GELI Support for UEFI

Eric L. McCorkle

September 9, 2017

Disclaimer

The content of this presentation does not constitute a statement on behalf of or represent the position of any company or organization. The work described herein was not carried out on behalf of any company or organization, including any employer.

Background

- `geli` is a block-level disk encryption scheme for FreeBSD.
- Support for booting in X86 BIOS mode from `geli` volumes added by Allan Jude.
- UEFI boot is very different from X86 BIOS.
- GELI support for UEFI necessary to support modern hardware, UEFI features (secure boot, UEFI variables, etc.)

UEFI Boot Process

- UEFI specification provides a number of APIs for device I/O, memory allocation, driver registration, etc.
- *No* direct access to devices, control over addresses, etc.
- Firmware looks for EFI System Partition (ESP), loads boot application from standard location
- UEFI spec calls for minimum 200Mib ESP, can be larger
- FreeBSD UEFI boot process has two steps
- `boot1` is a UEFI application installed to ESP, looks for boot partition, loads `loader.efi`
- `loader.efi` presents the standard FreeBSD boot shell

Table of Contents

UEFI Benefits and Challenges

- Benefit: (mostly) free from space constraints, unlike X86 BIOS
- Challenge: level of abstraction precludes safely passing arbitrary memory from `boot1` to `loader.efi`
- Challenge: harder to properly implement GELI driver
- Challenge: split code base between `boot1` and `loader.efi`
- Benefit: a lot more tools to work with

Issue: Key Transmission

- User should only input password once
- Naïve implementation would require *three* separate times
- Need to transmit keys from `boot1` to `loader.efi`, then to kernel
- X86 BIOS pushed password onto `loader`'s stack, then as an environment variable for the kernel
- UEFI has stronger separation between stages
- Want to support multiple passwords
- Hashed passwords better (only incurs one hashing delay)
- Ideally, provide straightforward migration to hardware key storage mechanisms

Issue: Split Codebase

- `loader.efi` uses `libstand` API with UEFI backend
- `boot1` used completely separate codebase with its own interface
- Codebases were almost completely independent, significant duplication
- `boot1` codebase tended towards minimality, difficult to maintain and improve
- Any change would require two separate implementations with different underlying designs
- This code duplication hampered both current as well as planned future work

Issue: GELI Driver Architecture

- GELI is designed around GEOM, a multi-layered device interface
- Can support arbitrarily-complex schemes (GPT/GELIs inside GELIs, and so on)
- Boot loader support is more limited
- `boot1` would require complete overhaul to support GELI-like structures
- `loader.efi` has ability to support "one-layer" schemes (GELIs on partitions)

Table of Contents

Timeline of Work

- Attempt to refactor `boot1`
- First attempt at unifying `boot1` and `loader.efi` codebases (EFIzation)
- First working GELI driver!
- Time in code review, use on real hardware
- ZFS boot environment issues identified, non-trivial changes to HEAD
- Design revision, simplification, establishment of new branches
- Key intake buffers go into kernel
- Full-disk root-on-ZFS under GELI working on real hardware

Table of Contents

First Refactor of `boot1`

- Purpose was to "rough out" a PoC
- Introduced a "providers" API to compliment boot modules
- Created even more code duplication, highlighted need to unify codebases
- Abandoned in favor of unification

Table of Contents

UEFI Driver Primer

- EFI_DRIVER_BINDING API allows registration of new drivers
- Drivers have probe/attach functions for EFI_HANDLEs
- Can attach various interfaces to an EFI_HANDLE
- Can also create new EFI_HANDLEs to represent virtual devices
- New devices will be automatically probed by all registered drivers
- UEFI spec guarantees GPT/MSDOSFS drivers

EFIzation Effort

Observation: UEFI provides an API with the same functionality as `libstand`; use it instead!

- First effort to unify `boot1` and `loader.efi`
- `boot1` and `loader.efi` would use `EFI_SIMPLE_FILE_SYSTEM` interface
- Produced "shim" drivers: UEFI-to-`libstand`, `libstand`-to-UEFI
- `libstand` drivers sat under `EFI_SIMPLE_FILE_SYSTEM` interface
- `boot1` directly utilized `EFI_SIMPLE_FILE_SYSTEM` to find and load `loader.efi`
- `loader.efi` continued to use `libstand` interface, which talked through the other shim to UEFI API

UEFI Drivers

- GELI was implemented directly as a UEFI driver
- GELI used `EFI_DRIVER_BINDING` API to register itself as a driver, created new device handles for GELI volumes it detects
- Benefit: this carries over across the `boot1`/`loader.efi` boundary
- `efipart` *mostly* converted to a UEFI driver
- Issues with `bcache` prevented full conversion

Table of Contents

Managing Keys in the Loader

- UEFI driver interface solves one half of the problem raised by the `boot1`/`loader.efi` gap
- `EFI_HANDLE`s registered in `boot1` are available in `loader.efi`
- This provides *access*
- Still need *keys* to pass into the kernel

UEFI KMS Interface

- UEFI defines a key management system (KMS) interface
- Implemented a simple in-memory key database as a driver which provides this interface.
- GELI driver attempts to locate a KMS during initialization
- GELI stores/retrieves keys from its KMS
- Kernel metadata step also locates the KMS, transfers all keys into the kernel via the `keybuf` interface

Kernel Key Intake Buffer (`keybuf`)

- Provide a better way of getting keys into the kernel
- Uses kernel metadata functionality to deliver (by default) up to 64 keys, each up to 4096 bits long
- Keys have a type code indicating their format
- Picked up by `crypto`, then subsequently available to other drivers for initialization
- GELI passes in hashed passwords
- Designed to be extended to work with hardware crypto

Boot Crypto Framework (boot_crypto)

- Inherited code from X86 BIOS implementation, but created a separate codebase
- X86 BIOS is space-constrained and only supports AES; UEFI is not space-constrained
- boot_crypto is designed around a generic algorithm interface with pluggable backends
- Designed to anticipate overhaul of crypto framework
- Also designed to support hardware crypto device implementations

If/When Trustworthy Hardware KMS/Crypto Exists. . .

Thinking ahead to a time when there is a trustworthy hardware KMS implementation was a consideration in this design

- In-memory KMS detection aborts if it detects another KMS device (this also deals with `boot1` and `loader.efi` both having to attempt to register the in-memory KMS device)
- GELI should "just work", as it talks through the KMS and `boot_crypto` interfaces
- `boot_crypto` would need to add support
- "Keys" would likely consist of ID numbers for keys stored in KMS
- `keybuf` interface could easily add another key type for key IDs

Table of Contents

Benefits of EFIzed Approach

- `boot1` reduced to a very minimal program, uses same codebase as `loader.efi`
- Seamless integration with firmware-provided drivers
- Dropped MSDOSFS driver
- Provided framework for hot-plugging support (`bcache` got in the way of full implementation)
- Laid groundwork for exporting driver code to others

Drawbacks of EFIzed Approach

- UEFI does a bad job at supporting non-Microsoft systems and interfaces
- `EFI_SIMPLE_FILE_SYSTEM` interface is designed around MSDOSFS, sits uncomfortably in a VFS interface
- Difficult to present the same information in boot shell as in current `loader`
- ZFS boot environments lost when talking over UEFI interfaces

Personal takeaway: started with moderately positive views on UEFI, ended with moderately negative views.

Simplified Unification

Eventually, code changes in HEAD broke the patches in non-trivial ways, and the drawbacks of EFIzed approach were becoming clear.

- Moved UEFI-to-`libstand` shim out to an independent review (still up for review)
- Dropped `libstand`-to-UEFI shim altogether
- Refactored `boot1` to use `libstand`
- Recovered simplicity, information at boot shell, ZFS boot environments
- Casualty: progress towards hot-pluggable devices at boot time

Refurbishing Efforts

- efipart had moved away from a static-numbered, array-based storage scheme for device handles (right move)
- efipart had also split up device handles by drive type (also right move)
- Found an integer overflow bug in efipart_realstrategy when attempting to read past the end of a device (caused crash)
- efipart was manually parsing partition tables and using base device handles
- This didn't work at all with GELI, so had to revert to direct access through partition device handles

Table of Contents

Kernel gets `keybuf` Interface

- `keybuf` patch went into kernel first
- X86 BIOS GELI support started using `keybuf`
- Legacy environment variable method still supported, eventually to be phased out
- *Definitely* the right move to put `keybuf` in first
- Anyone using a recent kernel can use UEFI GELI without a kernel update

Testing on QEMU and Real Hardware

- QEMU testing setup had a large number of GELI disks, including encrypted/unencrypted UFS, ZFS, also an X86 BIOS setup
- Tested all combinations on QEMU
- I had also been using a root-on-ZFS laptop with its L2ARC/Intent log stored on GELI volumes since the EFIzed version
- Finally, converted a laptop over to a full GELI root-on-ZFS setup
- Works perfectly (except I forgot the `-R` when taking the ZFS snapshots. . .)

Table of Contents

Plans for GRUB/Coreboot

- GRUB is (reportedly) the best way to achieve a Coreboot setup
- Coreboot is arguably a better option (where it's supported)
- GRUB already supports GELI, but needs to be updated to use the `keybuf` interface
- Initial conversations with GRUB developers indicates this shouldn't be hard

My Long-Term Plans

My overall agenda can be described as "OS-level tamper-resilience"

- Full-disk encryption (GELI)
- Trust framework and kernel/module signing
- Active use of coreboot/setup guides
- Secure suspend/resume
- Other uses of trust framework