# Managing System Images with ZFS

Allan Jude, Klara Inc
allan@klarasystems.com

## Abstract

The author describes existing procedures, tools, and ongoing development to improve the process of updating appliances, remote systems, and individual computers using ZFS. This paper describes a mechanism for replacing the operating system image with a newer image in a safe and atomic fashion. This system allows for fail-safe unattended upgrades of remote appliances and machines with a built in automatic recovery mechanism in the event of failure. Current and planned enhancements to 'poudriere image' are described as well as improvements to support tools including bectl and zfsbootcfg.

## 1. Motivation

FreeBSD is a popular choice for building appliances because of its liberal license and composable base system, but each vendor is tasked with building their own upgrading mechanism for both minor security updates, and major OS upgrades. FreeBSD would benefit from a standardized upgrading mechanism that is adaptable to each vendors requirements. We examined the existing updating frameworks in FreeBSD and then set out to build a better mousetrap. Leveraging experience gained while managing 100s of remote servers we describe how upgrades can be handled in an automatic, yet safe fashion, without merging configuration files or requiring manual intervention.

## 2. Prior Art

The problems of building system images and keeping systems up to date are not new. FreeBSD has evolved a number of systems over time to try to address these issues, and many of them worked quite well for a time. However, each has developed shortcomings as the state-of-the-art has advanced.

### 2.1. NanoBSD

NanoBSD is a build system first introduced to FreeBSD in 2004 by Poul-Henning Kamp[1] for the purpose of making it easier to generate compact-flash disk images for embedded systems running FreeBSD. The resulting image would consist of three partitions, the 3rd held persistent configuration data, and was usually quite small, and the remaining space was divided evenly to form two system image partitions. The current operating system would be written to both of these partitions, and in the future when it came time to upgrade, the inactive partition would be overwritten with a newer image, and a flag set to boot from that partition, one time only. If that boot succeeded, then the upgraded partition would become the new default. If boot failed, power cycling the device would automatically revert to booting from the image that had been running before the attempted upgrade. Upgrades could continue like this, alternating back and forth between the two partitions, always leaving the system with a known good image to fall back to. The filesystems were also mounted read-only, so power loss or other issues could not corrupt the filesystem or require a fsck(8) at boot. Configuration data was copied from the dedicated configuration partition to a memory-backed filesystem at boot, and optionally saved off to the dedicated partition when necessary.

NanoBSD allowed easy customization of FreeBSD, including and excluding features as needed to make a functional but minimum system image. NanoBSD was adopted by many FreeBSD based appliances, including pfSense and FreeNAS.

NanoBSD is limited to UFS, and both FreeNAS and pfSense have since switched to ZFS boot environment based solutions instead.

### 2.2. freebsd-update

Binary updates for FreeBSD were first introduced in 2001 as experimental accompaniments to some security advisories in the summer of 2001 by Colin Percival[2]. In this timeframe, the freebsd-update client was available via the ports tree.

Version 2.0 of freebsd-update was merged into the FreeBSD base system in 2006, and the building of the updates was handed off to the FreeBSD Security Officer, rather than Colin personally building the updates.

freebsd-update binary updates are created by comparing the build output of the unmodified source code (build 1), to the same source code built with the clock changed backwards by 400 days (build 2), to detect changes in files that are related to “build stamps”, rather than changes to the code. The offsets that contain “build stamps” are identified and recorded. Then a 3rd build is done of the patched source code, and it is compared to the original release binaries, excluding the offsets of the “build stamps”. Any files that still differ have been modified by the patches to the source code, and need to be distributed as part of the binary update. Lastly, a 4th build is done, again on the patched source code, to locate the new offsets of the "build stamps", as changes to the source code are likely to have moved the build stamps. In some cases, the first of the four builds can be replaced with downloading the original -RELEASE ISO instead.

PC-BSD previously attempted to use freebsd-update to provide updates to its users, but found the freebsd-update-server to be too slow (requiring 3 or 4 full buildworlds) and fragile. They also found that freebsd-update did not support updating stable branches or head, only releases. Users also complained about the way merging of configuration files was handled (manually, labour intensive), especially the fact that freebsd-update does not ignore changes to VCS ID lines, and the chance of leaving merge markers in configuration errors by mistake is too high.

Currently freebsd-update is x86 only (i386 and amd64) and still requires a lot of manual intervention by secteam when trying to release security advisories. Support for other architectures is likely quite feasible, but the read/write access pattern of the freebsd-update client is not very conducive to SD cards and other low speed, low endurance flash.

Recent work on reproducible builds should reduce the number of "build stamps" that exist in the official releases of FreeBSD, but this is unlikely to be able to eliminate an entire build cycle due to the way freebsd-update is structured.

### 2.3. pkg base

The pkg(8) package manager has been used, since 2014, to install, manage, and update third party software, and is one of the most familiar aspects of the FreeBSD Operating System for end users. Naturally, this argues for using pkg to distribute and update the base system itself. There have been discussions and some development along these lines, but a complete solution has not yet emerged.

Currently development seems stalled and there is a lack of an overall design for what a packaged base system will look like. Trying to balance flexibility, and allowing a user to opt out of individual pieces of the base system has resulted in the system being split into many 100s of packages. This will make updates smaller, but makes dependency tracking more difficult and results in a very verbose package listing. However, pkg(8) has better automated 3-way merge handling for configuration files. Conflicts are left for the user to deal with later, leaving the original file in place, and installing the new version with the suffix ".pkgnew".

PC-BSD/TrueOS attempted to use the pkg-base system as it existed, but found a number of limitations, especially around upgrades. TrueOS is currently starting fresh with trying to have the base packages built via the ports tree, rather than the main OS build infrastructure, with a target of approximately 10 packages, rather than the current 800 or so packages.

## 3. ZFS

ZFS is an advanced filesystem that combines the role of volume manager and filesystem together to make managing storage easier for the administrator. ZFS is a CoW (copy-on-write) filesystem, so blocks are not overwritten in place, but written to a new location, and then the old location becomes free again later. ZFS is transactional like a database, so each group of writes (transaction) either fully completes, or is rolled back. This allows ZFS to move from consistent state to consistent state, without ever requiring intervention from tools like fsck. So, if the system crashes or unexpectedly loses power, the filesystem does not require any recovery steps or consistency checks. It finds the more recently completed transactions and mounts the filesystem from that point. This obviates the need to have the filesystem be read-only, as it is in NanoBSD, to avoid being inconsistent.

A traditional filesystem can only operate upon a single disk, so volume managers were created that would allow multiple disks to be combined into a single logical disk that could be presented to the filesystem. The volume managers also grew features like parity and redundancy (RAID), to protect the filesystem from the failure of one of the disks that makes up the volume. By combining these roles into a single system, the ZFS filesystem has direct knowledge of the fact that the filesystem is backed by multiple disks, allowing for dynamic stripe sizes and other optimizations.

The biggest advantage to this approach, is that ZFS filesystems each share all of the free space from the "pool" of available storage. Whereas in a traditional filesystem, the volume must be partitioned into fixed-size chunks at the time of filesystem creation. While most filesystems support growing, this requires there be contiguous free space available at the end of the existing filesystem. In addition to the inflexibility of this arrangement, it can lead to the available free space being fragmented across multiple partitions, where the total amount of free space is sufficient for an upcoming task, but no individual partition has enough free space to meet the demand. With ZFS, as files are written to one of the filesystems, the free space is taken from the pool and allocated to that filesystem. Each filesystem does not have a fixed size, but instead can take available space from the pool as needed, in a thin-provisioned manor. It is also possible to reserve space for a specific filesystem, or limit a filesystem with a quota.

## 4. ZFS Boot Environments

With ZFS, creating additional filesystems a low cost operation. This allows the NanoBSD concept to be taken much further. Instead of two fixed sized partitions for system images, ZFS allows a number of system images limited only by the available space in the pool. Additionally, the copy-on-write nature of ZFS can used to share unchanged blocks between system images to save space.

The concept of ZFS boot environments, originally from Solaris (as is ZFS), is having multiple root filesystems that the operator can choose from at boot time, using the loader menu. A snapshot of the working system can be taken at any time, and then cloned to create a writable filesystem, with the contents of the system image as it existed at the time of the snapshot. This allows the operator to easily revert to a previously working image, without the storage cost of an entire system image. Additionally, instead a single dedicated configuration partition, every filesystem other than the root filesystem is retained as the boot environment is changed. User data, such as home directories, log files, configuration, databases, etc are all retained as long as they live in a filesystem other than the root. The system image can be build such that packages (/usr/local) are included in the system image (so a failed package upgrade can be reverted), or be kept in a separate filesystem, so they are not impacted by changes to the system image.

To provide an equivalent to the fail-safe upgrade mechanism that NanoBSD has using GPT partition flags, FreeBSD has the zfsbootcfg utility. This writes the name of the selected boot environment into the on-disk filesystem label, where boot1 (gptzfsboot) reads it, and then overwrites it with zero bytes. It then sets it as an environment variable when invoking the loader. The loader then uses it to load the kernel from that filesystem, and pass that filesystem as the root to the kernel when it boots. This provides the same “next boot, and next boot only” use this newly upgraded filesystem instead of the default. If the boot is successful, the default can be changed and the upgraded system image will now boot by default. If not, power cycling the device will revert to the previous system image, and the upgrade can be retried, or debugged.

ZFS features a replication protocol, that can serialize a filesystem or hierarchy of filesystems into a stream that can be stored for later recreation of the filesystem. This protocol also includes support for incremental replication between snapshots of a filesystem or series of filesystems. Using this system it is possible to incrementally update a clone of a system image, reducing the size of the binary updates. In 2018 the replication protocol was enhanced to take advantage of ZFS's transparent compression features. If data is compressed on-disk in the filesystem, it can optionally remain compressed during replication, further reducing the size of the incremental system image updates.

## 5. Building System Images

How does one build a clean system image using ZFS? The FreeBSD package building framework, poudriere, includes a system image building feature.

### 5.1 Poudriere

In the previous generation of packaging on FreeBSD, before the pkg tool, packages were built in a distributed fashion, using a number of machines, but it was often fragile, and duplicated a lot of effort. As work was distributed, it often came to pass that the same common dependencies were build by each worker, and the effort is coordinating the distributed system, and maintaining the control software was to onerous.

Thus, poudriere (powderkeg in french) was invented. Rather than trying to build one package at a time, as quickly as possible, using all of the CPUs, it instead creates a number of jails (by default equal to the number of CPU cores), and in each one builds a single package using only a single thread. This tends to make much more efficent use of the available CPU cores, since parts of the build that are inherently single threaded (configure scripts and the like) do not result in all other CPUs going idle. An additional advantage to using a jail for each builder, is that the jail can be reverted to a clean environment between each build, ensuring there is no contamination of the build environment. Before building each port, a clean copy of the operating system is setup, the binary packages of the dependencies are installed, and then the build is run. The build jail also has no access to the internet, ensuring that the build process cannot reach out to the internet and modify its behaviour. While not required, using ZFS for these build jails makes the cleanup process nearly instantaneous, as the filesystem is just reverted to a clean snapshot between each build.

The system image used for these build environments can be created by downloading the official releases of FreeBSD (no compiling required), or by building from source, optionally with an external patch applied before the build process. Poudriere supports both svn and git, as well as tar and copying an existing source tree. The former options include support for incremental updates (svn update, git pull). Existing system images can also be updated with freebsd-update. Poudriere can also cross-build for different architectures, making it possible to produce system images and packages for arm, arm64, risc-v, etc, from a standard amd64 machine.

These basic system images must be created in order to build packages, so when the need arose for Gandi.net to build virtual machine images of FreeBSD for their public cloud, it was a logical extension to poudriere, and the “image” subcommand was born. The image command takes an existing poudriere jail (which can be compiled, or created from an official release and binary updates), excludes listed files you do not want or need, adds an overlay (your own custom files that are added on top of the system image), installs a list of packages, and then generates an image in the specified format. Supported formats include ISO (cd/dvd image), optionally using an memory filesystem which itself is optionally compressed, or the same for a USB stick image, a raw disk image which is suitable to be written directly to a disk or converted to various hypervisor image formats, an embedded or firmware image, a UFS dump, or a ZFS replication stream. The ZFS replication stream comes in two flavours, the entire pool (a compound stream containing all filesystems), or a single stream containing just the root filesystem (the boot environment).

The author added the support for ZFS replication as an output format for poudriere image, and is continuing to work on enhancements. Currently building incremental replication streams is still a work-in-progress. The existing ZFS support defaults to a pool layout identical to that created by bsdinstall, but is easily customizable. This controls what filesystems are created for the whole pool image, or what files are included in the root filesystem versus having their own filesystem in the boot environment mode. This can be used to control if packages are part of the system image, or if they are managed independently.

Distribution of a boot environment system image is just a matter of feeding the contents of the image file to the `zfs recv poolname/ROOT/environment_name` command. For a full pool image, create a pool if one does not exist, and then `zfs recv -F poolname`. Beware, this will irrevocably overwrite the pool with the contents of the new system image.

### 5.2 Customizing Images

In order to achieve our requirement to retain system configuration through upgrades, while avoiding the need to merge every change to /etc, we must make some small customizations to the system image as part of its creation. We create an additional filesystem, /cfg, to store the subset of files from /etc we wish to have been persistent. We then replace the versions in /etc with a symbolic link to /cfg. Our configuration moves the following files to /cfg: fstab (for swap configuration, ZFS does not use fstab by default), hostid (so the hostid persists), rc.conf.d (directory), rc.conf.local, resolv.conf, ssh (directory), sysctl.conf.local.

The remaining files in /etc are replaced with the latest version during each system image upgrade. Of course, since these files now reside on a separate filesystem, they will not be available during the boot process, until the additional filesystems are mounted. This is doubly true since fstab is one of the files we have relocated, so the OS won't have a list of other filesystems to mount. To address this, we abuse a little known feature of FreeBSD's init process. Designed to allow booting into a chroot environment, the loader.conf variable init_script runs a script to prepare the chroot, and init_chroot sets the directory to chroot the system into. We use the init_script feature to run a small script that finds the poolname/cfg filesystem and mounts it early in the boot process, before /etc/rc is run, so that our replacement configuration files will be readable.

## 6. Upgrading Systems

The process of upgrading a system is straightforward. First build the new system image, as a ZFS boot environment, then receive it under a new name to the pool on the target device. Use the zfsbootcfg utility to configure the system to boot into the new environment only once. Configuration such as hostname, network settings, SSH keys, will be retained via the /cfg filesystem. Attempt a reboot. As the system boots, it will erase the zfsbootcfg temporary configuration. If there are no problems, the system will now be running from the new environment. It is left as an exercise to the reader to develop a procedure to determine that the new system image is working as expected, and set the upgraded system image as the boot default. If this is not done, or the system fails to boot correctly, rebooting will use the previous system image.

Whether packages are included in the boot environment will depend on the type of deployment. In an appliance type configuration, it makes sense to bundle the packages into the system image, so updating the system image updates the packages as well. This also avoids any potential dependancy solving issues, as the new system image always contains a freshly installed set of packages. In the case of a server deployment, or an appliance where the set of packages may be customized, it may make sense to have /usr/local as its own filesystem and managing packages separately from the system image. Reducing what is included in the system image makes the incremental updates smaller and less risky. Deploying OS security updates does not need to involve changing the versions of the packages that are installed on the system. Excluding the packages from the system image allows the two to be updated separately. They could still both be managed using the same ZFS update mechanism, just as separate filesystems.

In the case of a laptop or workstation, it makes sense to include the packages in the boot environment, so that a package upgrade can be undone if it causes issues. The advantage to boot environment is that rolling back to an older system image won't roll back the user data, so no work is lost even if rolling back to a weeks old system image.

## 7. Further Enhancements

There is still much to be done to improve this mechanism and provide a flexible but fail-safe upgrading mechanism. Creating a upgrading mechanism that is adaptable enough to serve the majority of use cases is challenging, but the result will ultimately be beneficial to the entire FreeBSD community.

### 7.1 Poudriere Image

Currently the layout of ZFS datasets in poudriere image is very basic, and defaults to the same configuration used by the FreeBSD installer. The author envisions a series of different templates for different use cases, and a more expressive configuration syntax that requires less ZFS expertise than the current system.

In addition to further enhancements to the overlay system to make it easier and more performant to include additional material in the system image, it is desirable to have support for a post-built chroot script to perform operations on the image. This is complicated in the cross-platform image case.

Poudriere image has a relatively wide selection of output formats, but the level of flexibility is still rather limited. Increasing the options available to the user without creating a combinatorial testing nightmare is a delicate balancing act. Even just the selection of disk layout and bootcode combinations: MBR, BSD, or GPT partition table, Legacy BIOS, UEFI, or dual mode, regular or advanced bootstrap code, encrypted with GELI or not, UFS (one or many partitions) or ZFS, etc quickly leads to over a 100 possible combinations to test, let along try to support. Then each of those can be packed as a raw disk image, CD/DVD ISO, vmdk, vhd, qcow2, Amazon EC2 image, and more. Finding what is more useful and most supportable and trying to limit creep will be challenging.

### 7.2 Bootstrap and Boot Code

Updating the bootstrap, the tiny bits of assembly that load the more complicated and feature rich loader, can be complicated and error prone. Worst of all, there is currently no fallback mechanism if this goes wrong. In the legacy (BIOS) boot case, the bootstrap is a complicated process (see "Booting from Encrypted Disks on FreeBSD"[3], proceedings of AsiaBSDCon 2016) consisting of multiple phases. While the boot0 and boot1 rarely change, boot1 is usually combined with boot2, so is updated when boot2 changes. With ZFS, changes to boot2 are often required when new features are introduced, as boot2 contains a minimal, read only implementation of ZFS that is used to read the boot loader from the ZFS root filesystem. This is also where the zfsbootcfg logic happens. There is currently no provision for recovering from a failed boot2. If the newly installed version doesn't boot, the zfsbootcfg logic may never happen, and its effects are only on the later stages of the boot anyway. In the common GPT case, boot1 and boot2 are combined and live in the partition with the "freebsd-boot" type. It may be possible to have have two "freebsd-boot" partitions (they are limited to 536 KB each) and teach the boot0 step (stand/i386/pmbr/pmbr.S) to use the GPT flags similar to NanoBSD to mark the freebsd-boot partition as failed, and on a successive boot, use the backup bootstrap code. Then have the boot failed flag removed by the loader when it runs successfully, assuming it can detect from which of the two freebsd-boot partitions it was spawned.

For the loader itself, the build world process keeps the previous version of the loader as /boot/loader.old, but relies on an operator interupting the boot2 phase and manually specifying that the alternative loader should be used. Luckily, the loader exists in the boot environment, so a failure here should be solved by zfsbootcfg's boot-once feature.

However, in the UEFI boot case, things are a bit different. Previously, the EFI System Partition (ESP) contained boot1.efi, which had just enough of a filesystem driver to read /boot/loader.efi from the target filesystem and hand off the boot process to it. However, the duplication of features and code complexity have resulted in this approach being replaced with placing loader.efi directly into the ESP. Since boot1 was going to have to be updated to support new ZFS features on a regular basis, it no longer made sense as originally envisioned, a rarely changing shim that would load the more featureful loader.efi. So now we may need to develop a fail-safe mechanism for updating the EFI loader. This can likely be implemented with the EFI Variables service provided by the system firmware as managed by FreeBSD's efibootmgr. The new loader would be installed to the ESP in a different location, and use the EFI nextboot feature to use the new loader only once. If this is successful, the regular loader can be replaced.

Additionally, there are plans to extend zfsbootcfg to have a more expressive configuration. Currently it writes a raw string to an reserved area of the ZFS on-disk label, containing the pool and dataset name to mount the root filesystem for. Another ZFS user, Delphix, has a similar system, except theirs keeps only a counter, of the number of boot attempts. When a system manages to stay up for 10 minutes, it resets this counter to 0. If the boot attempts counter reaches 3, the system instead boots a rescue image that calls home for a technician to intervene. This came out of the constraints of the Amazon cloud environment, where there is no out-of-band console access to resolve boot issues. The author envisions a more structured data store (a packed nvlist) that could contain multiple parameters, combining the existing boot environment selection, a failure counter, a specific rescue environment, and even arbitrary environment variables to set to impact other parts of the boot process.

### 7.3 ZFS

There are a number of features that could be implemented in ZFS to make an upgrade system easier to implement and more powerful. The first is actively being discussed on the monthly ZFS leadership calls, which is a more controlled zpool upgrade process. Rather than upgrading to all of the latest features, this would allow the user to specify a level to upgrade the pool to, such as “compatible-2019”, which is the lowest common denominator of features supported by all main stream ZFS ports as of January 1, 2019. This ensures that a pool can be upgraded to get new features, but won't upgrade past what might be supported by the FreeBSD boot code, or what can be imported on OS X.

A zpool bootcode command to take care of updating the boot code on all disks in the pool could make this procedure much safer. Currently on FreeBSD, when you upgrade a pool, the output of the upgrade command includes an example command to upgrade the bootcode on your disk. However, this example command assumes you are booting in legacy BIOS mode, on a GPT partitioned disk. If you are booting in UEFI mode, or the default configuration the installer uses, where both types of bootcode are installed, this command could end up overwriting the EFI bootcode partition with the legacy bootcode, resulting in an unbootable system. Since it will not be possible for ZFS to accurately predict where the bootcode might need to go, the proposal is to relocate the bootcode to the location specified by the ZFS on-disk format. There is a 3.5 MB reserved area after the first two labels before the start of the filesystem. This is already used for the bootcode if the partition table type is MBR instead of GPT. This would then just involve the zpool command writing the data to space owned by ZFS. It would require a different version of boot0 for GPT, that found the freebsd-zfs partition and read boot1 and boot2 from the correct offset, rather than looking for boot1 and boot2 in the freebsd-boot partition. However, what should it do in the case where there are multiple freebsd-zfs partitions?

In the case of UEFI boot, things get sticky as well. Currently FreeBSD acts a bit different than most other ZFS ports when given an entire disk, rather than a partition. FreeBSD just writes the ZFS on-disk format directly to the device, with no partition table, similar to the old “dangerously dedicated” BSD partition table scheme. In illumos , where ZFS originated, when given an entire disk, ZFS writes a GPT partition table, and creates a single large partition and puts the ZFS contents in that. Recently, the zpool create command has been extended to have an optional parameter to create an ESP partition of a defined size to store the EFI loader, and then use the rest of the space for the ZFS partition. With this configuration, the ESP partition is somewhat controlled by ZFS, and could be overwritten with the newer loader as part of the zpool bootcode command. However, unlike the freebsd-boot parttion, of the reserved area in the ZFS label for other legacy bootcode, the ESP is not unformatted space, it is a FAT filesystem, and designed to container the loaders for multiple operating systems if the machine has multiple operating systems installed. It can also contain EFI native applications for diagnostics. As such, it may not be advisable to overwrite the entire partition with an image that contains only the FreeBSD bootcode. Mounting the ESP and copying files to it doesn't seem like something that should be done by the zpool bootcode command.

Lastly, another ZFS feature in the works is support for a newer compression algorithm, ZStandard (ZSTD). On a default installation of FreeBSD, the current default compression algorithm, LZ4, compresses the image approximately 2:1, but ZSTD can compress as much as 3:1, making the system image, and incremental updates to it, smaller.

### 7.4 bectl and libbe

bectl and libbe are the recently introduced base system utilities for managing boot environments on FreeBSD. Started as a Google Summer of Code project in 2017, the work was later completed by Kyle Evans and merged into FreeBSD 12.0.

In addition to replacing the previous shell script, beadm, bectl also provides a C library, libbe, that can be integrated into appliance control planes and used by other tools, like package managers. Kyle is also in the process of adding “deep boot environment” support, which will handle children under the root filesystem. One of the popular examples for this is having a /usr/src for each boot environment, that is a separate database, but /etc/rc.d/zfsbe takes care to mount the usr/src child for the current boot environment. This way the source tree that corresponds to the running kernel is always mounted. This could also allow things like /usr/local (for packages) to be a separate filesystem, but still be swapped in sync with the boot environment.

Additionally, libbe makes it easier to integrate a customized mechanism for deciding that the boot-once of a system image is working as expected, and make it the new default.

## 8. Additional Considerations

There are still many more factors to consider when designing an upgrading mechanism. Most of what is described here would work in an air-gapped environment, where the updated system image was delivered on read-only media such as a DVD, but some minor changes may need to be made to accommodate this.

A lot more consideration needs to be paid to security and authenticity. An update mechanism needs to be secure from a number of different threat vectors. The updated system images should likely be signed to prove their authenticity, which would require distributing the trusted keys with the original system image. If updates are fetched from the internet, not only do you need to consider the identity of the host you are fetching from, but also the integrity of the new system image. Additionally, there may be a need to hide the details of the update (what previous version the system is being upgraded from), which may require randomizing parts of the request to ensure the resulting response size hides the details of the update.

There is also the considerations around encryption. FreeBSD supports high performance full disk encryption using GELI. There is currently some support for GELI in the FreeBSD bootcode and loader, but it is geared towards interactive use on a laptop. There are some design documents, but no current work in progress to support headless operation of GELI by storing encryption keys on dedicated removable devices (such as a USB stick or smart card). However, since GELI operates at the block layer, below ZFS, it doesn't have much direct impact on the upgrading mechanisms. There is however a forthcoming feature in ZFS to provide filesystem native encryption. This does not encrypt all data on the disk, but provides for confidentiality of the user data and metadata of the individual filesystem. This allows only a subset of the filesystems to be encrypted, possibly with different keys. The main advantage to this design is that individual keys can be unloaded when they data is not needed, placing the data safely at rest. Importantly, the scrub (data integrity check) and resilver operations (recovering from failed disks) can operate without requiring the encrypt keys be loaded. ZFS replication can optionally send the ciphertext rather than the plain text version of the filesystem in the replication stream. This would allow the distribution of encrypted system images and updates. However, FreeBSD does not yet support this feature, and once it does, support for encrypting boot environments would likely be futher behind if it is judged useful to have at all.

## 9. Conclusion

FreeBSD has the tools to build a fail-safe updating mechanism, and with a bit more time to polish it, we hope to arrive at a flexible, yet easy to use solution that works for the majority of FreeBSD machines, be they appliances, embedded devices, servers, or laptops.

## References

[1] Poul-Henning Kamp. *Building a FreeBSD Appliance With NanoBSD*, 2005.

https://papers.freebsd.org/2005/phk-nanobsd/

[2] Colin Percival. *An Automated Binary Security Update System for FreeBSD*, 2003.

http://www.daemonology.net/freebsd-update/binup.html

[3] Allan Jude. *Booting from Encrypted Disks on FreeBSD*, 2016.

http://allanjude.com/bsd/AsiaBSDCon2016_geliboot_pdf1a.pdf