivable or at least transparent. If VM, PM, VFS or another core service fails, the system will panic. An audio driver crash will cause a short but noticeable sound glitch.



```
     • HOME
     • HOMEING STARTED
     • DOWNLOAD
     • DOCUMENTATION
     • COMMUNITY
     • NEWS
     • WIKI

   Announcing MINIXCon 2017 to be held in Munich 18 Sept. Submit a
                                                                proposal fo
r a talk!

         What Is MINIX 3?
   MINIX 3 is a free, open-source, operating system designed to be highly
                                                                       reli
able, flexible, and secure. It is based on a tiny microkernel
                                                              running in kerne
l mode with the rest of the operating system running as
                                                       a number of isolated,
protected, processes in user mode. It runs on x86
                                                   and ARM CPUs, is compatible
with/NetBSD.panddrunspthousands2ofpNetBSD13-1137TMP.html.bz2
-- press spaceUfor next page --                              Left3toogoaback.in o
ur community!    Read More Download                                          H)e
lp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

Nevertheless, it is a testament of the level of MINIX's reliability that few mainstream operating systems can match. Among the list of published papers available on the MINIX 3 website, some of them are quite interesting. One experiment in particular [http://www.minix3.org/docs/jorrit-herder/dsn-dccs09.pdf] is worthy of note. To test MINIX 3's reliability, hundreds of thousands of faults were injected at runtime in a network driver. The driver crashed a lot, sometimes the PCI bus locked up, but the kernel and the rest of the system never crashed.

Another advantage with user-space components is that they can be debugged (almost) like any other user program. There's no need to reboot the computer during the development cycle for non-essential services, and there's no kernel-related debugging issues to watch out for. In some ways, this is similar with NetBSD's rump Anykernel, where drivers can be run, developed and debugged as user processes.

There are four virtual terminals which are accessible with the keystrokes Alt-<Fn>. Being root is nice, but using an unprivileged account is less dangerous: create one with useradd (-m) and set its password with passwd if you wish.

Software packages

MINIX 3 (and by extension NetBSD) is rather barren when freshly installed. The only two supplied editors are vi and mined, let's fix that. First, we need to grab the list of available packages with `pkgin update` before we can list all of them with `pkgin available` and perform installation with `pkgin install`.

So, to install vim, simply run `pkgin install vim`. Unfortunately for emacs lovers, their favorite program has not yet been ported. But rest assured that the Editors War is thankfully not settled with the availability of nano.

Over 7,600 packages are available, but only a small number of them are tested at this point. MINIX lacks most of the heavy-weight ones like KDE, Firefox or LibreOffice. However, there are usually simpler alternatives you can choose from. You can, for example, use the text web browser lynx, but MINIX's tty subsystem can't quite keep up with it: it's time to bring out X11.

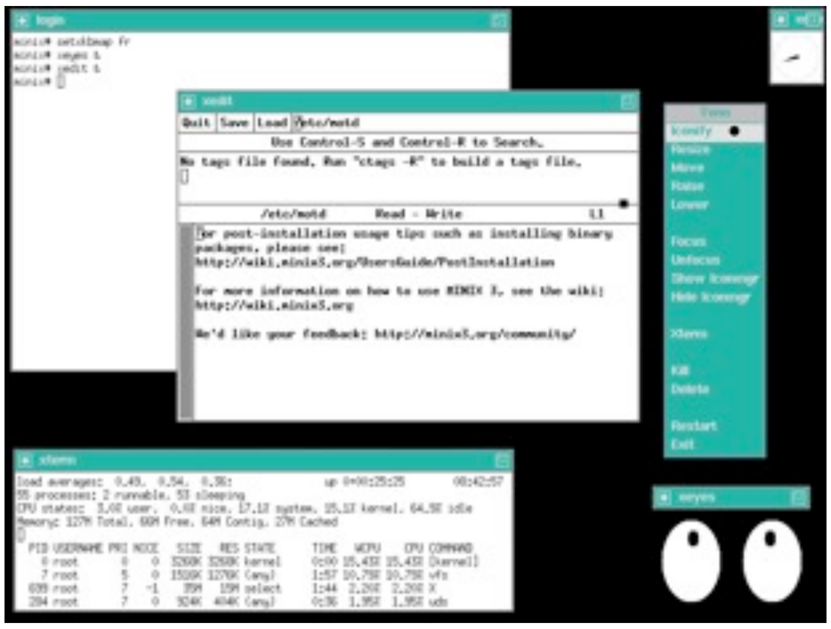
X11 and GUIs

MINIX 3 ships with X11 out of the box but the auto-detected settings for QEMU are not optimal. Generate a configuration file with Xorg -configure and edit the file. I have rewritten the "Screen" section as follows:

```
Section "Screen"
     Identifier "Screen0"
     Device     "Card0"
     Monitor    "Monitor0"
     SubSection "Display"
          Modes "800x600"
          Depth 24
     EndSubSection
EndSection
```

Move the file over to /etc/X11/xorg.conf and X11 is ready to go with startx. If for some reason you have issues with the mouse, restart QEMU with the environment variable SDL_VIDEO_X11_DGAMOUSE set to 0. As with NetBSD, the default window manager is TWM, usable but quite the throwback to the eighties. Here is a crash course on how to use it with the default settings:

- you can drag windows around with the title bar;
- the upper-left icon on a title bar will iconify the window, the upper-right icon will resize it;
- click on an empty space to bring up a menu of actions;
- focus is given when the mouse is moved over a window;
- look out for the xterm window with the title "login": if you quit this window, X11 will exit and you will be dropped back to console mode.

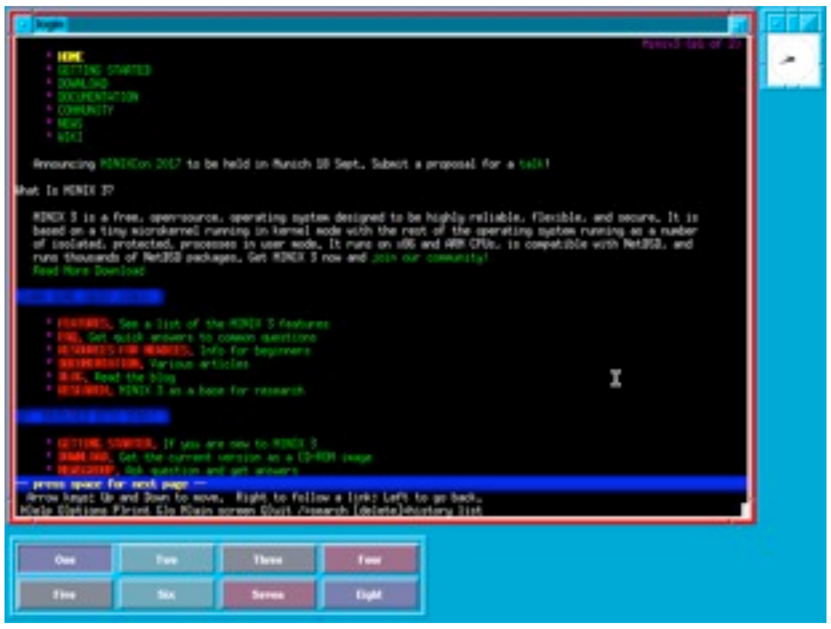


```
minix# setxkbmap fr
minix# xeyes &
minix# xedit &
minix#
```

```
/etc/motd
Use Control-S and Control-R to Search.
No tags file found. Run "ctags -R" to build a tags file.

/etc/motd        Read - Write        L1
For post-installation usage tips such as installing binary
packages, please see:
http://wiki.minix3.org/UsersGuide/PostInstallation

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org

We'd like your feedback: http://minix3.org/community/
```

```
load averages:  0.49,  0.54,  0.36;                 up 0+00:25:25      00:42:57
55 processes: 2 runnable, 53 sleeping
CPU states:  3.0% user,  0.4% nice, 17.1% system, 15.1% kernel, 64.5% idle
Memory: 127M Total, 60M Free, 64M Contig, 27M Cached

  PID USERNAME PRI NICE   SIZE   RES STATE      TIME   WCPU    CPU COMMAND
    0 root       0    0  3268K 3268K kernel     0:00 15.43% 15.43% [kernel]
    7 root       5    0  1536K 1276K (any)      1:57 10.79% 10.79% vfs
  609 root       7   -1    35M   15M select     1:44  2.20%  2.20% X
  284 root       7    0   324K  404K (any)      0:36  1.95%  1.95% uds
```

If you have an AZERTY keyboard like me, you can change the X11 keyboard map with setxkbmap fr. While the main use of X11 under MINIX 3 is having access to as many xterm windows as we need, the rest of the basic X11 utilities are available too: xedit, xeyes, xclock etc. Also, console utilities that the MINIX 3 tty layer can't handle will properly work under xterm. Therefore, you can finally browse the Web with lynx. Remember to postfix your X11 commands with & in order not to block the terminals.



```
                                                          Minix3 (p1 of 3)
   * HOME
   * GETTING STARTED
   * DOWNLOAD
   * DOCUMENTATION
   * COMMUNITY
   * NEWS
   * WIKI

  Announcing MINIXCon 2017 to be held in Munich 18 Sept. Submit a proposal for a talk!

What Is MINIX 3?

  MINIX 3 is a free, open-source, operating system designed to be highly reliable, flexible, and secure. It is
  based on a tiny microkernel running in kernel mode with the rest of the operating system running as a number
  of isolated, protected, processes in user mode. It runs on x86 and ARM CPUs, is compatible with NetBSD, and
  runs thousands of NetBSD packages. Get MINIX 3 now and join our community!
  Read More Download

   * FEATURES. See a list of the MINIX 3 features
   * FAQ. Get quick answers to common questions
   * RESOURCES FOR NEWBIES. Info for beginners
   * DOCUMENTATION. Various articles
   * BLOG. Read the blog
   * RESEARCH. MINIX 3 as a base for research

   * GETTING STARTED. If you are new to MINIX 3
   * DOWNLOAD. Get the current version as a CD-ROM image
   * NEWSGROUP. Ask question and get answers
-- press space for next page --
  Arrow keys: Up and Down to move.  Right to follow a link; Left to go back.
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list
```

The default X11 environment is rather bland, so let's personalize it. Copy /etc/X11/xinit/xinitrc to /root/.xinitrc and edit it. This is the shell script that is launched with X11. You should at least set your X11 keymap here with setxkbmap and perhaps change twm for ctwm. If you want to get rid of the login xterm, exec your window manager at the end instead of xterm. To personalize twm or ctwm, check out their man pages, or you can try other window managers from pkgin instead.

Help, I'm stuck!

If you hit a roadblock, look out for documentation. The MINIX 3 website has lots of material. In particular, a wiki at www.wiki.minix3.org and manpages at www.man.minix3.org are available. Remember that MINIX 3 imported the NetBSD userland. Hence, most of the NetBSD guide applies for non-administrative tasks too. You can reach out to the MINIX 3 community at the minix3 Google group or at the various IRC channels listed on the website. Don't worry, we don't bite :-)

Conclusions

Despite its various limitations (no kernel threads, no SMP, no 64-bit support, few device drivers etc.), MINIX 3 is a refreshing alternative take on operating systems when compared to the mainstream. There are lots of other topics to talk about (architecture, live updates, development, current projects, to name but a few) that unfortunately can't fit inside an introduction. While its reliability features are impressive and the efforts to turn it into a serious system are real, MINIX 3 still has an educational value for those who want to take a peek behind the scenes. MINIX doesn't boast of many developers as Linux or the BSDs. However, don't confound lack of development manpower with lack of maturity. MINIX 3 is a diamond in the rough and everyone is invited to help polish it.

Useful Links

http://minix3.org - Main website
http://minix3.org/conference/2017/ - MINIXcon 2017
https://www.youtube.com/watch?v=MG29rUtvNXg - Tanenbaum talk from 2015 about MINIX 3
http://wiki.minix3.org/doku.php?id=publications - List of academical papers about MINIX 3

About the Author

Jean-Baptiste Boric is your typical French geek student at ENSIMAG. Whenever he's not busy procrastinating, he tinkers with computers and electronics. His notable achievements to date include various contributions to the MINIX 3 project and a third-party firmware for the HP Prime calculator. Also, he definitively did not submit a buggy pull request once that deleted the entire source tree and Git repository of a MINIX 3 core developer.

Interview with Professor Andrew Tanenbaum, The Creator of MINIX3

Andrew Stuart Tanenbaum is an American computer scientist and professor emeritus of computer science at the Vrije Universiteit Amsterdam in the Netherlands. He is best known as the author of MINIX, a free Unix-like operating system for teaching purposes, and for his computer science textbooks, regarded as standard texts in the field. He regards his teaching job as his most important work. Since 2004 he has operated Electoral-vote.com, a website dedicated to analysis of polling data in federal elections in the United States.

Can you tell our readers about yourself and your role nowadays?

I was a professor at the Vrije Universiteit in Amsterdam for 43 years. I officially retired in 2014, but I am still active. For example, I am still writing books and I am an ACM Distinguished Speaker, so I get invited to various universities to give talks. I am still involved with MINIX, including the planning for the MINIXCon 2017 conference in Munich, Germany, in Sept. 2017.

How you first got involved with programming?

I was a student at MIT. Somehow I learnt that MIT had acquired a PDP-1 minicomputer. I went over to it and asked if I could use it. They said yes and gave me a manual. I basically taught myself to program in PDP-1 assembly language. I also took courses on programming at MIT.

While having a wide field of expertise, which area do you put noticeably more emphasis on and why?

I think my primary interest has always been operating systems. I am particularly concerned about dependability and security since there is so much poor software out there.

You created MINIX. Can you tell us the idea behind it? What was its purpose?

When UNIX V6 was released, it became an instant hit at universities. A professor in Australia, John Lions, wrote a book describing how it worked line by line. Many universities began using the Lions' book in their courses.

When Bell Labs released UNIX V7, the new license said it was forbidden to teach it in classes or write books about it. So, I and many other professors were stuck. I decided to write my own operating system from scratch, without any of the Bell Labs code, so that students could learn how an operating system worked. That led to MINIX being released in 1987. It was an instant hit.

Could you tell more about the correlation between Minix 3 and NetBSD that allows Minix to run thousands of NetBSD packages on it?
After MINIX 3 had been up and running for a couple of years, we decided that there were not enough packages for it. Therefore, we decided to make the outside of it (what the user sees) much more compatible with NetBSD. We also ported many NetBSD headers, libraries, compilers, tool chains, etc. Internally, MINIX 3 is a multi-server system with excellent fault tolerance properties. For example, each device driver runs as a separate user process, outside the kernel. If a driver crashes, it can be replaced on the fly without affecting the system. The result of this design is that one effectively has a fault-tolerant system that can run NetBSD packages.

Can MINIX run FreeBSD packages too?
We have never tested for this. Any FreeBSD package that can run on NetBSD will probably work.

What is your most interesting IT issues and why?
My biggest concern is reliability and security. For nearly all users, computers are more than fast enough, but they are still very unreliable compared to televisions and other devices. I think a computer should have a failure mean time of 50 years so that almost no one will ever experience a failure. We are a long way from there, and need a lot of work to get there.

What tools do you use most often and why?
I have a Mac Pro and a MacBook Pro that I use most of the time. I use emacs and the shell a lot for day-to-day work. I run a Website (www.electoral-vote.com) and a lot of the code there is in awk. The thing I really like about awk is that it is stable. Its updates are fairly infrequent. I think the last one was about 30 years ago. I dislike software that changes all the time. As a general rule, each new version of a program is bigger, slower, and less stable than its predecessor, all in the name of adding new features that almost no one wants.

What was the most difficult and challenging implementation you have done so far? Could you give us some details?
MINIX almost didn't happen. It was working pretty well, but would crash after about an hour for no reason and no consistent pattern. I could not figure it out. As a last ditch effort, I wrote a hardware simulator and ran MINIX on the simulator to try to find out what was happening. It ran perfectly on the simulator, just not on the hardware. I could not figure it out. I came within a hair of giving up. If I had given up, there wouldn't have been a Linux since Linus Torvalds was a big MINIX user and built Linux using MINIX. If there was no Linux, there would be no Android, since Android is built on Linux. I told one of my students that MINIX crashed after an hour for no reason and he said he heard that the CPU gave interrupt 15 if it got warm. There was nothing about this in the manual, and of course, the simulator didn't have this interrupt. I changed the code to catch the interrupt and sure enough, that was the problem. But it took me six months to find it.

What future do you see for Open-Source systems?
I think it is being used more and more. A number of companies, especially startups, use open-source. I think it will have a good future.

Do you have any specific goals for the rest of this year?
I think I am going to work on a revision of one of my books. I am certainly going to the MINIXCon 2017 conference and some other conferences. If I get invitations as an ACM Distinguished Speaker, I might give some MINIX talks.

Do you have any untold thoughts that you want to share with the readers?
Gee, I can't think of any off hand. I did write a paper: "Lessons learned from 30 years of MINIX." You can read it at the ACM digital library (acm.org/dl)

What is the best advice you can give to programmers?
Put a tremendous amount of effort into making your code solid, portable, robust, and correct. Don't worry about speed. For most applications, computers are more than fast enough already. But people get very annoyed when software crashes or hangs or doesn't work correctly. Keep the code simple and clean, and don't take shortcuts. Don't optimize it until it is finished and you have measured its speed and determined that it is not fast enough. That rarely happens nowadays. Test it thoroughly. The goal is that it should work perfectly all the time. If you can't achieve that, it isn't a good code.

Thank you

Amid the fever of "fake news" and multiple governments' desire to limit encryption in the light of even more terrorist atrocities, is the core principle of social media and the World Wide Web – that of freedom of expression – coming to an end?

by Rob Somerville

A friend I have known for many years made a very astute and profound observation recently - "One of the greatest forms of inhumanity is shutting down the opinion of others," he said. What was so surprising about this comment was not so much the veracity and truth surrounding the observation, but rather that it was my friend that said it. A retired policeman, to the casual observer, he would not naturally fall into the category of the liberal warrior fighting for the right of freedom of speech. This was what made his comment much more poignant, having recently attempted to broach a taboo subject with a group of people, despite his experience of life - he was closed down and told to shut up and get back in his box. Irrespective of using a reasoned argument, and refusing to stoop to using below the belt tactics of the ad hominem attack or sophistry, he was genuinely shocked at the level of resistance he encountered.

And so it is with new media, the web, and the expansion of technology. We are once again facing that moment in history where the established order is terrified by the forces released now that the Pandora's box of freedom of expression has been democratised. Anyone with an internet connection can now become an overnight sensation – more often than not by appealing to the basest of emotions or refuting the most controversial of issues. The last time this happened on such a scale was when the printing press was invented. The material that was not officially sanctioned was confiscated, burnt, and publishers were fined and imprisoned. Time and again, the same modus operandi has been applied – those that culturally rock the boat are ostracised, demonised, tagged and hopefully dismissed and ignored. The only time that this technique fails with spectacular results is when they (whoever "they" may be) misread the underlying mood of the majority of people, refuse to accept the spiral of silence and honestly believe that they are still in control of the narrative. This has clearly been demonstrated by the shock waves that have rocked the established order with the results of the recent US election, British EU referendum and the recent UK election which resulted in a hung parliament. Anyone who denies the role played by the internet and social media in these outcomes is indeed living in cloud cuckoo land. This is why there exists a panicked move to redress the balance back towards censorship, control and re-establishing "trust".

Fake news is one of these buzz-words that I find truly intellectually offensive. It is only with the onward march of time that the true agenda and bias of traditional media is ever exposed, and like political parties, the traditional press gathers around the totem pole of tribalism and brand. The overriding principle is that of setting your stall to appeal to a particular audience, and certain viewpoints and moral positions are taken as de rigour. Or to put it another way, true, rational quality journalistic objectivism is in a short supply. One man's fact is another man's distorted statistic, and there is little chance of finding a sieve that takes such subjective reality and extracts the valuable objective juices. We are all prisoners of our deceptions. To categorise those who attempt to raise their head above the parapet of the established consensus and voice their opinion – no matter how controversial – as fake, is indeed a crude ad hominem attack. What is important is the chemistry of dialogue, interaction and the forming of understanding, for so often we do not have sufficient empathy to understand the shape and fit of the shoes of our fellow man or woman. Unless we have a common canvas onto which we can all freely share and paint our experiences, the boil of bitterness and isolation can but fester and grow out of sight. Which is why the whole concept of silencing the network of people who feel confident enough to blog, tweet, post from the comfort of their keyboard is such an anathema – no matter how uncomfortable they may make us feel. For it is only when such negative views are aired, that any push back or counter action may be taken. By silencing those we disagree with, not only are we discarding any opportunity to learn something and engage, but we are giving consent for others to silence us.

What we have is in effect is an existential battle between control and expression. The technological genie is out of the bottle, and the more attempts that are made to force him back in and bury the bottle of human advancement, equality and expression in the concrete grave of mediocrity, silence and dumb obedience, the louder the outcry will be. Social media, like all other platforms that allow interactions between human beings, is a volatile place. Time-wasters, trolls, opportunists, the bigoted, uneducated and those with hidden agendas will always be attracted to any platform that gives them credibility or power. In that regard, technology is damned in the same way as religion, politics, capitalism or indeed books. As individuals, it is crucial that we use judgment and discernment as it is so easy to be taken in by our natural biases, and to embrace the siren witch of subjectivity. Ironically, our enemies can prove to be better friends than we possibly believe, exposing weaknesses that we would rather not admit to. Like the troll or the angry man, you need to peel back the visceral response from both perspectives (yours and theirs) to extract any nugget of truth or value, if indeed it exists. It is interesting that from a cultural perspective that both the joker and the clown have value - even in certain card games, the joker has a low and high value.

We are at a major crossroads of history. There is a huge disconnect and disparity in our world, between truth and lies, rich and poor, predator and prey. The only gap between the two is the silence of those that choose not to speak. We must encourage them to share their story and move the narrative from bias to consensus. While there are those that will always subscribe to the principle that the majority are not always right, there are certain core values that we share as a race. Until we can embrace these common values as a starting point, we are lost. And the most important of these is the freedom of expression, to be listened to objectively. Without that, not only have we lost our democracy but our humanity too.

MAGAZINE BSD

Editor in Chief:

Ewa Dudzic

ewa@bsdmag.org

www.bsdmag.org

Contributing:

Jean-Baptiste Boric, Rafael Santiago, Andrey Ferriyan, Natalia Portillo, E.G Nadhan, Daniel Cialdella Converti, Vitaly Repin, Henrik Nyh, Renan Dias, Rob Somerville, Hubert Feyrer, Kalin Staykov, Manuel Daza, Abdorrahman Homaei, Amit Chugh, Mohamed Farag, Bob Cromwell, David Rodriguez, Carlos Antonio Neira Bustos, Antonio Francesco Gentile, Randy Remirez, Vishal Lambe, Mikhail Zakharov, Pedro Giffuni, David Carlier, Albert Hui, Marcus Shmitt, Aryeh Friedman

Top Betatesters & Proofreaders:

Daniel Cialdella Converti, Eric De La Cruz Lugo, Daniel LaFlamme, Steven Wierckx, Denise Ebery, Eric Geissinger, Luca Ferrari, Imad Soltani, Olaoluwa Omokanwaye, Radjis Mahangoe, Katherine Dizon, Natalie Fahey, and Mark VonFange.

Special Thanks:

Denise Ebery

Annie Zhang

Katherine Dizon

Senior Consultant/Publisher:

Paweł Marciniak

Publisher:

Hakin9 Media SK,

02-676 Warsaw, Poland Postepu 17D, Poland
worldwide publishing
editors@bsdmag.org

Hakin9 Media SK is looking for partners from all over the world. If you are interested in cooperation with us, please contact us via e-mail: editors@bsdmag.org

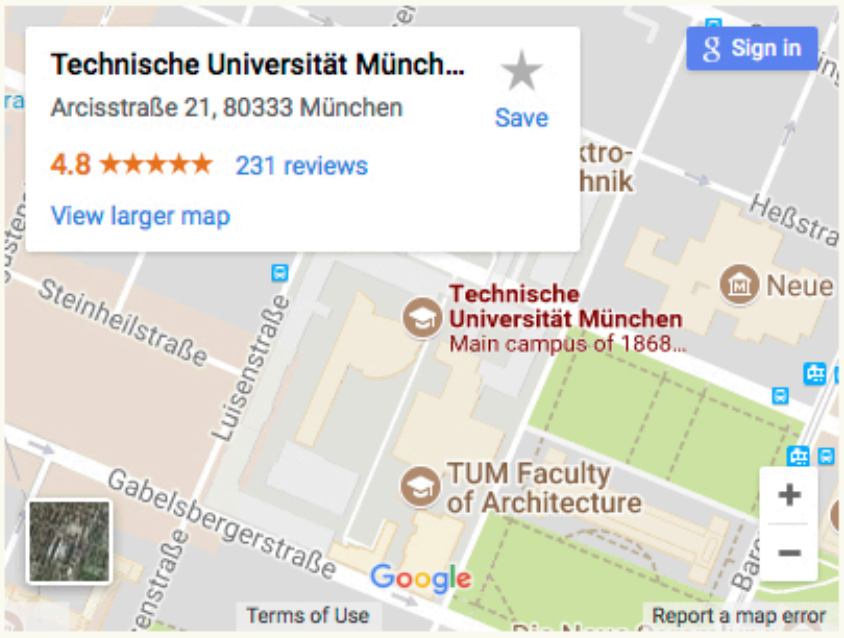
MINIXCon 2017 Announcement

After the successful MINIXCon 2016 in Amsterdam, the MINIXCon steering committee has decided to hold MINIXCon 2017 in Munich, Germany, on Sept. 18, 2017. The Chair of Operating System (Prof. Dr. Uwe Baumgarten) will be our patron from the Technical University Munich. The idea is to exchange ideas and experiences among MINIX 3 developers and users as well as discussing possible paths forward. Future developments will now be done like in any other volunteer-based open-source project. Increasing community involvement is a key issue here. We also will make a special effort to get industry involved.

The videos of the MINIXCon 2016 talks can be found here .

Date, Time, and Location

Monday, 18 September 2017 from 09:00 to 18:00 at the Theresianum, Technical University of Munich.



Giving a Talk

The call for presentations is now open. To propose a talk, please see the Call for Proposals page:

www.minix3.org/conference/2017/cfp.html

Registration

To attend the conference, you need to register here. The early registration fee is 50 euro (20 euro for students) for registrations before 4 Sept. 2017. After that, it is 75 euro for regular and 30 euro for students. This includes morning and afternoon coffee breaks, and lunch. The lunch will include a vegetarian option.

Transportation

You can fly to Munich Airport almost anywhere in the world. However, if you are coming from somewhere in Europe within 400 km of Munich, the train is probably faster and more convenient.

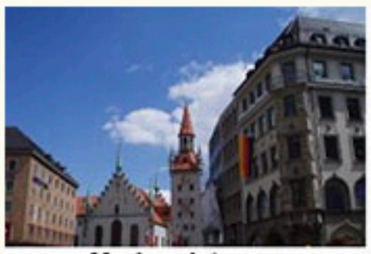
Hotels

Munich has over 400 hotels in a wide variety of price classes. Tripadvisor.com has ratings of almost 400 of them.

Dont be afraid about the prices of the hotels. The Bavarian Beerfast starts on Saturday before the MinixCon2017. The hotel that is physically closest to the campus is Hotel Königswache. You can get more affordable hotels reachable by subway (Stop Theresienstrasse, NO, not Theresienwiese!) If you plan to do some sightseeing in Munich before or after the conference, the Pinakotheken are reachable on foot and of course the Oktoberfest Another attraction are the Eisbach Surfers.

Munich Tourism

While MINIXCon 2017 is undoubtedly the biggest attraction in town, Munich has hundreds of things to do and places to see. Tripadvisor has a long list of Here are a few of the better-known ones.


Marienplatz

BMW museum

Town hall

Deutsches museum

Surfing the Eisbach wave