Ethereal Wake

FreeBSD Encrypted ZFS Root on EFI

2020 Dec 25 | Permalink | 1751 words, 15 min read | Tags: Boot, ELI, FreeBSD, Sysadmin, UEFI, ZFS

On my home server, I use FreeBSD. Sadly, the BSD's have been falling behind Linux in the past decade but they still appeal to many people and even have a few tricks left in them. Most notably, as Linux has been struggling with next generation file systems for a few years now, FreeBSD has integrated ZFS *for over a decade*.

While modern Linux distros become ever-more complex interplays of components, the BSD's have remained relatively simple. Custom installation (by hand or script) is trivial when the system is distributed as a tarball or two versus a constellation of packages numbering in the thousands. Just boot into the live CD, format the file systems, unpack the tarballs, install the boot loader, and reboot into your new system.

A custom installation allows you to fulfill your OCD tendencies, but it requires you to have a strong understanding of how the system is constructed. Unfortunately, FreeBSD's documentation has not been keeping up with the evolution with the system's capabilities. For example, the page on `efi(8)` is extremely short and does not address modern concerns like ZFS and full disk encryption. ~~In fact, much of the limited information on that page is downright misleading.~~ [**NOTE:** This has been fixed as of FreeBSD 13.]

> **NOTE:** These instructions were written at the time of FreeBSD 12.2. They are still applicable as of FreeBSD 13.0, but recent updates to the documentation have fixed many of the complaints.

A traditional boot process comes in three phases:

1. The system firmware loads the *first stage boot loader*.
2. The *first stage boot loader* loads the *second stage boot loader*.
3. The *second stage boot loader* loads the kernel.

The use of two separate boot loaders seems unnecessarily complicated from a naive perspective, but it serves an essential purpose on many systems. The firmware's boot capabilities are limited so it may not be possible to fit all the functionality required to load a kernel into that space. As the most obvious example, the classic PC BIOS would only load 512 *bytes* into memory, clearly insufficient for reading a modern file system or configuring the CPU for protected mode. Many systems ended up with *three* boot loaders just to work around this limitation.

~~Those EFI instructions are clearly written with this model in mind.~~ However, EFI is not a limited system. In fact, it's practically a full-fledged operating system, capable of loading drivers and providing high-level services to the operating system or its boot loader. This means that not only is the first stage boot loader not required, the actual boot loader can be made much simpler since many of the things it's traditionally been required to do are now provided by the firmware.

Going back to those instructions, the fundamental problem is that the EFI first stage boot loader does not support ZFS root with GELI (FreeBSD's full disk encryption module). Following those instructions would produce a system incapable of supporting such an arrangement, despite it being supported by the legacy `gptzfsboot(8)` first stage loader. But the work around is easy: EFI is perfectly capable of loading the second stage loader directly and the second stage loader supports both of these features.

While FreeBSD 13 is going to shake things up with the transition to OpenZFS 2.0, that's still a few months away. In the meantime, I might as well share my notes as to how to get this system working. To keep things simple, I've tried to eliminate many of the more complicated organization schemes to focus on the core details.

Partitions

The first thing we need to do is partition the disk using the `gpart(8)` command. As we are discussing the use of EFI, that usually means the GPT partition scheme.

```
gpart create -s gpt ${disk}
```

With that done, we can create the EFI system partition and any other partitions we need on the disk. Since most modern disks have 4k native sectors, that should be your minimum alignment. Larger alignments won't hurt anything beyond loosing an inconsequential amount of space in order to create the alignment. If it eases your conscience, Microsoft uses megabyte alignments by default.

For later convenience, be sure to assign each partition a label which will be stored in the partition table. These labels are system-wide, so make sure the labels on each disk are unique (e.g. append the disk number).

```
gpart add -t efi -a 4k -s 512M -l efi0 ${disk}
gpart add -t freebsd-zfs -a 4k -l zdata0 ${disk}
```

With the partitions created, we need to prepare the file systems.

ZFS Pool

Since we're going to need some files from it anyway, let's start with the ZFS pool.

First, we need to use the `geli(8)` command on each partition to initialize the full disk encryption:

```
geli init -g -s 4k gpt/zdata0
geli init -g -s 4k gpt/zdata1
```

And once initialized, we need to attach the newly initialized GELI devices:

```
geli attach gpt/zdata0
geli attach gpt/zdata1
```

Finally, we can create our ZFS pool on the encrypted devices using zpool(8):

```
zpool create zdata mirror gpt/zdata0.eli gpt/zdata1.eli
```

If you're working from the live CD, you'll probably need to add something like `-o altroot=/tmp` since the system won't be able to create the mount point in the (read-only) root directory using.

> **Note:** Notice the lack of worry regarding `ashift`. By initializing our GELI partitions to the native sector size of the disk, it's not even possible for ZFS to choose anything smaller.

FreeBSD Installation

With the pool created and imported, we can now create our root directory for FreeBSD. By tradition, these are direct children of `${zpool}/ROOT`. So let's create our first dataset using `zfs(8)`.

```
zfs create -p zdata/ROOT/freebsd
```

Installing the system is just a matter of unpacking the distribution tarballs. You can download these from the FreeBSD website. At a minimum we're going to need `base.txz` and `kernel.txz`, but you may find `lib32.txz` and `src.txz` to be useful. Before you install everything, ports are better sourced through `portsnap(8)` while the rest are mainly for system development.

```
tar xfC base.txz ${altroot}/zdata/ROOT/freebsd
tar xfC kernel.txz ${altroot}/zdata/ROOT/freebsd
```

> **Note:** For those not familiar with the BSD `tar(1)`, it can automatically determine the compression so there's no need to remember `z`, `J`, or the other silly options when

> extracting a tarball. In fact, the utility completely ignores those options when reading an archive.

I typically use this installation as template from which I can *clone* not only the base system, but any jails I plan to operate.

```
zfs snap zdata/ROOT/freebsd@install
zfs clone zdata/ROOT/freebsd@install zdata/ROOT/system
```

Finally, we need to tell ZFS which dataset we want to boot from.

```
zpool set bootfs=zdata/ROOT/system zdata
```

As a security measure, I like to reduce permissions on `ROOT` so users can't poison systems that aren't loaded.

```
chmod 700 ${altroot}/zdata/ROOT
```

EFI System Partition

Now, we have everything we need to prepare our EFI system partition. First, we need to format the system with FAT using `newfs_msdos(8)`. Typically, we use FAT32 for this this, but some systems may be able to handle other FAT variants.

```
newfs_msdos -A -F 32 -b 4096 -L EFI /dev/gpt/efi
```

Now, we simply need to copy the second stage boot loader from our FreeBSD installation into the appropriate location.

```
mount -t msdos /dev/gpt/efi /mnt
mkdir -p /mnt/EFI/BOOT
cp ${altroot}/zdata/ROOT/freebsd/boot/loader.efi /mnt/EFI/BOOT/BOOTX64.EFI
umount /mnt
```

As described in the introduction, we are copying `loader.efi` and not `boot1.efi` ~~as described in `uefi(8)`. This is intentional~~.

Preparing for First Boot

At this point, we can start installing whatever we want to install into our system. But there are some things we need to take care of.

By default, the kernel will not have support for GELI or ZFS. Since these are part of the root file system, we need to make sure the bootloader loads these modules or the kernel will panic. To do this, we need to edit /boot/loader.conf:

```
aesni_load="YES"
geom_eli_load="YES"
zfs_load="YES"
```

NOTE: `aesni(4)` is included as part of `GENERIC` as of FreeBSD 13 and does not need to be included in `loader.conf`. In previous versions, this was required to enable hardware acceleration of cryptographic operations.

We will probably need to have a minimal /etc/rc.conf (adjust appropriately):

```
hostname="something"
# Update to reflect whatever your actual hardware
ifconfig_igb0="DHCP"
# If you're not planning on using sendmail
sendmail_enable="NONE"
# Logs certain GPT-related events on boot
gptboot_enable="YES"
# ZFS fault management daemon
zfsd_enable="YES"
```

And we might want a little more than the default `devfs` and our ZFS pool, so create an initial /etc/fstab:

```
# Device        Mountpoint          FStype    Options         Dump    Pass#
/dev/gpt/efi0   /boot/efi           msdosfs   rw              0       0
fdescfs         /dev/fd             fdescfs   rw              0       0
procfs          /proc               procfs    rw              0       0
```

Be sure to create directories for any mount points you list in `fstab`:

```
mkdir ${altroot}/zdata/ROOT/system/boot/efi
```

Done

With that, we can now export the ZFS pool and boot into our new system.

```
zpool export zdata
shutdown -r now
```

Inside our new system, we can run some basic checks:

```
root@test:~ # geli status
     Name  Status  Components
da0p2.eli  ACTIVE  da0p2
da1p2.eli  ACTIVE  da1p2

root@test:~ # zpool status
  pool: zdata
 state: ONLINE
  scan: none requested
config:

        NAME             STATE     READ WRITE CKSUM
        zdata            ONLINE       0     0     0
          mirror-0       ONLINE       0     0     0
            da0p2.eli    ONLINE       0     0     0
            da1p2.eli    ONLINE       0     0     0

root@test:~ # bectl list
BE      Active Mountpoint Space Created
freebsd -      -          1.11G 2020-12-26 00:10
system  NR     /          980K  2020-12-26 00:14

root@test:~ # mount
zdata/ROOT/system on / (zfs, local, nfsv4acls)
devfs on /dev (devfs, local, multilabel)
/dev/gpt/efi0 on /boot/efi (msdosfs, local)
fdescfs on /dev/fd (fdescfs)
procfs on /proc (procfs, local)
```

From this point, make the system yours:

- Set the root password and create additional users.
- Override the default timezone.
- Install interesting packages.
- Register it with Ansible, Puppet, or whatever system management tool you are using.

Variant: EFI Mirror

If you are using multiple disks, you will probably partition each disk the same way. This means each of your disks would have space reserved for the EFI system partition. If you're doing that, you might as well set up a mirror so they all contain the same data and EFI can fall back to any of them in the event of a disk failure.

We set up a mirror in a similar manner as we did with GELI, but instead use `gmirror(8)`:

```
gmirror label efi gpt/efi0 gpt/efi1
```

And like GELI, we need to make sure the kernel loads the appropriate driver, so update `/boot/loader.conf` to read:

```
aesni_load="YES"
geom_eli_load="YES"
geom_mirror_load="YES"
zfs_load="YES"
```

Finally, revise the earlier instructions to use `mirror/efi` instead of `gpt/efi0` when dealing with the file system.

```
root@test:~ # gmirror status
      Name    Status  Components
mirror/efi  COMPLETE  da0p1 (ACTIVE)
                      da0p1 (ACTIVE)
```