Executable Integrity Verification

Abstract

Determining if a given executable has been trojaned is a tedious task. It is beyond the capabilities of the average end user and even many network administrators. Furthermore trojaning application is becoming a growing problem, and a favorite mechanism whereby attackers deliver malicious software to target machines. The ability to have an automatic process that verifies the integrity of any application would be of immense usefulness to any operating system, computer network, or computer user. This process disclosed in this publication provides such a mechanism that will verify the integrity of any given application. This mechanism is automatic and requires little or no input from the user. Products like Tripwire give a partial solution by using CRC checks on key system files. But they do not go far enough and they are not automatic and transparent to the end user.

Background

Patent 5,956,481: The purpose of this invention is to inspect a file upon opening so that if it is likely the file contains a virus, the user is notified. For example it warns the user that the file contains macros, which potentially could contain macro viruses. This invention is currently embodied in Microsoft Windows which notifies users if a file contains Macros.

Patent 7,603,614: This invention attempts to determine if a given executable or file is a Trojan horse by comparing the Cyclic Redundancy Check (CRC) values of known Trojan horses, to the suspected application.

Patent Application 10/231,557: This invention uses a weighted scoring system to determine if a suspected file or executable is likely to be a Trojan horse.

Patent 6,687,836: This invention deals with using hash algorithms so a user can verify whether or not he or she has correctly entered their password.

Runtime Execution Monitoring to prevent Malicious code: This paper (http://palms.ee.princeton.edu/PALMSopen/fiskiran04runtime.pdf) describes a process wherein each block of a program is hashed and the flow of the program is monitored in real time as it is executing. this is designed to prevent buffer over flow attacks.

eXecute Only Memory (XOM): In this invention (http://www.stanford.edu/class/cs259/slides/07-xom.pdf) all applications are distributed in encrypted form, and decrypted at run time. This is meant to prevent tampering with the application or illegal copying of the application.

Tripwire: In this product (http://www.tripwire.com/) checksums are created of certain key system files and are checked periodically to see if there are changes. Here is the description from their website “Tripwire Enterprise captures a baseline of server file systems, desktop file systems, directory servers, databases, middleware applications and network device configurations in a known good state. Ongoing integrity checks then compare the current states against these baselines to detect changes. While doing this it collects information essential to the reconciliation of detected changes, ensuring they are authorized and intended changes. Tripwire Enterprise can crosscheck detected changes with either defined policies (policy-based filtering), documented change tickets in a CCM system or a list of approved changes, automatically generated lists created by patch management and software provisioning tools, and against additional ChangeIQ capabilities.”

Tripwire Open Source: This product is a host based IDS. It uses processes similar to what the commercial Tripwire product does and was in fact based on their product.

Code Signing: This the process of digitally signing executables and scripts to confirm the software author and guarantee that the code has not been altered or corrupted since it was signed by use of a cryptographic hash. This invention depends on the vendor of the software product having signed the product prior to distribution, and having used a certificate from a trusted certificate authority. While similar in some respects to the current invention, the entire purpose of the current invention is to overcome weaknesses in inventions like code signing.

Discussion

A Trojan horse is an application that appears to be a legitimate application, but contains a malicious payload. That payload could be a virus, spyware, rootkit, logic bomb, or any type of malware. Usually the malicious software is actually wrapped with a legitimate file. For example one might wrap a spyware program to an innocuous file such as a game.

There are many methodologies to accomplish this wrapping process. There are tools such as EliteWrap (http://homepage.ntlworld.com/chawmp/elitewrap/) which will accomplish the task for you. Another methodology is actually built into the NTFS file system used by Windows operating systems. That method takes advantage of Alternate Data Streams (ADS). Alternate Data Systems is a methodology within NTFS that allows one to tie one file to another (http://support.microsoft.com/kb/105763). The user then will only see one of the files in Windows Explorer or when listing files from the command line. However when that file is executed, the hidden file will also be executed. This is a well known vulnerability in the security community (http://www.windowsecurity.com/articles/Alternate_Data_Streams.html).

Whatever the specific methodology for tying malicious software to an innocuous program, when the process is done that program is said to have been ‘Trojaned’.

Current methodologies for determining if a given file or executable has been Trojaned are frequently ineffective. The current methods depend solely on looking for signatures of known Trojans (patent 7,603,614), or simply if the file has features that might contain a Trojan (patent 5,956,481 and patent Application 10/231,557:).

The EC Council (the vendor sponsoring the Certified Ethical Hacker certification test) recommends that if one suspects a given executable is Trojaned, the user should compare an MD5 hash of the executable with the MD5 hash provided on the installation media. This method requires:
1. The user must suspect an executable has been Trojaned.
2. The user must then elect to perform a test of that executable.
3. The installation media must have a hash of the original executable on it.
4. The user must have a mechanism for hashing the current executable.

This methodology while effective is extremely cumbersome. It also is dependent upon both user knowledge, and upon the executables vendor having provided a hash of the executable on the installation media. Furthermore this methodology is only implemented if and when a user suspects a particular executable has been Trojaned. It is therefore obvious that Trojaned executables would frequently be missed.

What this current process (Executable Integrity Verification) proposes to do is to take a similar approach, however to make the process
1. Fully automated.
2. Transparent to the user.
3. Not dependent upon the software vendors providing hashes.
4. Applicable to all executables on a given system.
5. Applicable to multiple environments (i.e. the home/individual computer and the organizational network computer).

In this process, Executable Integrity Verification, each new program installed upon the system would have a hash value computed and stored. In most systems, including Windows, only administrators can install software, therefore this hashing process would also be tied to administrative privileges. The process of computing a hash upon installation is shown in figure 1.1.

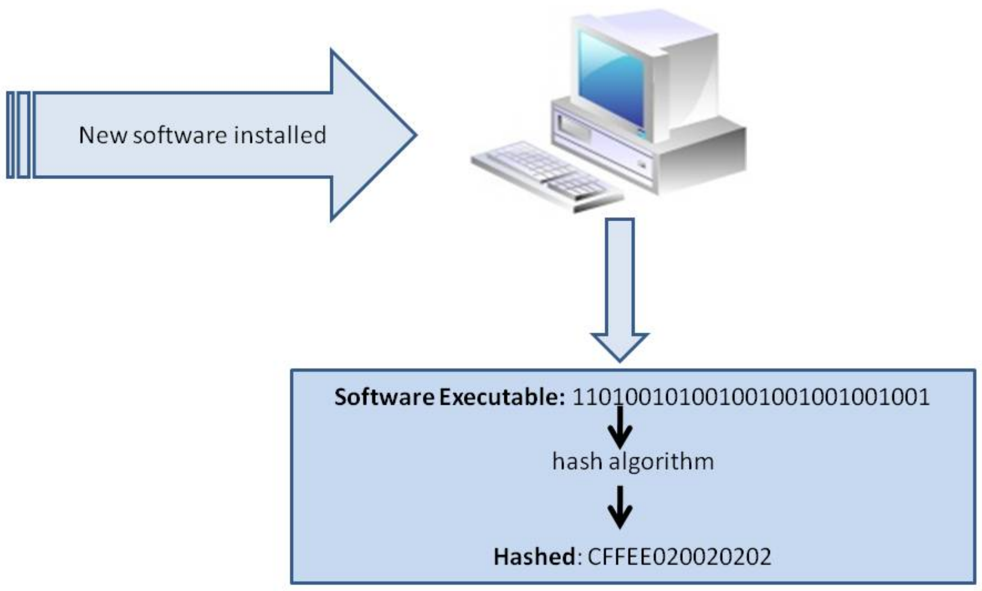


Figure 1.1 hashing executables.

The hash of each executable would then be stored in a secure location. In one embodiment of the process that would be a secure server on the organizational network. This server is called a *hash value server*. In this embodiment, executable hashes would be stored in much the same way digital certificates are stored. This is shown in figure 1.2.

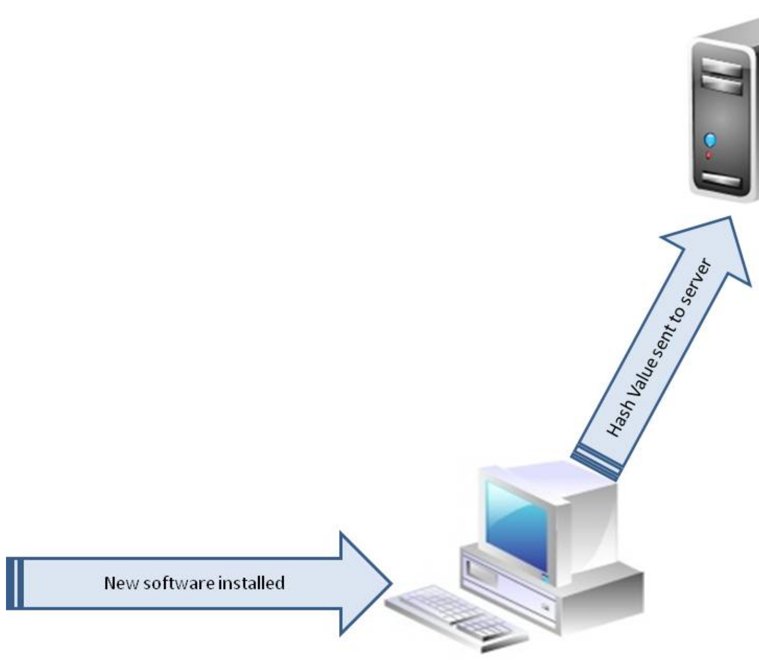


Figure 1.2 Storing executable hash values.

Then when any executable was launched, a query would be issued to the hash value server sending that server the hash value of the current executable. The server would then compare that hash value to the stored value. If the hash values matched the server would respond authorizing the requesting computer to execute the application. If the hash values did not match then the server would respond warning the requesting computer that the applications integrity was compromised. This is shown in figure 1.3.

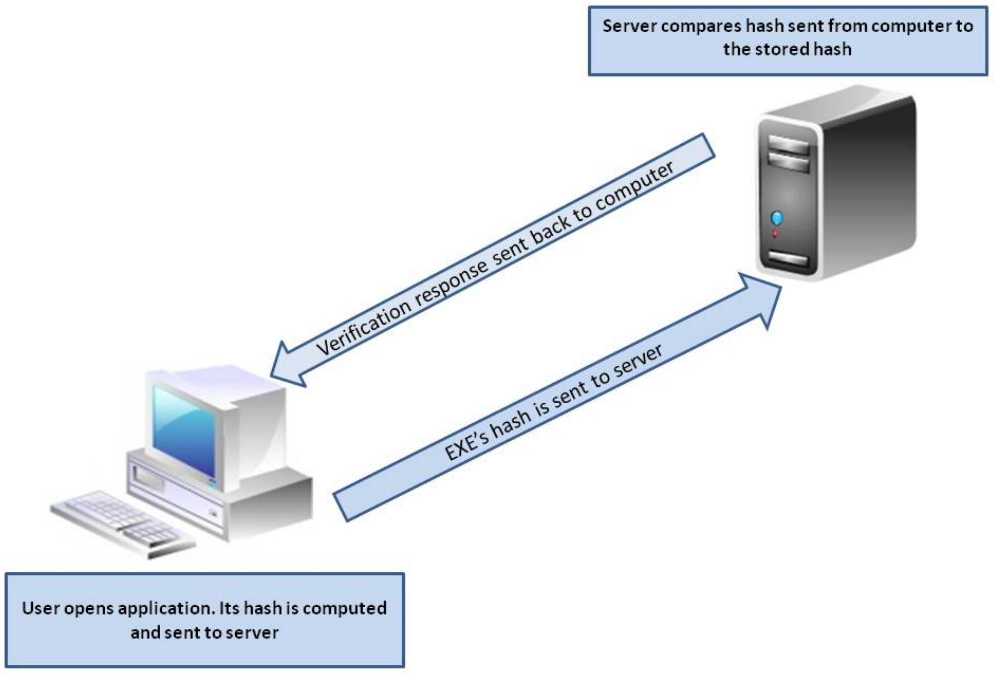
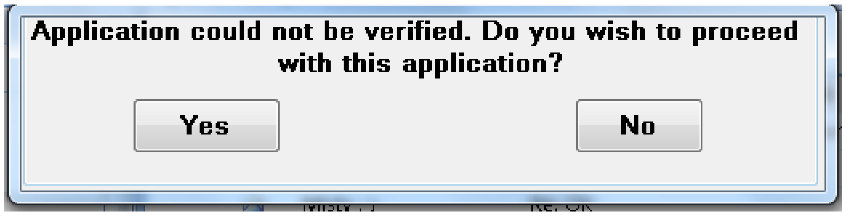

Server compares hash sent from computer to the stored hash
Verification response sent back to computer
EXE's hash is sent to server
User opens application. Its hash is computed and sent to server

Figure 1.3 Hash Verification

In cases where no hash value was found on the hash value server, the server could either respond denying the program permission to run or in another embodiment of the process, warn the user that the integrity of the program could not be verified. In the second embodiment the user could then decide to proceed with execution or not. This warning is shown in figure 1.4.



Application could not be verified. Do you wish to proceed with this application?

Figure 1.4 Application Warning

This embodiment utilizing a hash value server, would be effective in an organizational network setting but not as effective in a home setting or any situation in which an individual computer was operating independent of a network. In such settings another embodiment of the process could be used. In this embodiment of the process, immediately following the installation of any new executable on the users computer, a hash would be calculated for that executable. That hash could be stored in a secure location on the user's computer (rather than on a hash value server), such as a system folder only accessible to users with administrator/root privileges. This is shown in figure 1.5.

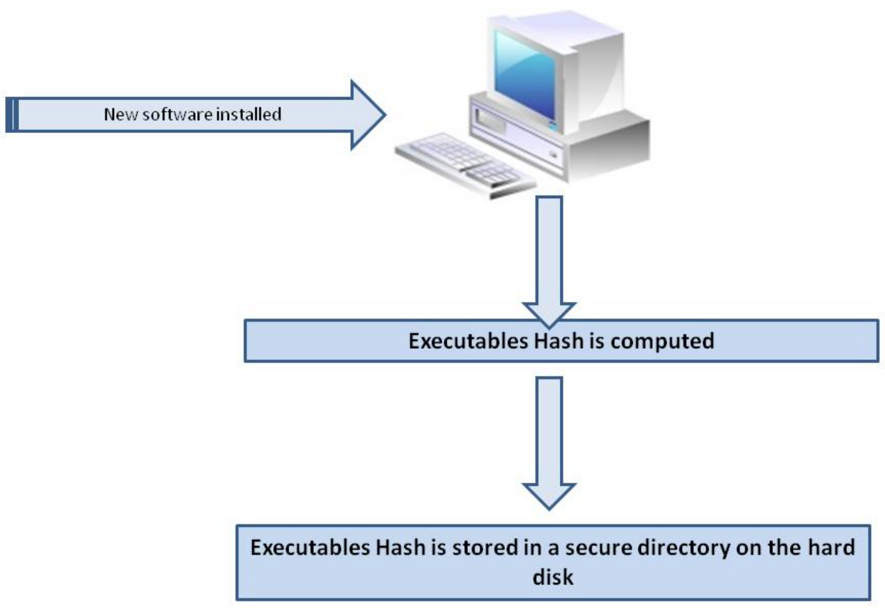

New software installed
Executables Hash is computed
Executables Hash is stored in a secure directory on the hard disk

Figure 1.5 Hash stored in secure folder.

In yet another embodiment of this process the hashes stored on the users' computer would be encrypted. This is shown in figure 1.6.

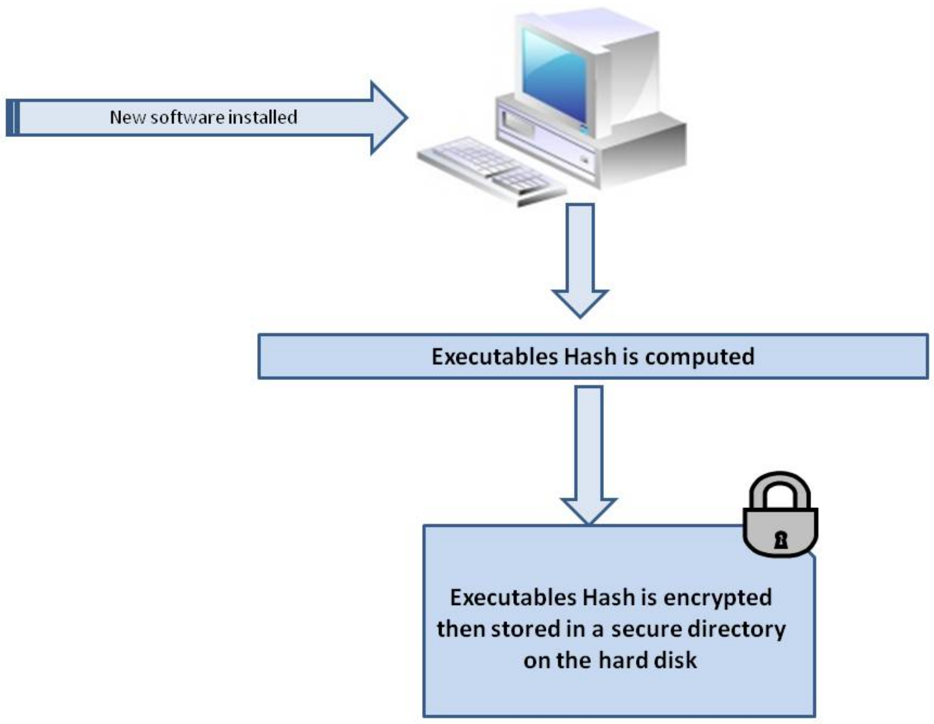

New software installed
Executables Hash is computed
Executables Hash is encrypted then stored in a secure directory on the hard disk

Figure 1.6 Encrypted Storage of the Hash values

Note: it is highly recommended that the hash values be encrypted. Windows already offers an additional mechanism for encrypting the hashed passwords. This mechanism is called Syskey.   If the hash values are stored locally and are not encrypted, there is the possibility of an attacker trojaning an executable then putting that trojaned executables hash into the hash value repository.

In another, more secure, embodiment of the process, after installing a new program the hash for that program would be calculated and the user would be prompted to insert some removable media (CD, DVD, USB, etc.). The hash value would be stored on that removable media. Applications could only be executed when the removable media was present so that hash values could be compared.

This process is not limited to a specific hashing algorithm. One could use SHA-1, SHA1, MD4, MD5 or any hashing algorithm. The purpose of the hash is simply to verify the integrity of the target application.

This process may seem, on the surface to be similar to code signing. However there are several important differences including:

1. With code signing, the software vendor must sign the code. If it was not signed, then the operating system has no way of verifying the software. With the current process (Executable Integrity Verification), any software that is installed on the computer, even if the software vendor did not sign it, will have a hash calculated at installation. Therefore this process works with any software.
2. With code signing the purpose is to verify that software being downloaded from the internet is what it purports to be. Once the product is installed, it is not then checked at each execution. With the current process (Executable Integrity Verification) each time software is executed, it is checked. This specifically will prevent techniques like using Alternate Data Streams that attach other files to existing files. Code signing won't do that.
3. Code signing is dependent upon third party digital signatures. With this current process (Executable Integrity Verification) the operating system creates its own hash for each executable aat install time. So there is no reliance on third parties, nor even a need to be connected to the internet.

References

http://msdn.microsoft.com/en-us/library/ms537361(VS.85).aspx
http://www.verisign.com/code-signing/index.html
http://en.wikipedia.org/wiki/Code_signing

Note: while this process uses hashing, hashing is not the cornerstone of the process. Many processs utilize hashing as part of their process.