Verifiedexec: An Introduction

Brett Lymn

Introduction

The verifiedexec feature has been in the NetBSD kernel since late 2002 and is designed to provide a method of ensuring that the binary that is being run, and the underlying shared library files, have not been tampered with. It can detect the difference between an executable being directly executed by the shell and one being used as a shell interpreter in a shell script. This means different behaviours can be applied to an executables direct invocation vs one being used as a shell interpreter. Verifiedexec not only covers executables but also handles arbitrary files, this allows not only the shared libraries that executables may rely on to be protected but also gives the opportunity to protect files such as configuration files from tampering.

This paper will discuss the history of verifiedexec, where it is at in the current version of NetBSD and some possible features that could be added in the future.

Origins

Late in the last millenium there was a dramatic rise in the number of attacks using either trojan horses or root-kits reported on mailing lists such as Bugtraq. Whilst reading these reports I started to wonder why the kernel was being so cooperative in running whatever random executable that it was fed and if there could be a way of allowing known good executable to run but refuse to run one that have either been tampered with (a trojan horse) or one that had been installed to perform a malicious function (a root-kit). I felt that techniques using available at the time such as file attributes and mount permissions were difficult to correctly implement and a single error in the set up could leave the machine vulnerable or that conflicting requirements for some paths may require compromises that reduce the security of the configuration. There were programs such as tripwire that would scan the file system for modifications but these ran at userlevel and were vulnerable to such as shimming libraries to hide a root-kit by presenting unmodified files when a scan was run. With this in mind I started thinking about how the kernel could tell if an executable was one that it should run or not. I thought that if the kernel kept a list of fingerprints and then evaluated the fingerprint of an executable prior to execution then we could tell if the file has been tampered with. The major concern I had with this approach was that the performance would be poor since every time something was executed the fingerprint would be evaluated. Not only would this break demand paging due to the entire excutable file being read in every time but would also create an overhead simply by performing the fingerprinting operation. To avoid the performance impact I decided to cache the results of the fingerprint comparison. The problem with caching is that if someone manages to change the on-disk version of the executable after the fingerprint evaluation result has been cached then the kernel will happily run the modified file. This is not such a big problem for direct attached storage because the kernel has control over the access to the storage but for things like NFS or even SAN attached storage the control over write access can no longer be guaranteed. For the initial implementation of veriexec I took the view that to use verified exec the machine must only have direct attached storage. This restriction may be able to be lifted by a method described later. The NetBSD kernel sources were modified to implement the idea and measurements performed to evaluate the performance impact. Without caching the fingerprint evaluations a make of the NetBSD kernel took 1.7 times longer to finish, a very large performance hit. With caching the fingerprint evaluations there was a 5% increase in the time it took to make the kernel which is a considerable improvement over the non-caching performance. This code was subsequently committed to the NetBSD kernel tree and became available for general use.

Current State

Over the years the verifiedexec code has undergone some refinements and grown some capabilities that were not available in the original commit. The performance was improved by switching the in-kernel list of fingerprints from a simple list to a hash of lists which reduces the fingerprint lookup time. The tool used to load the fingerprints, veriexecctl, gained the capability to read back the in-kernel fingerprint list. The in-kernel code was modified slightly to be less instrusive on unrelated kernel structures. Also, a convenience script was developed that builds a list of fingerprints by scanning the file systems on the machine, this allows a user to quickly bootstrap a running configuration.

To run veriexec you need a kernel that has the support code compiled in, this requires a kernel config file with the following:

```
options FILEASSOC
pseudo-device veriexec
```

There are options to select what fingerprint methods are supported by the kernel, veriexec supports RMD160, SHA256, SHA384, SHA512, SHA1 and MD5. Taking out fingerprint methods has very little impact on the size of the kernel, the facility is there more for compliance. The removal of a fingerprint option from the kernel configuration ensures that a fingerprint method that may be considered insecure in some contexts (e.g. MD5 due to the ability to produce collisions) cannot be inadvertantly used.

To generate a file of fingerprints suitable for loading using `veriexecctl` the user can run the `veriexecgen` program. It is not mandatory to do so, one can generate the file completely from scratch if so desired. The fingerprint file has the format:

```
path type fingerprint flags
```

Where the fields mean the following:

path
: The absolute path to the file

type
: The fingerprinting algorithm used for the file

fingerprint
: The fingerprint for the file generated using a tool like `cksum`

flags
: A comma separated list of options including direct, indirect, file and untrusted. For the sake of brevity I won't detail what all these flags mean. Please refer to the veriexec man pages for the documentation on these flags.

Although, at this point, I would like to highlight the difference between the direct and indirect flags as the purpose of these two flags do not seem to be widely understood. When coding the veriexec modification I observed that path in the kernel that a binary executable took was different to that of a shell script execution. For a binary the exec is fairly straightforward, exec does some checks and then sets up the framework for the binary to start executing. For a shell script exec again performs some checks, finds that the candidate is actually a shell script so the script is examined for the shell interpreter which is executed and the contents of the script fed to the interpreter. This difference in path allows a binary being used as a shell interpreter to be treated differently to one that is being executed from an interactive shell. This is the purpose of the direct and indirect flags. If a file is marked as direct then execution of the file is permitted from, for example, an interactive shell. If a file is marked as indirect then execution of the file is denied from an interactive shell but the file may be used as an interpreter for a shell script. This means that the administrator could install, for example, perl and flag the perl binary as indirect to veriexec. This would allow the users to run a set of perl scripts that have fingerprints but veriexec would deny any attempts to invoke perl from the command line.

Once the fingerprint file has been generated and modified to suit local operational needs it can be loaded into the kernel using the `veriexecctl` command. With the fingerprints loaded veriexec can be put into operation by using `sysctl` to set the `kern.veriexec.strict` attribute. The `strict` attribute can set to one of four values. These values are in the order of most permissive to most restrictive are:

0
: Known as learning mode. Allows the modification of the in-kernel fingerprints, gives verbose information about fingerprint mismatches, incorrect access and other things that may cause problems at higher `strict` levels.

1
: Known as IDS mode. Access to files with mismatched fingerprints is denied. Writes to files in the fingerprint list are allowed, any cached fingerprint evaluation will be flushed in this case. The access type is not enforced which means that files with the flag of "file" are able to be executed,

assuming the fingerprint matches. Some other rules around raw disk access too.

2 Known as IPS mode. All the previous levels rules apply. In addition, all writes are prevented to fingerprinted files, execution of non-fingerprinted files is denied and raw disk access to media holding fingerprinted files is denied. The access type is enforced. Access to kernel memory is denied.

3 Known as Lockdown mode. All the previous levels rules apply. In addition, access to non-fingerprinted files is denied. Write access is only allowed on file descriptors opened before this mode was invoked. New files cannot be created. Raw disk access is denied.

For general use most people would run veriexec at `strict` level 2 but would use levels 0 and 1 to refine and debug the fingerprint list without causing an embarressing lockout by failing to include a critical binary in the fingerprint list.

Future

There are some features that could be added to veriexec to improve its usage in some applications. The first one is an improvement in the handling of the untrusted flag. This flag is meant to flag to veriexec that the storage that holds the file in question is not under direct control of the kernel and could possibly be modified without detection by the kernel. When the untrusted flag is used then veriexec will force an evaluation of the files fingerprint for every access in an attempt to detect a modified file. The obvious problem with doing this is the performance impact but the more subtle problem is that even evaluating the fingerprint every time the file is accessed will not provide protection in all cases. The reason for this is that fingerprints are not checked when parts of the file are paged in so if there is a long running binary sourced from untrusted storage an attacker could overwrite the binary and then flush any pages associated with the binary from memory. Flushing cached pages is a trivial operation if the attacker has a login to the machine, they can simply use mmap to do the job. If the attacker does not have direct access then resource starving the machine deliberately would accomplish the same thing. Once the pages are flushed the attacker can force the binary down a path of execution that causes the pager to pull in the modified part of the binary without detection. Though this scenario does sound unlikely I have developed a proof of concept of this attack and it does work as described. To protect against this attack the kernel needs to check each page as it is retrieved to ensure it has not been modified. The problem with precalculating the fingerprint for each page in a file is that it would make the fingerprint file large, unwieldy and difficult to maintain. A simpler solution is to leverage the fact that the entire file is already being read in to evaluate the fingerprint. If the fingerprint for each page is evaluated at the same time as the entire file is being checked then the page fingerprints can be built dynamically. If they fingerprint for the entire file matches then we know that the page fingerprints are valid and can be used. If the file fingerprint does not match then the page fingerprints are destroyed as they too are invalid. The pager code in the kernel can then be modified to validate the fingerprint of a page being brought in from storage where necessary. This solution was implemented and it does prevent the proof of concept exploit code from functioning. At the moment this code is not in the NetBSD kernel tree because concerns were raised by other developers about the way in which some of the underlying pager code was merged into a common function. These concerns need to be addressed before this facility can me made widely available.

Another possible extension is to pre-load fingerprints for critical start up files. At the moment veriexec will only come into effect once the rc scripts have loaded the fingerprint tables and raised the strict level. This leaves a window of opportunity which could be exploited as the machine boots, by having a fingerprint list built into the kernel critical files could be protected from boot.

I have also considered signing the fingerprint entries which would make unauthorised modification of the fingerprint list more difficult and, possibly, allow for fingerprints to be loaded at any time. The major stumbling block for the feature was the lack of a BSD licence crypto framework. This block has been removed by the importation of the netpgp library so it is now feasible to implement this feature. Another application that could use crypto in conjunction with veriexec is the signing of executables. This feature has been implemented by another NetBSD developer and works as expected, this feature is not yet in the official tree.

Conclusion

Certainly, verifiedexec is not for everyone. Using it on a general workstation would present many difficulties to to the inability to create and run new programs. It is intended more for situations where the machine configuration is more stable and assurance of the trusted computing base is desired. Applications could be such things as firewalls, DMZ infrastructure or even kiosk machines where the build of the machine does not change often and it is highly desirable to prevent an attacker setting up undesirable applications such as root kits.