CSE522 Final Project Proposal - Kernel Space Executable Code Signing

Sam Frank, Ethan Vaughan, Erik Wijmans

November 2016

Contents

0.1 Modification Proposal 1
0.2 Purpose 1
0.3 Methodology 2
0.4 Files 3
0.5 Structures 3
0.6 Control Paths / Other Features 3
0.7 Proposed Tests / Experiments 4
0.8 Five Week Schedule 4

0.1 Modification Proposal

We are proposing a modification to the 4.4 Linux kernel, as well as an extension to the Executable and Linkable Format (ELF), to provide code signing verification and validation on binary execution. We will also create a small user-land application to generate and embed code signing certificates in said ELF files.

0.2 Purpose

Executable corruption, either intentionally by malicious actors or unintentionally by a variety of environmental factors, is a great security hurdle for any operating system. Linux distributions typically include some form of code signing / authentication in their package management systems, but (to our knowledge) no solution exists at a kernel-space level.

Our modification will prevent any binary execution by the `exec()` family of system calls on unsigned ELFs, ELFs that have changed hashes since their

original signing, or ELFs that have been signed by a non-trusted authority.

The signing will be two phase; first by the developer, and second by a central code-signing authority. This authority will be trusted by the kernel, which will store its public key, along with others it trusts.

0.3 Methodology

1. The application developer generates an asymmetrical RSA key pair.
2. The application developer gets verified by a trusted code signing authority, which grants them a signed certificate. The certificate is the athority's public key, developer's public key, and some identifying information (developer name, email, etc). The certificate is hashed, and that hash is encrypted with the authority's private key.
3. The application developer invokes our code singing application.
   (a) The signing application hashes the input binary past the header with SHA-3.
   (b) The signing application encrypts the hash with the developer's private key.
   (c) The signing application modifies the ELF header to include the developer's encrypted binary hash and the authority-signed developer certificate.
4. The user attempts to execute the ELF file via the `exec` family of system calls.
   (a) The kernel verifies that the certificate was signed by a trusted authority. This involves decrypting the certificate signature (hash) with the authority's public key, and comparing the hash to a newly calculated one on the rest of the certificate. If this fails, the call fails.
   (b) The kernel uses the developer's public key (from the certificate) to decrypt the binary's hash field. If this fails, the call fails.
   (c) The kernel hashes the ELF file past the header.

(d) The kernel compares the decrypted hash with its computed hash. If they're equal, `exec` continues as normal. If they're not equal, the call fails.

0.4 Files

`exec` family of system calls, `kernel crypto`, ELF format, linux file extensions

0.5 Structures

We do not anticipate needing to modify any kernel data structures for this project. We will, however, be modifying the ELF file format to include code signing information.

0.6 Control Paths / Other Features

The main control path that we are targeting for modification is the `exec()` family of system calls. Upon first launch of an executable, we must verify that the binary being launched is properly code signed. This will require the steps outlined in Methodology. Unfortunately, code signing verification is an expensive process, primarily due to the cost of hashing the binary. We plan to work around this issue by only verifying the binary the first time it is launched. After that, we will only perform code sign verification if the binary has been modified.

In order to accomplish this, we will have to store the binary's "last verified" time, in order to compare it to the last-modified time. Ideally, this information should be stored in the filesystem so that it can be persisted across system restarts. We plan to use extended filesystem attributes for this purpose. The last verified time will be stored and read in an extended filesystem attribute. In addition, we will use the security extended attribute class to lock down this data so that it cannot be read/written outside of the kernel.

0.7 Proposed Tests / Experiments

1. Load a properly signed binary
2. Fail to load an improperly signed binary
3. Fail to load a tampered with binary
4. Load a cached binary without having to rehash it (if and only if it wasn't modified since the last time it was fully checked)
5. Load cached binary that has been modified and properly rehash/recheck it.
6. Fail to load cached binary that has been tampered with.

Hopefully the operating system still has some semblance of performance

0.8 Five Week Schedule

1. Investigate kernel cryptographic APIs

   For **week 1**, we will start with investigating the cryptographic APIs that are available to us in kernelspace. This will likely involve modifying the kernel so it can generate a SHA-3 (or similar) hash of data that we supply, as well as decrypt an encrypted piece of data using some asymmetrical encryption algorithm (e.g. RSA).

2. Modify ELF

   For **week 2** we will modify the ELF format to include a hash of the file and hopefully we will be able to do this without ruining everything! If this goes well, we can add in a signed certificate during this week also.

3. Modify `exec`

   For **week 3** we will modify the exec system call to only load binaries that hash to the same value as is stored in the ELF file format. If this goes well, we will also add support for checking the signature of both the hash and the certificate.

4. Caching hashes

   For **week 4** we will investigate the use of extended filesystem attributes for storing the verified status of a binary. This will allow us to only perform codesign verification the first time the binary is loaded, and subsequently if the binary changes on disk.

5. Finishing things up

   For **week 5** we will complete (or die trying to) all the things that we didn't finish in weeks 1-4 because it proved more complicated than we initially thought.