Verifiedexec: An Introduction

Brett Lymn

Origins

- Idea formulated late last millenium
- A sudden rise of trojans and root-kits
- Why should the kernel run or read anything it is told to?
- How could the kernel tell if a file had been modified?

Original Implementation

- Decided to use an in-kernel list of fingerprints
- On file access the fingerprint of the target file is evaluate and compared to the in-kernel list
- The obvious problem is performance, 1.7x slower to build a kernel with verifiedexec evaluating fingerprints every time
- Evaluating every time also kills demand paging
- Use caching of the fingerprint evaluation to reduce performance impact to about 5%

Original Implementation 2

- The problem using caching is that the boundary of trust only extends to the walls of the machine case. NFS and SAN storage are a problem.
- There is a fix for this problem, more on this later
- Initial code was implemented and found to work as expected
- During the implementation it was found the exec path for a binary and a shell script were different.
- The implementation took advantage of this difference to provide an interesting feature.
- Code committed to the NetBSD source tree in late 2002.

Current State

- The kernel code has has many improvements:
- switched from a linear list to hash tables for the in-kernel fingerprint list
- uses file generation numbers instead of inodes (no longer FFS specific)
- support for more fingerprint hash functions

- ability to configure out certain hash functions
- removed abuse of unrelated structures

More Current State

- Tool for loading the fingerprints can read back the in-kernel list
- Tool for scanning all file systems and building an initial fingerprint list
- Separate sysctl facility for setting the veriexec mode of operation (aka strict level)
- Able to query supported fingerprint methods via sysctl

Operation

- kernel must have veriexec support compiled in
- Can select which fingerprint hash methods to support. Though removing fingerprint support does not affect kernel size much. More for compliance.
- Currently supports RMD160, SHA256, SHA384, SHA512, SHA1 and MD5
- Run the helper tool veriexecgen to scan the file systems and create a basic fingerprint file, edit the output to suit
- Load the fingerprints using veriexecctl
- Set the "strict" level using sysctl

The fingerprint file

- Has the following format:

```
path    type    fingerprint     flags
```

- Where:
  - path is the absolute path to the file
  - type is the fingerprint method
  - fingerprint is the actual fingerprint for the file
  - flags determine veriexec behaviour, some flags are direct, indirect, untrusted and file

What the flags mean

- **untrusted**, the file is on storage not under direct control of the kernel. Forces an evaluation of the fingerprint each access.
- **file**, a simple file that can be read. Things like shared libraries, configuration files
- **direct** a file that can be executed from the command line
- **indirect** an executable that cannot be executed from the command line but can be used as a script interpreter

- Multiple flags can be used in a comma separated list, e.g. a shell script would need both file and direct flags. There are convenience aliases, see the man page.

direct vs. indirect

- When implementing verifiedexec I noticed that the code path taken for the exec of a binary was different to that taken when a shell script was executed
- This difference in code path meant that a distinction could be made between an invocation from, say, the command line (i.e. direct) and the same binary being used as a shell interpreter (i.e. indirect)
- By making this distinction verifiedexec can permit a set of fingerprinted scripts to run but can block an attempt to run the script interpreter from the command line and thus prevent misuse of the interpreter
- It can also prevent the exec /bin/sh exploits, make /bin/sh an indirect execution along with all other shells. Copy a shell interpreter to an obfuscated name and make that the login shell for accounts.

Activation

- First step is to load the fingerprint list using veriexecctl
- Then set the sysctl kern.veriexec.strict to the desired level. The levels are:
    - **0** *learning mode* allows fingerprints to be loaded or updated. Is verbose about mismatches, incorrect file type access and other things that will cause problems later
    - **1** *IDS mode* denies access to files with mismatched fingerprints. Writes to fingerprinted files is allowed. Mismatched file type access is allowed (e.g. file vs direct). Along with other restrictions
    - **2** *IPS mode* all of the previous restrictions and also prevents writes to fingerprinted files, enforces file type access, plus more
    - **3** *Lockdown mode* all previous restrictions plus access to non-fingerprinted files is denied. Write access only allowed to file descriptors already open. Cannot create new files.
- It is expected most people would run at strict level 2 but use levels 0 and 1 to debug or validate fingerprint list

Future I

- The untrusted flag cannot protect a long running binary
- Attacker can overwrite pages in a binary on untrusted storage without detection
- The pager will bring in these pagers and the code will be executed
- Can build a set of page fingerprints in parallel with the fingerprint evaluation - if the latter is ok then the former can be used.
- Modify pager to check pages as they come in, terminate binary if there is a mismatch

Future II

- Digitally sign fingerprints
- Could mean fingerprints can be loaded whilst in operation
- Digitally signed binaries
- Both require in-kernel crypto support
- Could pre-populate a table with critical start up file fingerprints to narrow the start up hole