Design of a Public-Key Trust System for FreeBSD

Eric L. McCorkle

June 7, 2018

Motivating Example

Consider a proposal for signing the kernel and modules:

- Extend Executable Linkable Format (ELF) to carry public-key signatures
- Sign kernel and modules with a private key for each build
- Kernel and boot loader carry the verification (public) key
- Loader checks kernel/module signatures before booting
- Kernel checks module signatures before allowing them to be loaded
- UEFI and GRUB both have equivalent facilities

Cryptography as Trust

Signed kernels and modules are an example of *cryptography as trust*.

- Cryptography is most often viewed as a *confidentiality* mechanism
- However, it can also fulfill other purposes, such as *authorization*
- In FreeBSD (and many other systems) the kernel enforces authorization rules
- Relies on memory protection, internal tables, user IDs, etc to restrict *who may access/modify*
- Signed kernel modules allow authorization to restrict the *content* of the modules/kernel

Public-Key Cryptography in System Context

Public-key cryptography can extend and/or strengthen many security features of operating systems:

- Signed kernels, modules, executables, libraries
- Distribution and delegation in a capabilities-based access control system (capsicum)
- Strong (cryptographic) data access controls
- "Traditional" public-key functions (session key negotiation, protocols)
- System-level trust management

Trust Management

Trust management is vital in a public-key system.

- Some public key (or set of them) serves as a *root of trust*
- Trust can be extended to additional keys through signatures
- Chains of trust can be formed by signing each successive key with the previous key
- Public-Key Infrastructure (PKI) systems allow for a tree-like structure
- Other systems (PGP) use a web-of-trust (general graph)

Table of Contents

Trust System Design for FreeBSD

- *Runtime Trust Database*: In-kernel API for managing root/intermediate keys
- `devfs`-based interface for adding/revoking intermediate keys from userland
- *Trust base configuration*: Configurations for building in root keys, loading intermediate keys at boot.
- Signed ELF binary format extension, conventions for signing vital config files
- NetBSD VeriExec integration

Kernel API

- Root certificates are established at boot, cannot be changed during runtime (without hardware intervention)
- Database tracks trust relationships, forms a forest with root keys as roots
- Intermediate certificates can be added, providing they are signed by an existing root or intermediate certificate
- All keys have a revocation list (initially empty), can be set for any key
- Any intermediate certificates in their signers' revocation lists are removed, along with their descendents
- Can check a signature against the database
- Can enumerate database

devfs Interface

- Present device nodes under `/dev/trust/`
- Control interface at `/dev/trust/trustctl`:
  - Write an X.509 certificate signed by a trusted certificate to install as an intermediate certificate (check revocation lists)
  - Write an X.509 revocation list signed by a trusted certificate to install it as the signer's revocation list (and do revocations)
  - Use binary DER encoding (easy/safe to parse) for input
- Enumeration interfaces at `/dev/trust/certs`, `/dev/trust/rootcerts`:
  - `/dev/trust/certs` reads back all certificates
  - `/dev/trust/rootcerts` reads back just roots
  - Read back certificates in PEM encoding, allows nodes to be used as CAcert configuration for many applications
- Could also render entire forest as directory structure

Obtaining Root Keys

There are several options for obtaining root keys at startup:

- Build directly into loader/kernel
  - Advantage: Secure, better cipher suite
  - Disadvantage: Inflexible, difficult to recover from mishaps, bad for standard images
- Obtain from secure boot infrastructure or hardware
  - Advantage: Integration with hardware/secure boot, flexible
  - Disadvantage: Often weak crypto suites (RSA 2048 is as good as it gets)
- Pass from loader to kernel via `keybuf`
  - Advantage: Flexible, full cipher suite
  - Disadvantage: Less secure than compiling in

Trust Base Configuration

- Establishes trust configuration for builds and system startup
- Store trust root certs at `/etc/trust/root/certs` (keys at `/etc/trust/root/keys` if we have them)
- Intermediate trust certs at `/etc/trust/certs` are loaded by `rc` at boot
- Trust root keys converted to C source, compiled into a static library
- Ultimately compiled into loader and possibly kernel
- Kernels may be signed with an ephemeral intermediate key, stored at `/boot/kernel/cert.pem`

Example Trust Configurations

- Preferred configuration is one locally-generated trust root key
- Third-party vendor certs don't have a corresponding signing key
- In the preferred configuration, all vendor keys are signed by the local trust root key
- Standard distributions can be signed with FreeBSD foundation's vendor key
- Likely will want to have installer generate the local key, then inject it into the loader, then sign FreeBSD's vendor cert
- Alternative config for high security networks has no local keys, only the network's vendor cert, builds produced and signed on a central machine

Formats and Tooling

- OpenSSL is part of FreeBSD base system
- X509 certificates used by many applications, sensible format
- DER binary encoding is best for input format to device nodes
  - Easy to parse
  - Disinguished encoding allows byte-to-byte comparisons
  - Can be generated by `openssl` command-line tool
- PEM encoding is preferable for outputting trusted keys (used by many applications)
- DER for input, PEM for output

Signed Executables

- ELF file format based on sections, already has conventions for extra metadata (DWARF, `.comment`, `.note`, etc)
- Cryptographic Message Syntax (CMS) supported by OpenSSL/PKI, allows for detached signatures
- Signed executables have a `.sign` section, containing a CMS detached signature
- Signatures are computed with a same-sized, zeroed-out `.sign` section
- Signatures in this scheme can be added/verified/removed using `objcopy` and `openssl`

A Note on Alternatives

Several altenative approaches exist:

- GRUB uses detached GPG signatures
- Linux has a system call-based kernel keyring feature

Reasons for not going with the alternatives:

- Signed ELF binary scheme is compatible with existing tools/installers; detached GPG signatures aren't
- `devfs` control interface can be used by existing applications w/o modification
- PGP-compatible tools not in FreeBSD base system
- Web-of-trust is arguably the wrong model for such a system
- Revocation in PGP systems done by the key *owner*, not the signitories

NetBSD VeriExec Framework

The NetBSD VeriExec framework also provides a file integrity checking mechanism

- MAC registry specifies authentication codes for arbitrary files
- MACs are checked upon loading files, `execve` calls, etc.
- Advantage: out-of-band integrity checks (doesn't require in-file signatures like signed ELF)
- Cannot manage *delegated* trust, less flexible than a public-key mechanism
- Basic integration: allow MAC registries to be loaded at any point, if signed by a trusted key

UUID-Marked Executables

- VeriExec associates MACs with a path; can be inflexible
- Signed ELFs can be moved around freely (advantage of in-file metadata)
- Hybrid mechanism: add a UUID to each ELF, can be generated with 128-bit hash (SHA-1, RipeMD-128)
- Allow VeriExec to associate MACs to UUIDs as well as paths
- Executables can be marked with UUIDs once, never need to be modified to add additional signatures
- UUID-marked executables can have other administrative uses

Table of Contents

The Quantum Machines are Coming!

- Decent estimate: quantum machines capable of attacking existing public-key crypto likely to arrive some time between 5 and 50 years from now
- Hidden-subgroup attack breaks RSA, elliptic-curve/classical discrete logs (all common public-key crypto)
- Grover iteration: quadratic-speed attack against symmetric-key, MACs, hashes (halfs bit security)
- Grover iteration is a theoretical attack (requires large quantum memory, very long stability)
- Short version: symmetric-key, hashes, MACs safe, public-key exchange/signatures broken

Post-Quantum Cryptography

- *Post-quantum* cryptography aims to develop *classical* cryptographic methods that are secure against quantum attacks (distinct from quantum cryptography)
- Viable post-quantum key exchange being deployed (SIDH)
- Post-quantum signatures don't have as nice a picture
- Hash-based signatures: reliable, very mature (date back to Lamport) but have serious caveats
- Other post-quantum signature schemes are still under active research, too new, or extremely impractical ($>$ 1Mib signatures, etc)

Hash-Based Signatures

- XMSS: Stateful hash-based signatures
  - Good for finite number of signatures
  - Signature size varies, but reasonable parameters give 1-4Kib
  - Non-standard interface: updates "state" on every signing operation
  - Re-signing with old states destroys security properties
  - Adam Langley: "Giant foot-cannon"
- SPHINCS: Stateless (big) hash-based signatures
  - Classic public-key signature interface, no state
  - Signature size is 40Kib

Using Hash-Based Signatures in Trust

The trust framework provides use cases where both schemes can be used practically:

- Stateful signatures are ideal for batch-signing: create key-pair, sign, destroy private key
- Stateful signatures also good for non-persisted key used, controlled by kernel, generated at boot and destroyed at shutdown.
- Could use stateful signatures to issue delegated credentials valid only for system uptime
- SPHINCS signatures good for signing big messages, or signing relatively small numbers of messages
- Ideal for VeriExec manifests (likely to be much larger than 40Kib)

Table of Contents

Applications

- Signed kernel and modules
- Trusted boot
- Signed executables/configuration files
- System-wide certificate configurations
- Delegation of capabilities to remote systems

Implementation Roadmap

- Crypto library for kernel/loader (crypto overhaul is an open topic)
- In-kernel runtime trust database, `devfs` interface
- Modify loader to check signatures
- Add code to check kernel module signatures
- Implement `signelf` (done)
- Modify build system to produce static library containing root keys from trust base config, sign executables
- Modify `rc` to load intermediate certificates at boot

Crypto Overhaul (brief)

- Kernel crypto, OpenCrypto generally in need of overhaul (old, poor organization)
- No public-key, no PKI parsing
- Loader only has a stop-gap measure for implementing GELI
- Popular options:
  - Import OpenSSL (tried once, failed)
  - LibreSSL (developed by OpenBSD)
  - BearSSL (new, still under development)
  - Earlier versions of this proposal included minimal PKI library
- Any solution will need to add new ciphers (use FreeBSD OID space to create new OIDs, upstream to crypto)

Kernel Key Database, `devfs`

- Basic forest data structure with public keys/revocation lists
- Hardware interface: likely have abstraction layer for storing individual keys
- Maintain forest structure in kernel
- `devfs` interface ends up being a straightforward use of kernel API
- Kernel/Loader then has API for checking public-key signatures (main goal)
- Use this to check signatures on executables, files, etc.

The `signelf` Utility

- Batch signer, streamlined tool for signing large numbers of ELF binaries
- More convenient than using `objcopy`/`openssl`
- Gets keys/certs from system trust configuration by default
- Can generate an ephemeral key-pair for signing
- Writes out verification key for ephemeral key-pair, destroys signing key
- Initial implementation using OpenSSL complete

Build System, `rc` Modifications

- Convert trust root certificates into C code early in build
- Create static library (`librootkeys.a`)
- Loader and Kernel can then compile in keys
- `rc.d` script to install intermediate certificates/revocation lists via `devfs` interface

Table of Contents