SEE TEXT ONLY

GELI and ZFS IMPROVEMENTS

A lot of new things are coming in FreeBSD 11, but this article will focus on a number of contributions I personally had a hand in. The focus of these changes is improving the way FreeBSD systems are booted, and how system updates are managed.

BCache Improvements

For many years the FreeBSD loader has implemented a simple block cache to improve performance by avoiding physical reads of the same sector multiple times. When this cache was originally developed, most systems booted from a single disk, or a RAID volume that was exposed to the operation system as a single logical disk. With the advent of ZFS, it is very common for the loader to need to access multiple disks in order to read all of the data required to load the kernel and boot the operating system. An IllumOS developer, Toomas Soome, who is working on porting the FreeBSD loader to IllumOS to replace the ancient version of grub that they currently use, developed a number of improvements to the block cache and contributed those back to FreeBSD. The improvements include growing the cache from 16 KB to 16 MB, implementing a rudimentary read-ahead cache, and making it support multiple devices. The 16 MB cache is divided evenly amongst the number of devices detected at boot. This can likely be optimized more in the future.

The read-ahead feature takes advantage of the fact that for spinning media, reading a modest number of contiguous sectors takes almost no additional time compared to reading just a single

sector. When the next sector is read, it is returned from the cache. If two different areas of the disk are being read concurrently, this can provide an immense performance gain. Additionally, support was added for caching data fetched from CD and DVD devices, which previously were not cached. This change in particular made a large difference for people who boot FreeBSD installer images via remote media systems such as IPMI. The kernel-loading portion of booting from a virtual CD (an ISO image) via IPMI over a LAN was reduced from 27 seconds to 7 seconds, thanks to the read-ahead cache. The difference becomes even more stark when the remote case is considered. Without the block cache, when installing FreeBSD on a remote server with 60ms latency, a single 2K block is read from the CD at a time, and the next block is not requested until that read operation for the previous block completes, 60+ms later. With the new read-ahead cache and other bcache improvements, the time to load the kernel from a remote CD dropped from over 12 minutes to under 5 minutes. Now when I have to build out a new server in Singapore with 225ms of latency, I am very grateful for Toomas's hard work. After a number of other contributions, Toomas Soome has officially joined the FreeBSD Project as a committer.

ZFS Boot Environment Menu

The basics of what boot environments are and how they work have been covered before (see the September/October 2015 issue of this publication). To quickly recap, a "Boot Environment" (BE) is an alternative root filesystem. ZFS provides low-cost snapshots and clones, meaning that before a change to the operating system like an upgrade, the operator can take a snapshot to be able to revert the system to in the case of a failed upgrade. These snapshots can be used to create clones, which are writable snapshots. Having a selection of these clones to choose from allows the operator to try different versions of the operating system, without disturbing the currently installed version. Some filesystems like /home are common between the different boot environments. This ensures that even as the operator tries different versions of FreeBSD, their same home directory is available in all of them. This also means updated contents of the home directory are not lost if an upgrade is reverted.

Here is a look at the boot environments that exist on my development machine:

NAME	USED	AVAIL	REFER	MOUNTPOINT
zroot/ROOT	11.3G	2.51T	392M	legacy
zroot/ROOT/9.0_router	392M	2.51T	392M	/
zroot/ROOT/9.1_before_upgrade	16.0K	2.51T	1.92G	/
zroot/ROOT/9.1_freebsd	822M	2.51T	822M	/
zroot/ROOT/9.1_pcbsd	16.0K	2.51T	2.05G	/
zroot/ROOT/9.2_beta1	848M	2.51T	848M	/
zroot/ROOT/9.2_pcbsd	16.0K	2.51T	2.10G	/
zroot/ROOT/before_10_stable_2014-09-11	16.0K	2.51T	2.68G	/
zroot/ROOT/10_stable_2014-03-24	647K	2.51T	3.35G	/
zroot/ROOT/11_1100093	16.0K	2.51T	3.35G	/
zroot/ROOT/11_r295359_zfsdebug	368K	2.51T	3.38G	/
zroot/ROOT/default	8.95G	2.51T	3.42G	/

As you can see, I have upgraded this machine in-place from 9.0-RELEASE through to 11-CURRENT. There is even a BE from when I was debugging a ZFS memory consumption issue in 11-CURRENT. In this BE the kernel contains a lot of additional DTrace probes to try to track the source of the problem. At the end of a debugging session, I could reboot back to my standard system, which was 10-STABLE. Being able to flip back and forth with just a reboot, while maintaining a common /home directory, and not requiring fragmenting my free disk space across separate partitions, is very powerful.

Originally, the only way to manage BEs was with the beadm(1) utility. This posed an obvious problem—if the system failed to boot, how do you run the beadm utility to change the active boot environment to the previous working system? PC-BSD adapted a solution similar to that used in IllumOS, where the beadm utility generates a list of BEs and stores them in a configuration file that is read by the GRUB boot loader. While this mostly works, it requires using a non-standard boot loader, and can run into problems when the config file gets out of sync with reality. The GRUB config file loads its modules from the original boot environment that was active when

GRUB was set up, which can cause the system to be unbootable if that BE is removed and the configuration file is not properly updated.

An easier and more reliable solution was needed. The FreeBSD loader already has the ability to read from ZFS and to list the available datasets, as part of the 'lszfs' loader command. I used this command as a template, and created a new function that populates a set of environment variables with a paginated list of boot environments. The main advantage to this approach is that it reads from the live filesystem, so it is always accurate. With help from Devin Teske, the loader's menu system, written in forth, now displays a menu and allows you to select which boot environment to use as your root filesystem. This was further improved with help from Toomas Soome. One drawback was that the loader menu system was not supported if the system was booted with EFI, because our EFI loader lacked the required serial emulation code. Again, Toomas Soome to the rescue. The missing features were implemented and the ZFS boot environment menu is available in the FreeBSD loader for both BIOS and EFI boot.

GELIBoot

Now ZFS BEs were supported across pretty much any configuration, except when the user opts for full disk encryption. When the pool containing the root filesystem is encrypted, a separate /boot filesystem (either a second ZFS pool, or a small UFS partition) is required. This plaintext partition is where the boot loader, kernel, and modules like GELI and ZFS are stored. The system needs to be able to bootstrap itself, which requires reading the boot loader, which in turn reads and executes the kernel, which loads the module to be able to access the encrypted filesystem. The issue is that this 'two pools' setup, breaks ZFS Boot Environments. When the kernel lives outside of the root filesystem, which is what is snapshotted and cloned to make the various boot environments, it becomes impossible to switch between BEs and load the matching kernel. Even if similar snapshots were managed between the two pools, it would be complex and error prone. A better solution was needed.

Being the naive junior developer that I am, I asked "how hard can it be?" For an initial implementation, a copy of `gptzfsboot` was made with the name `gptgeliboot`. The original idea was to create a single bootcode that could boot from an encrypted UFS or ZFS filesystem. Subtle differences in bits of the `boot2` code, and no clear way to define the behaviour of systems that use both UFS and ZFS filesystems, caused this approach to be abandoned. It was decided to implement optional GELI support in each of the existing GPT bootcodes instead. Initially, it was necessary to determine if the boot partition was GELI encrypted. As with most all GEOM classes, GELI stores its metadata in the very last sector of the provider, which is usually a partition, to avoid conflicting with the backup copy of the GPT partition table that is stored in the last sector of the disk. The task seemed simple—read the partition table, identify the starting LBA of the partition and its size, and read the last sector of that partition. The hardest part about working in the bootcode is that there are no error reporting facilities. There isn't even a `panic()`. Pretty much all there is to work with is `printf()`, and when things go bad, the system just hangs, unless you manage to crash the BTX loader, which will give you a dump of the assembly instruction pointers and the like. This made development very iterative and almost brute force. Make a change, build, install it, reboot, fail, add printf, build, install, reboot, fail, repeat. Of course, you must moderate the quantity of printfs, because there is no pager; once data scrolls off the top of the screen, it is gone forever.

After the partition has been determined to contain encrypted data, the GELI metadata needs to be read to determine which algorithm to use to decrypt it. Then the encrypted copy of the master key must be decrypted with the user-provided password, and only then is it possible to read from the partition. The existing bootcode happened to be structured in such a way that it was relatively easy to add decryption to the regular read functions when GELI was determined to be present on a partition. The first large roadblock was encountered after adding all of the dependencies for GELI (SHA256, SHA512, HMAC, AES -CBC, and AES-CTS). The size of the boot2 file (gptzfsboot) has grown from 47kb to 90kb. The boot1 code (gptldr), a 512 byte block of assembly that loads boot2 into a specific position in memory and executes it, only loaded the first 64kb of boot2. Because this code runs in 16-bit real mode, this is the largest block of data that can be managed at once. After many failed

attempts, and asking for assistance from many senior members of the community, Colin Percival finally solved the problem and extended `gptldr` to load a specified number of 32kb segments of boot2, allowing for it to easily be extended in the future.

In order to reuse existing code from GELI and OpenCrypto (the kernel's AES implementations, used in IPSEC), some changes needed to be made. A few declarations and ifdef's needed to be moved in the GELI code to allow it to be compiled for userspace. Once this was accomplished I set to the task of breaking the OpenCrypto framework up into separate files per algorithm, rather than a single monolithic file. This made it possible to reuse this code rather than importing another copy of AES-XTS for use in the loader.

Much to my surprise, the GELI source code was very elegant, and easy to follow, even by a junior developer who had little prior experience with cryptography. With this work complete, it is now possible to boot FreeBSD from an encrypted UFS or ZFS filesystem, with the only plaintext being the contents of the miniscule freebsd-boot partition (gptboot or gptzfsboot).

At this point, the system can be booted, but it prompts the user for the encryption passphrase an inordinate number of times. The test system was a ZFS mirror of two disks. `gptzfsboot` would prompt for the password for each of the two disks, then load the loader. The loader would then prompt the user for the passphrase for each disk, and load the kernel. At the mountroot prompt, the kernel would prompt for the passphrase for each disk, and finally the system would boot. With more than a few disks, this quickly becomes exceedingly cumbersome. Colin Percival, Devin Teske, and Kris Moore had already been suffering from similar problems and developed a solution. Colin implemented the `kern.geom.eli.boot_passcache sysctl`, which caches the password entered by the user at the mountroot prompt and attempts to reuse it on each new disk that is tested during the boot process. This was extended with Colin's help by Kris Moore, to allow the passphrase entered in the GRUB2 boot loader to be passed via the kernel's environment to GELI, so that if the password was correct, it would avoid re-prompting for the password at the mountroot phase. This avoided the issue where the mountroot password prompt would become buried by late device attach notices. Devin Teske added an option to `loader.conf, geom_eli_passphrase_prompt` that would cause the FreeBSD loader to prompt the user for the GELI passphrase ahead of time, and pass it to the kernel via the environment, the same way PC-BSD's GRUB2 was doing it, again with the goal of avoiding the mountroot prompt. Extra care is taken by the GELI kernel module to zero the passphrase from the environment before single-user mode starts. The password prompt that was implemented in GELIBoot was given a similar caching mechanism whereby it automatically tries the previously entered passphrase, and only if that fails, gives the user three attempts to enter the correct passphrase.

The GELIBoot code needs the passphrase the earliest, in order to read the loader from the encrypted disk. This raised the obvious question, how to pass the passphrase from the `boot2` stage, to the loader. The answer lay in `gptzfsboot`, where a flag is set, `KARGS_FLAGS_EXTARG`, which tells the loader to look for an additional argument after the end of the regular set of arguments. Here, it will find `struct zfs_boot_args`, containing information such as the pool that is being booted, and the root filesystem. The first member of this struct is 'size', which is set to `sizeof(struct zfs_boot_args)`. This allows the loader to safely access newer members of the struct, by first checking that the `offsetof()` the member is not greater than the `sizeof()` the loader's definition of the struct. This allows mismatched versions of the bootcode and loader to continue to work together. When a new member is added to the end of the struct, access to it is guarded by this mechanism. Using this design pattern, a new member was added to the `zfs_boot_args` struct to pass the GELI passphrase from `boot2` to the loader. This too is carefully zeroed as soon as the next boot phase starts.

Work then started on updating the FreeBSD installer to create this configuration for users. There are a number of limitations. The GELIBoot system currently only works with GPT formatted disks. MBR formatted disks are not supported because of a similar restriction in the size of the boot2 code. Originally it was not clear that the ZFS on-disk format allowed a larger bootcode to be installed. After discussion with Toomas Soome, I learned that the ZFS disk label leaves 3.5 megabytes of space for boot code, so it will be

possible to expand the MBR boot2 code. This will likely be required as ZFS grows additional features anyway, so GELIBoot will most likely be extended to include MBR support in FreeBSD 11.1. Currently the FreeBSD EFI boot loader does not support GELI. Another developer, Eric McCorkle, who wrote the original EFI ZFS support, is working on this, and it is also expected to be included in FreeBSD 11.1. However, if you use GPT and boot via the legacy/BIOS method, the installer will create just a single fully encrypted ZFS pool and boot from that. In all other cases, the previous method of using a second plaintext pool to store the loader and kernel is used.

The remaining limitation is also that only GELI passphrases are supported. The GELI master key is encrypted using a key derived by PKCS#5 v2 from the passphrase. GELI also supports using a key file (or the combination of both a passphrase and a key file); however, GELIBoot does not support key files. In the future, support for key files stored on external media such as a USB device is planned. With the UEFI version of GELIBoot, Eric plans to support storing key material in the TPM.

You can read more about my adventures in implementing GELIBoot in the proceedings of AsiaBSDCon 2016, or download my paper here: http://www.allanjude.com/bsd/AsiaBSDCon2016_geliboot_pdf1a.pdf.

Automatic Boot Recovery

FreeBSD contains a toolkit called nanobsd that has been used by many projects, including FreeNAS and pfSense, to build appliances. It contains a firmware-style upgrade feature, where the system is set up with two partitions, and when an upgrade is performed, the new system is installed into the inactive partition, and a partition table flag is set. On the next boot, the partition containing the newer system image will be booted, and the failed flag will be set on it. If it boots successfully, the flags will be removed from the partition. If booting fails, when the system is power cycled, the boot loader will see the failed flag and automatically revert to booting the original image.

There is a desire to have a similar system for ZFS, without requiring separate partitions. The basic idea is to set a pool property, altbootfs, to indicate a new boot environment to attempt to boot. When the boot loader detects the presence of this property, it will mark the partition as failed, and boot from the alternate boot environment. If the system comes up properly, a startup script will remove the failed flag and promote the boot environment to being the default. If the system does not boot correctly, when the next boot is attempted, it will boot the default boot environment, which contains the system from before the upgrade was applied. This will protect appliances and servers from requiring hands-on assistance to recover from a failed upgrade.

ZFS RAID 10

The FreeBSD installer has supported creating mirrored ZFS pools since 10.1; however, if your system contained a large number of disks and you selected the mirror option, the installer would create a single mirror containing all of the disks. While this provides extremely good redundancy with 8 or more drives, this is likely not the user's intent. In FreeBSD 11 the installer includes a RAID-10 option, which creates a ZFS mirror consisting of pairs of disks. An even number of disks must be selected for this option to be available. A system with 8 disks will now create a pool of 4 mirror sets each containing 2 disks. This is the highest performance configuration, especially in terms of IOPS. Mirror sets also provide the most flexibility. Additional disks can be added in pairs, rather than requiring an entire additional RAID-Z vdev. In budget-constrained configurations, this allows small numbers of disks to be purchased when available space runs low. Mirror sets also allow additional space to be gained by replacing smaller disks with larger ones. Once both members of a mirror have been individually replaced and resilvered, the additional space is available. With an 8-disk RAID-Z, all 8 disks would need to be replaced before additional space became available.

And So Much More

The rest of the FreeBSD community has not been idle during this time either; a lot of great work has been done recently. FreeBSD 11 will also include:

- The ZFS ARC is now resizable at runtime, allowing greater control over the amount of memory used by ZFS
- Improved interaction between ZFS and the VFS layer improves responsiveness under memory pressure and allows ZFS to return more memory to the system when needed

- ZFSd is now available; this daemon manages hot spares and automatically reattaching disks that are temporarily detached
- ZFS now supports the SHA512 and Skein checksum algorithms
- The in-kernel SHA2 implementation has been replaced with a much more performant one
- bhyve now supports Windows Guests, and has a VNC backend to provide video console access
- libxo allows a number of utilities in the base system to output JSON, XML, or HTML instead of only plaintext
- Ifconfig now supports multiple output formats, including printing subnet masks in dotted-quad and CIDR notation, and traditionalhex output
- More features are enabled by default, including Netmap and IPSec

More to Come

While 11 is a major milestone, there is plenty more to come. The 11.x branch features a new support model, where each point release is supported for three months after the next point release. This new model will allow more frequent releases and greatly lessen the burden on the Security and Ports teams having to support 2-year-old releases. Many new features will be available in 11.1, although for some of the major changes you'll have to wait for 12. Over the next few months I hope to see a number of new features land:

- Packaged base system
- GELI in the UEFI loader
- Support for more ZFS features in the loader
- ZFS compressed ARC
- ZFS compressed send/recv
- ZFS scrub/resilver speed improvements
- Installer create larger EFI partitions
- Easier management of EFI partition and its contents
- Installer put swap partition before data partition to allow data partition to be easily grown
- Installer optionally install a basic set of packages (graphical environment)
- libucl (standardized configuration file format) config files for many utilities and daemons in base
- libxo in more utilities
- librification of more utilities (ifconfig, netstat)

ALLAN JUDE is VP of Operations at ScaleEngine Inc., a global HTTP and Video Streaming CDN, where he makes extensive use of ZFS on FreeBSD. He is also the host of the video podcasts BSDNow.tv (with Kris Moore) and TechSNAP.tv. He is a FreeBSD src and doc committer, and was elected to the FreeBSD Core team in the summer of 2016. Allan is the coauthor of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W. Lucas.