SEE TEXT ONLY

BY MICHAEL W LUCAS

Best Security Practices

We're talking security. Everybody likes security.

Hackers! Theft!

Big wall displays showing the dotted line where the Bad Guys routed their traffic through country after country! One team, two team, red team, blue team! It gets your heart pumping without even needing to get up out of your chair.

What nobody likes is doing all the things that prevent all the exciting hacker stuff. Applying patches and locking down all those fiddly little bits feels like wasting time. In reality, though, best practices save time. The energy you spend applying patches is trivial next to the energy you'll spend trying to scrape an intruder out of your servers. Best system administration practices, like diet and exercise, can feel flat-out tedious. Also, much like diet and exercise, few people can agree on what best practices are.

Here I present what I consider essential security practices for every host. In FreeBSD's case, figuring out what best practices are is complicated by decades of obsolete documentation. FreeBSD 12 obsoletes much of what was best practice in FreeBSD 6, and what we did with FreeBSD 1.1.2 is utterly irrelevant.

Your organization might impose additional practices. If you have a centralized logging server, an LDAP server for authentication, or an SSH certificate authority, use them. If you have an automation system such as Ansible, have it implement these practices across your environment.

Always start by listening to FreeBSD. It'll tell you most of what you need to know.

Status Mails

FreeBSD schedules automatic system checks every day, week, and month, using the periodic(8) command. The results of these commands get emailed to the system administrator. The simplest way to find out what's going on with a host is to regularly read those status mails.

The emails are sent to the root account on the local host. Many programs can email root when they go belly-up. It's best to go into /etc/aliases and redirect root's email to an address that someone actually reads, as shown here.

```
root: flunkies@mwl.io
```

Save the file and run `newaliases(8)`.

Most sysadmins I know have a long-standing goal of reducing the amount of email they receive. Yes, adding these messages impedes that goal. These emails have been collaboratively developed to contain the absolute minimum information necessary, however. You need notices such as "this package has a security vulnerability" and "your FreeBSD version just passed end-of-life and maybe you should upgrade while you can."

Additionally, you can adjust the checks these jobs perform. Maybe you have a rock-solid monitoring system that always catches when a partition or pool is about to fill, and you don't want the emails to contain filesystem utilization information. Disable and enable checks via `/etc/periodic.conf`. Like many other FreeBSD services, you'll find default settings in `/etc/defaults/periodic.conf`. By setting the task name to YES or NO in `periodic.conf`, you'll toggle that check.

Take a look through `/etc/defaults/periodic.conf` and the actual scripts under `/etc/periodic`. Some of the disabled jobs will be useful in your environment.

If you have a large staff, reading server status mails is a wonderful task to delegate to junior sysadmins. When I'm the senior sysadmin, I'll usually set the host to send all root mail to an alias on my mail server. That alias redirects all email to myself and my Trusted Lieutenant—or, rather, a lieutenant who thinks she's worthy of trust and swears she'll meticulously read every email and either resolve any issues or bring them to my attention. I can perform spot checks on those emails, watching for a host to report a problem. When my trusted lieutenant neither resolves nor reports the issue, I demote her to Lieutenant Formerly Known As Trusted and the real fun begins.

The pre-scheduled `periodic(8)` jobs don't cover everything you might need to schedule, however.

Update Checks

Once upon a time, the only way to apply security patches to FreeBSD was to build the operating system from source. Knowledgeable folks could avoid building the whole tree by building only the affected programs, but that's highly questionable with software like OpenSSL that has its grubby little fingers dang near everywhere. But the worst bit about applying security patches by hand wasn't building them; it was knowing that a patch was available and determining if the underlying problem affected your hosts.

The freebsd-update(8) program vastly simplifies security patches for the vast majority of FreeBSD users. Security patches and upgrades require only simple commands. Best of all, freebsd-update(8) tells you if a patch is available. Add a job to check for security updates to root's crontab.

```
1    1    *    *    *    freebsd-update cron
```

This tells freebsd-update(8) to wait for a random number of minutes then query FreeBSD's update servers for any new security patches for your version of FreeBSD. If it finds any new patches, it downloads them and emails root. Your Trusted Lieutenant has another chance to notice messages from your hosts.

When you notice a new set of security patches, go check out the corresponding security advisory. There's always one. See how badly this problem affects you. Do you need to apply patches over lunch, or will they wait for after hours or maintenance day? Or do you need to re-issue all your TLS security certificates because of OpenSSL… again?

Once you know how bad the problem is, and how immediately it affects you, you can apply those patches. Verify that you have a good backup. If you're using ZFS, create a new boot environment so

you can easily revert. (While I've never had a freebsd-update(8) security patch go badly, I've been a sysadmin too long to not prepare a fallback path) Now you can apply your patches.

```
# freebsd-update install
```

You'll be prompted to reboot. Depending on what gets patched, freebsd-update might tell you to reboot and run freebsd-install once more. Follow its instructions.

The freebsd-update(8) check also notifies you if your FreeBSD version has passed its End Of Life date. That makes it vitally important you upgrade to a newer version. Freebsd-update(8) makes updating to a newer release easy, but it won't do any downloads for you. It doesn't know which release you'll want to upgrade to. Are you going to run FreeBSD 11.5, or are you jumping to 12.1? That's a decision only you can make. Pick a version and use the `-r` flag to specify it.

```
# freebsd-update upgrade -r 12.1-RELEASE
```

Annoyingly, the release name must appear in the same case as the official release name. It's not 12.1-release, it's 12.1-RELEASE. You'll be prompted to compare a few critical configuration files that you've changed on your host. Decide if you want to keep or discard the changes. The program downloads the updates, and then you'll need to install them with another `freebsd-update install` command. For a full version upgrade, you'll certainly need to reboot and run the install command again.

If you're building FreeBSD from source rather than using freebsd-update, that's fine. I'm sure you have your reasons, and they might—*might*—even be valid ones. Just make sure that your environment is set up to distribute the upgrade across your network as quickly as possible. That's most often done by building on one centralized host and letting the other servers NFS mount `/usr/src` and `/usr/obj` for speedy installations.

Once you've upgraded the operating system, consider your packages.

Packages

FreeBSD's base system is deliberately small, compared to many other open source operating systems. It doesn't ship with a modern graphical environment, an SQL database, or even a web server. All those functions come from add-on packages. While those packages are easy to install and maintain, they require the same devoted, loving attention the base system needs.

FreeBSD's packaging system includes a tool to check for known security vulnerabilities in installed packages, pkg-audit(8). This tool gets run every day as part of the daily status checks, but those runs only provide the name of packages that have known security vulnerabilities. Running the tool on its own highlights the package problems. You can have your automation system run pkg-audit(8) across all of your hosts to get a master list of all vulnerable packages.

```
# pkg audit
python27-2.7.14_1 is vulnerable:
python 2.7 -- multiple vulnerabilities
CVE: CVE-2018-1061
CVE: CVE-2018-1060
CVE: CVE-2017-9233
CVE: CVE-2016-9063
CVE: CVE-2016-4472
CVE: CVE-2016-0718
CVE: CVE-2012-0876
WWW: https://vuxml.FreeBSD.org/
freebsd/8719b935-8bae-41ad-92ba-
3c826f651219.html
…
5 problem(s) in the installed packages found.
```

We have a few packages with security problems. The CVE identifiers let you look up the exact flaw, what it affects. You could use this information to determine if any of these vulnerabilities can affect your environment. The easiest fix is to see if FreeBSD has an upgraded package available with `pkg upgrade`. If you're using ZFS, create a boot environment before upgrading your packages. This will let you easily revert any changes.

```
# pkg upgrade
Updating FreeBSD repository catalogue...
Fetching meta.txz: 100%    944 B   0.9kB/s    00:01
Fetching packagesite.txz: 100%    6 MiB   6.4MB/s    00:01
…
The following 2 package(s) will be affected (of 0 checked):

Installed packages to be UPGRADED:
       perl5: 5.26.1 -> 5.26.2
       freetype2: 2.8_1 -> 2.8_2

Number of packages to be upgraded: 2

14 MiB to be downloaded.

Proceed with this action? [y/N]: y
```

The package manager will download and install the latest packages. You can then run `pkg audit` again to see which packages remain. In this case, python27 shows up again. The package upgrade remediated the installed Perl and freetype2, but didn't change python. And looking at all those CVE numbers in python27, that's a pretty long list of problems.

How can one program have so many problems?

Because nobody's fixed them, obviously.

Maybe the problem comes from the original package. Python 2.7 might have known security problems, but the python authors might have decided that eliminating those problems would unacceptably change how python behaves. In some software, the software author might dispute that a security issue needs fixing. In any case, it's better that you know the problem exists.

Perhaps the software vendor has created a fix, but the FreeBSD port isn't yet updated. Maybe the port is updated, but the new package isn't yet built and distributed to the mirrors. Maybe the fixed package is available in the bleeding edge packages, but not in the quarterly branch deployed by default.

If the problem affects you critically, look to see where the fix is held up. Open source software is community-maintained. This is your chance to contribute. If you can't fix the problem, you can at least determine how long it will be until a fix is available. If it won't ever be fixed, you can make plans to mitigate your risks.

Packet Filtering

I strongly recommend running a packet filter on all hosts. Even a simple packet filter that says "all outbound connections are allowed, but only these inbound connections" will help protect you against rogue software. Don't rely on your network firewall to block all malicious traffic; if someone weasels their way into the network, they might attack your host from another machine. While FreeBSD has three different firewall suites, my informal surveys show that 80% of FreeBSD sysadmins prefer pf(4), so we'll use that.

Start by creating a simple firewall configuration in `/etc/pf.conf`.

```
ext_if="em0"
set skip on lo
scrub in
block in
pass out
pass in on $ext_if proto tcp to ($ext_if) port {22, 53, 80, 443}
pass in on $ext_if proto udp to ($ext_if) port {53, 33433 >< 33626}
pass in on $ext_if proto icmp
```

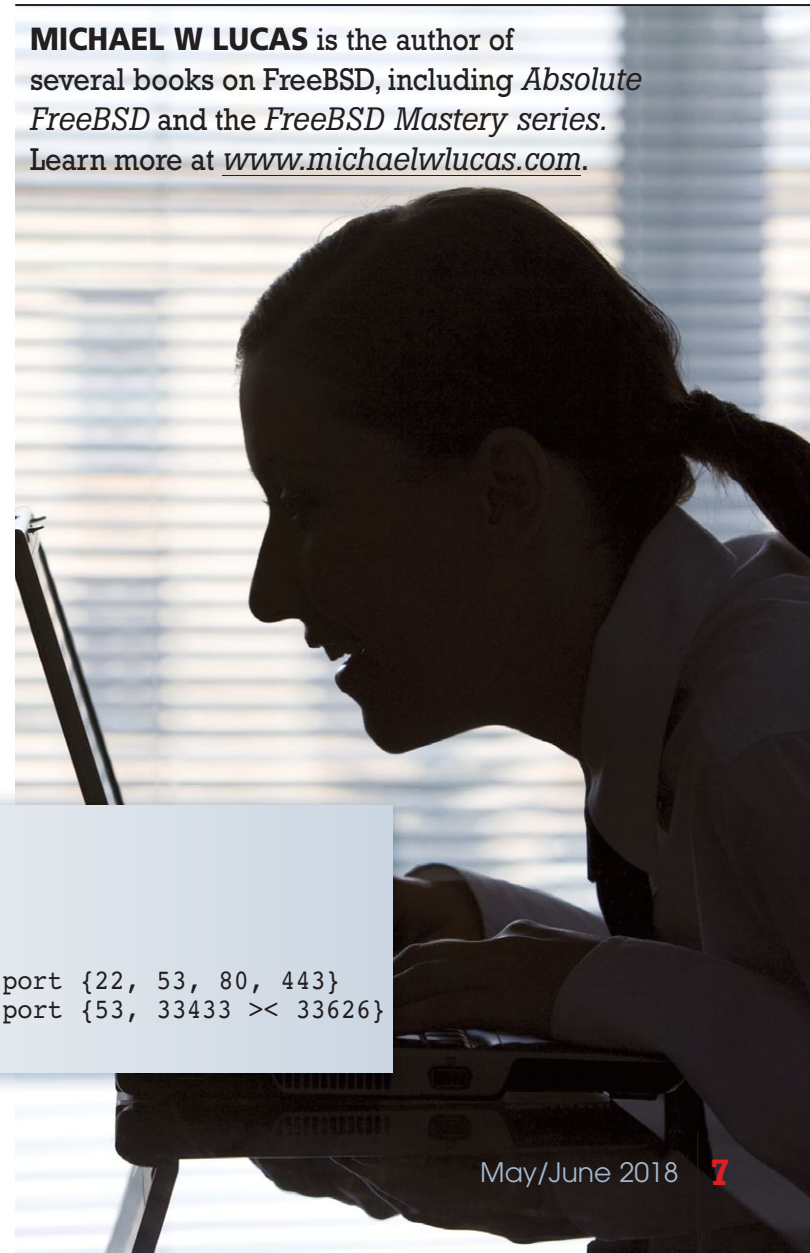
This ruleset disallows all inbound traffic but permits outgoing traffic. We then allow connections on four TCP ports: 22 (SSH), 53 (DNS), 80 (HTTP) and 443 (HTTPS). We allow a few more UDP ports: 53 for DNS, and 33433 through 33626 for traceroute. Finally, we allow inbound ICMP.

If your host doesn't run DNS, remove the TCP and UDP references to port 53. If you don't have a web server, you can remove TCP ports 80 and 443. You probably want traceroute and ping, for diagnostic purposes if nothing else.

Enable PF with the /etc/rc.conf entry `pf_enable=YES`, then start it with `service pf start.`

I could go on at length about unprivileged users and securelevels and vulnerability assessment, but you'll find extensive information about all of these in any systems administration book. If you start your FreeBSD systems here, though, you'll have a good start. ●

MICHAEL W LUCAS is the author of several books on FreeBSD, including *Absolute FreeBSD* and the *FreeBSD Mastery series.* Learn more at *www.michaelwlucas.com*.



SEE TEXT ONLY

Protect Your Secrets

Do you create complicated passwords? How hard is it for you to remember the new ones? Do you keep your credentials on a Post-it note near your monitor? If you do any or all of the above, we have an alternative that will help protect your privacy and make it easier for you to stay safe.

YubiKey Overview

A lot has been said about YubiKey, which is produced by Yubico. It has become one of the most popular solutions offering a secure 2FA — Two-Factor Authentication. It also may be used as a secondary password factor U2F — Universal 2nd Factor. It offers strong authentication and is easy to use.

YubiKey is a USB-like device. In the simplest use-case, when we connect YubiKey to a computer, it is detected as a keyboard in the operating system and the device allows us to store up to two passwords. Depending on how long we press the button on the device, it will release the first or the second password. In this simple use-case, we can use it to integrate with almost any web service.

In this scenario, we can use it to remember our passwords and forget about using Post-it notes. For more advanced users, this scenario can help protect their data. Rotating passwords is difficult, as it is problematic to remember new ones. Now, we can rotate our most often used key (for example access to our password vault) and not have to bother remembering it (although it is always a good idea to have backup). While keeping our primary password on the YubiKey device, we can still use other devices (such as a mobile phone) as a second factor.

By Jarosław Zurek,
Michał Borysiak,
and Mariusz Zaborski

Authentication Methods

Besides storing static passwords, some models of YubiKey also support more sophisticated authentication methods such as:

- **One-time Password (OTP)** – authentication mechanism, generating passwords that can be used once.
- **OATH – HOTP** – event token, generating 6- or 8-character OTP passwords using the HOTP algorithm.
- **OATH – TOTP** – 6- or 8-character OTP passwords, the TOTP algorithm is based on time function.
- **PIV (Personal Identity and Verification)-Compatible Smart Card** – enables the use of private RSA/ECC keys stored on YubiKey for signing and decryption. This mode works like a smart card.
- **OpenPGP** – encryption and signing using RSA and ECC private key, stored on YubiKey, using standard suits like PKCS #11.
- **U2F** – an open authentication standard that enables secure access to any number of online services. Only one single device, without additional drivers, or client software.

It is possible to use two-step verification with web services like Google, Facebook, GitHub, and Hotmail. This is useful because even if attackers steal the first factor of authentication (username and password to the account), they are not able to log in. We can ensure this by using a YubiKey device which supports U2F. Of course, we must enable two step verification on a chosen web service.

Models

There are a few different types of YubiKeys that can be used for various purposes. The most basic version is the *FIDO U2F Security Key*. It supports static password authentication and may be integrated with the most popular applications like Gmail and Facebook using U2F. It does not support methods like HOTP or OpenPGP. This device is the most affordable product they offer at the time of writing this article and the cost is US $18.

More advanced models, known as the YubiKey 4 generation (which also includes YubiKey 4C, YubiKey Nano and YubiKey 4C Nano) already support basic cryptographic methods like OTP or OATH modes. Differences are in device size and USB port type. Their use is definitely wider than the previously mentioned model and some examples will be explained in detail later in the article.

Another available option is YubiKey NEO. An additional feature supported by this model is communication via NFC. For example, after tapping the device, a smartphone can read OTP emitted by YubiKey. A summary of differences can be seen in Table 1.

	FIDO U2F Security Key	YubiKey 4 generation	YubiKey NEO
OTP		✔	✔
OATH – HOTP		✔	✔
OATH – TOTP		✔*	✔
OpenPGP		✔	✔
U2F	✔	✔	✔
Secure Element	✔	✔	✔
Smart Card (PIV)		✔	✔
Supports NFC communication			✔

**Requires additional app (lack of built-in Real Time Clock).*

Table 1: Comparisons of different YubiKey models

FreeBSD Tooling

YubiKeys come with a set of open source tools that are necessary to integrate YubiKey in a Unix-like environment. For FreeBSD users most of these tools are available in the ports collection as well as in the binary package repository. Two of the most interesting packages are `security/ykpers` and `security/yubico-pam`. The `security/ykpers` package contains several command line tools to manage YubiKey:

- `ykpersonalize(1)` —is necessary to program YubiKey; it handles configuration options for almost every YubiKey feature.
- `ykinfo(1)` —is useful for retrieval of basic information about YubiKey.
- `ykchalresp(1)` —allows for signing data using YubiKey in a challenge-response mode of operation.

GELI Full Disk Encryption

GELI is the most popular disk encryption method on FreeBSD. One of the most secure ways of using GELI and FreeBSD is to use two-factor authentication with a passphrase and a key file. The key file is kept on a memstick. When we boot our machine, we need to provide both factors. After decrypting our device, we

remove the memstick with the file. In that way, if somebody steals our computer they will also need to see our password and steal our memstick. If we want to be even more paranoid, we can keep a kernel and a key file on the memstick. In that way, even if somebody has access to our computer, it makes it impossible to integrate with our software. An intruder could still alter our hardware, but this is much harder.

FreeBSD has recently begun supporting full disk encryption, which means that even a kernel is encrypted—only the small boot loader partition is not. Unfortunately, in the current implementation, we do not support a key file, so we can only use a passphrase to encrypt our disk. Thanks to YubiKey, which can be detected as a simple keyboard, we can use it to provide a passphrase during boot. Using it in that way, we lose one factor. We can mitigate this by using a passphrase which we provide using a normal keyboard and a second part of the password which is much longer and is kept on YubiKey. In that scenario, somebody not only would need to see what passphrase we are typing, but also would need to steal our YubiKey.

To make the device to meet these requirements, we program the second slot of Yubico in static mode:

```
$ ykpersonalize -2 -o static-flag -o append-cr
```

Thanks to the `append-cr` flag we do not have to press Enter after each use of this slot. In the case of GELI, we would first provide our passphrase and then the second factor using YubiKey. By default, there is no output when typing a password in GELI. In our case, we would see that the passphrase was submitted because the factor provided by YubiKey would contain trailing ENTER.

Integration with FreeBSD Login

Yubico provides a PAM module which can be deployed within existing authentication systems. The module can be found in the `security/pam_yubico` package and it works in two modes: online and offline authentication. The former provides a second factor based on one-time passwords, but it delegates authentication to Yubico cloud services. Therefore, it requires a stable internet connection. On the other hand, it is very easy to setup a newly bought Yubico as a second authentication factor for your system account.

Every device has a secret on its first slot preset by the manufacturer. The slot works in so-called Yubico OTP mode. Each password generated from this slot consists of a static 12-character part and the remaining dynamic part. The static part never changes and it can be considered the device's public identifier. These factory defaults are already known by Yubico cloud services. All that we need to do is to retrieve the API token and the user ID using a special form available on Yubico's website.

The second mode of operation is more practical. It requires a YubiKey to work in a challenge-response mode when the device can be issued to sign data sent by a user application. In this particular example, our application is the Yubico PAM module.

First, we have to program the device to work in a challenge-response mode. In the example below, we will use the first slot:

```
$ ykpersonalize -1 -o chal-resp -o chal-btn-trig -o chal-hmac -o hmac-lt64 -o serial-api-visible
```

We want the device to wait for a user confirmation of each operation which is guaranteed by the `chal-btn-trig` flag. The user has to press the key located on the YubiKey while logging into the system, unfortunately twice. The first press is needed to check whether the challenge located in a file matches the device response. If so, the second press generates a new challenge-response pair and stores it for later use. We need to generate an initial challenge which is stored by default in `~/.yubico` directory:

```
$ ykpamcfg -1 -v
```

The next step is to configure the PAM module. We want to use the second factor together with a static password to protect any login to our computer. For this purpose, we will modify the `/etc/pam.d/system` file so the "auth" section will look as follows:

```
# auth
auth    required    pam_unix.so        no_warn try_first_pass
auth    required    /usr/local/lib/security/pam_yubico.so mode=challenge-response
```

Now issue the `sudo -s` command, type the static password and press the button on the device. It waits for a user action up to 15 seconds and an LED indicator is blinking during this time. Authentication will fail either when the user does not take action or if YubiKey is not connected to the USB port.

Integration with SSH

Another useful application of YubiKey is strengthened remote authentication to an SSH service. A commonly described method utilizes the Yubico PAM module using the online mode of operation. However, a Yubico token is compliant with OATH-HOTP standards and, therefore, it can work with any authentication server which supports this standard. For example, Wheel Cerb AS is such a multi-factor user authentication solution. We will use the `pam_oath` module which is included in the `security/oath-toolkit` package and does not require a connection to any external cloud service.

We need to reprogram our YubiKey to support OATH mode. Unfortunately, we have only two slots on our device so we need to overwrite one of the previous configurations:

```
$ ykpersonalize -1 -o oath-hotp -o oath-hotp8 -o append-cr
```

We want to use 8-digit long, counter-based passwords. The output of the command should contain a line with a key in hexadecimal form:

```
      ...
key: h:c621245c5f05eefec1d9f2960f34b865849dd074
  ...
```

We need to store the user's name and the key in the `pam_oath` database file on the target machine (of course "alice" and the key should be replaced by real values):

```
$ echo "HOTP alice - c621245c5f05eefec1d9f2960f34b865849dd074" >> /usr/
local/etc/users.oath
```

The next step is to modify the `sshd` PAM configuration in order to enable the `pam_oath` module. We can achieve this by modifying the `/etc/pam.d/sshd` file so the "auth" section will look as follows:

```
# auth
…
auth        required      pam_unix.so          no_warn    try_first_pass
auth        required      /usr/local/lib/security/pam_oath.so usersfile=/
                          usr/local/etc/users.oath window=16 digits=8
```

We need to ensure the `sshd` configuration which is placed in the `/etc/ssh/sshd_config` file contains a few options set to the appropriate values as follows:

```
ChallengeResponseAuthentication yes
PasswordAuthentication no
UsePAM yes
```

Finally, reload `sshd(8)` service and try to login to the remote server typing the static password and then using the dynamic password from the token.

Conclusion

YubiKey is a remarkable device that can be used in a corporation or by individuals to increase their security. This small device supports many different authentication methods and can be used with many popular web services as well as with programs like SSH. It also allows us to leverage some of the imperfections of tools that do not support a 2FA. It can be used as a second factor or to keep the primary password. It is an interesting alternative to other solutions like mobile applications. YubiKey also allows us to painlessly change our passwords without the need for any memorization. ●

JAROSLAW ZUREK is a software developer at *Wheel Systems* where he supports a project creating a privileged session manager. He is interested in cryptography, TLS, and low-level/hardware programming.

MICHAL BORYSIAK is a software developer at *Wheel Systems,* where he works on centralized authentication systems. He is fascinated by low-level operating system concepts, networks and cybersecurity.

MARIUSZ ZABORSKI is a lead software developer at *Wheel Systems*. He has been a proud owner of the FreeBSD commit bit since 2015. Mariusz's main areas of interest are OS security and low-level programming. At *Wheel Systems*, Mariusz leads a team that is developing the most advanced solution to monitor, record and control traffic in an IT infrastructure. In his free time, he enjoys blogging (http://oshogbo.vexillium.org).