Verified Exec - Extending the Security Perimeter

Brett Lymn
Copyright 2000-2004. All rights reserved.
December 12, 2003

SYNOPSIS

At the Australian Unix Users Group (AUUG) 2000 conference a paper was presented by the author[1] for an idea which, at the time, was called Signed Exec. This idea had been implemented as a proof of concept on a NetBSD kernel and was shown to function as expected. After the presentation of that paper work continued on Signed Exec to address some of the areas that were highlighted as future work in the paper. Near the end of 2002 the name Signed Exec was changed to Verified Exec and the code was committed to the NetBSD-current kernel tree. After the code was committed, another NetBSD developer suggested that with some extensions Verified Exec could be used to secure files on untrusted media, something that was not considered in the original implementation. A method of verifying files on untrusted media was proposed and implemented. By modifying the kernel paging routines and verifying the individual pages as they are returned from storage provides a method of ensuring files on that storage have not been tampered with since they were first checked. This paper will detail how this was done.

PRIOR IMPLEMENTATION

The original proof of concept for Verified Exec was called Signed Exec. This was the subject of a paper at the AUUG 2000 conference which detailed the work done at that time. Signed Exec worked by modifying the code path for exec in the kernel to MD5 fingerprint the exec candidate file and compare that file with an in kernel list of fingerprints. By doing this fingerprint verification any modification to the executable file could be detected and, optionally, prevented from running. Significant speed improvements were realised by avoiding calculating the fingerprint of the executable every time it was run. This was done by caching the results of the comparison of the evaluated and in-kernel

fingerprints in the v-node structure associated with the executable file. Since the NetBSD kernel tends to keep the v-node structure for a file in memory as long as possible this structure provided a ready made cache for the fingerprint comparison. Without caching the fingerprint comparison a test run of building a NetBSD GENERIC kernel took 1.7 times longer than the same build performed without Signed Exec. With caching the fingerprint the impact of Signed Exec on building a NetBSD GENERIC kernel could not be measured.

At the time the AUUG 2000 paper was presented there were some areas of Signed Exec that required more work, one major area perceived was the fact that Signed Exec only adequately covered statically linked binaries. As it was, an attacker could modify a shared library and Signed Exec would fail to detect the modified library when running a dynamically linked binary. Another area that required work was the ability to add different fingerprint methods, the implementation the AUUG paper was based on was hardwired to use the MD5 fingerprinting and it was very difficult to add a different fingerprinting method to the framework without extensive modifications.

THE VERSION IN NETBSD-CURRENT

Some time after the AUUG 2000 paper was presented, work resumed on Signed Exec to address some of the areas of further work highlighted. Firstly, the code was rearranged. All the Signed Exec portions of code were removed from where they had been and put into a separate module, the exec path fingerprint check was reduced to a single function call into the Signed Exec code. The contents of the Signed Exec module were rewritten to include a fingerprint type and the function that acted as the external interface to Signed Exec was used as a fingerprint type dependent selector. This code rearrangement meant that the task of adding a new fingerprint type is much simplified, only small changes to the kernel files need to be made to the existing to support the new fingerprint type.

After the code rearrangement, the next task undertaken was to develop a method of protecting dynamically linked binaries. Protecting a statically linked binary was simple, the file the kernel loaded and ran is self contained so fingerprinting that single file is enough to protect the binary. For a dynamically linked executable, many files containing the shared library code are opened and their contents mapped into the memory space of the running executable so simply verifying the fingerprint of the executable is not sufficient to prevent tampering. Originally, it was thought that it would be required to find a method of fingerprinting mmap'ed regions to protect the shared libraries. After some investigation it was found that a far simpler solution lay in verifying the fingerprint of a file during the file open operation, performing the fingerprint check on open turns out to have other benefits which will be discussed later. The VFS function vn_open() was modified to verify the fingerprint of a file on open.

The problem with doing the fingerprint verification on open is that not all files are fingerprinted nor is it possible to fingerprint all files. Consider the logistics of fingerprinting a file open for write. Due to this, only read only files are fingerprinted. The vn_open() code was modified to search two lists of fingerprints for the file being opened, firstly a list of fingerprints for files and secondly the list of fingerprints for executables. Two lists were used to try and reduce the size of the fingerprint list for performance reasons, in normal operation it is expected that only one list will be searched, searching both lists should be a rare occurrence. If these list searches fail then the file is not specially protected and the open proceeds. If the file was found on one of the lists then the mode of the open is checked and an attempt to open a fingerprinted file for write is rejected. This test effectively means that any file that has a fingerprint is automatically made read only. Then the fingerprint of the file is performed and the results compared, if the fingerprint matches vn_open() runs to completion, if the fingerprint comparison does not match then the error EPERM is returned. By performing the fingerprint in vn_open() the aim of protecting shared libraries from tampering was achieved, an unexpected benefit of fingerprinting files in this manner was that any file on the system could have a fingerprint associated with it. This meant that system critical configuration files could be fingerprinted and would be verified as being unmodified before being used.

Once the issues with protecting dynamically linked binaries and the code rearrangement had been addressed the code for Signed Exec was committed to the NetBSD-current source tree. During the commit it was decided to change the name from Signed Exec to Verified Exec. The reasoning being that nothing was really being signed by this method but what was being done was more a verification of integrity. Verified Exec is now available as an option to all users of the NetBSD-current source.

VERIFYING AT THE PAGE LEVEL

Once Verified Exec was committed to NetBSD-current some users were dissatisfied with the assumption of the secure perimeter. A secure perimeter is the boundary inside which all the components are trusted and can be relied upon to build a secure system. As it was, the secure perimeter of a Verified Exec system was, effectively, the case of the machine. One of the basic assumptions of Verified Exec was that the kernel had full control over the hardware such that nothing else could tamper with the underlying storage to modify data on that storage without the kernel knowing about it. This assumption was a critical requirement of Verified Exec because it allowed the kernel to cache the results of the fingerprint comparison and rely on that comparison to be valid at a later point in time, if something else modified the fingerprinted file without the kernel knowing about it then the next time that file was read or executed the kernel would blindly allow the modified file to be used because the cached status

indicated the file was good. This meant that Verified Exec could not trusted to detect modified files if those files were stored on a SAN (Storage Area Network) or on a NFS mount, in both cases other machines could modify the files on the storage without the Verified Exec machine knowing about the change. One strategy that could be taken, and has been done by TrojanProof[2], is to detect the type of device the files reside on, if the device is not a local storage device then the caching of the fingerprint status is disabled, effectively forcing the fingerprint to be verified every time the file is referenced. This approach sounds reasonable but it has at least one flaw. The detection of modification of the file will only happen when the file is first executed or read. Consider the situation where a long running program is loaded from storage outside the secure perimeter, not all the pages that comprise the program may be in memory at once. The non-resident pages will be brought in from storage if the execution path requires that page by the virtual memory subsystem. This is called demand paging, pages of the executable are paged in on demand. Also, if the system is under memory pressure then unused pages of an executable may be discarded by the underlying virtual memory system. Note that the executable is still running resident in memory but pages for that executable may need to be read from storage. Now consider the implications of an attacker replacing the binary image with a trojaned copy on the untrusted storage, the Verified Exec kernel has no knowledge the replacement has occurred. If that trojaned copy were to be run from the untrusted store then the tampering would be evident as the fingerprint would not match, assuming the fingerprint was verified every time, but for a long running executable there is no checking done because the executable is still running. The crux here is that if the binary is still running the virtual memory system may bring pages into memory unchecked from the untrusted storage if those pages had been discarded due to, for instance, memory demands or deliberate invalidation of the pages. An attack scenario can be envisioned where the attacker replaces the binary on the untrusted storage with a trojaned one that has a trojan on an uncommon execution path, performs some operations that increase the memory demand on the Verified Exec machine and then triggers the trojaned code by forcing the long running binary down the correct execution path. In this situation the binary fingerprint verification, even if it happens every time the binary is executed, will be bypassed because the virtual memory pager is simply replacing pages for an executable that has already been verified.

Initially, after the release of the Verified Exec code, the problem of storing executables or files on untrusted storage such as NFS or SANs was avoided by simply not doing it. At the time of the release it was thought that maintaining a list of fingerprints at the page level for all the required files was too cumbersome and unwieldy. Also, generating the fingerprints in such a system would have been difficult to do correctly. After some thought and discussion a system was proposed that used the current Verified Exec fingerprinting technique but could also operate at the virtual memory page level. Given that the whole file was being read during the fingerprint verification process it was seen that this was an

ideal time to generate the fingerprints of the memory pages. It was reasoned that if the fingerprint of the full file matched the expected value then there was a good level of confidence that the fingerprints of the individual pages were valid for that executable. By building the array of page fingerprints in parallel with the full file fingerprint any chance of a race condition existing between the full file fingerprint and the building of the array of page fingerprints would be obviated.

The NetBSD kernel was modified to incorporate the changes required to perform the page level fingerprinting. Since the original Verified Exec framework was still to be used there were only minor changes that needed to be made, some extra functions were added to the Verified Exec kernel code to perform the page level fingerprinting. The v-node structure had a pointer to a dynamically allocated array of page fingerprints added and the pager code was modified to include a call to the Verified Exec page fingerprint verification function if required, the pager code was adjusted so that failure of the comparison of the page fingerprint with the expected fingerprint would return an error to the calling subsystem. This error would result in application requesting the page terminating. This action could result in a denial of service but no compromise of the Verified Exec machine would occur.

The Verified Exec configuration file had an "untrusted" option added to the syntax so that executables and files that were stored on untrusted storage could be flagged and page level fingerprint verification performed on them, this was done to allow the administrator control over what files, if any, will have page level fingerprinting performed on them. Fine grain control over the files that required page level fingerprinting was deemed necessary due to the increased kernel resources required to evaluate and store the page fingerprints.

At the time of writing this paper, the modifications to the NetBSD kernel source has been completed and the page level fingerprints have been shown to work as expected. No measurements as to the impact of the page level fingerprints have been performed on a running system but this impact is expected to be low in a normally running system. It is expected that adding page fingerprints will make the startup of an executable slower due to the added overhead of fingerprinting not only the whole file but the pages in that file as well but after this initial start up the operation should be unaffected. It is expected machines with less memory will be more affected as they would be more likely to require more pager activity in normal operation which will magnify the effect of performing the memory page fingerprint.

CONCLUSION

The Verified Exec feature give an administrator a valuable tool to use in helping to lock down a machine that needs to be resistant to tampering in a hostile

environment. It provides a method of automatically monitoring and verifying the files on a systems against a known good state. If the security boundary of the machine resides entirely within the case of the machine then Verified Exec can be sufficient to prevent tampering but if the security boundary extends to storage that is not under the direct control of the kernel then there is scope for a file to be tampered with without detection. By adding fingerprinting of the individual pages of a file the virtual memory system can ensure that any pages that are brought in from storage by the VM pager have not been tampered with. This means that files or executables may be stored on servers that are not secured but still have confidence that the files or executable retrieved from that server is a true and correct copy.

REFERENCES

[1] AUUG 2000 Conference paper, this can be found at http://users.on.net/blymn/veriexec/

[2] TrojanProof uses a similar technique to that presented in this paper, in fact the caching technique described in [1] by the author has been included in a later version of TrojanProof. More information can be found at http://www.trojanproof.org/