A Lightweight Binary Authentication System for Windows

Felix Halim, Rajiv Ramnath, Sufatrio, Yongzheng Wu, Roland H.C. Yap

Abstract The problem of malware is greatly reduced if we can ensure that only software from trusted providers is executed. In this paper, we have built a prototype system on Windows which performs authentication of all binaries in Windows to ensure that only trusted software is executed and from the correct path. Binaries on Windows are made more complex because there are many kinds of binaries besides executables, e.g. DLLs, drivers, ActiveX controls, etc. We combine this with a simple software ID scheme for software management and vulnerability assessment which leverages on trusted infrastructure such as DNS and Certificate Authorities. Our prototype is lightweight and does not need to rely on PKI infrastructure; it does however take advantage of binaries with existing digital signatures. We provide a detailed security analysis of our authentication scheme. We demonstrate that our prototype has low overhead, around 2%, even when all binary code is authenticated.

1 Introduction

Malware such as viruses, trojan horses, worms, remote attacks, are a critical security threat today. A successful malware attack usually also modifies the environment (e.g. file system) of the compromised host. Many of the system security problems such as malware stem from the fact that untrusted code is executed on the system. We can mitigate many of these problems by ensuring that code which is executed only comes from trusted software providers/vendors and the code is executed in the correct context. In this paper, we show that this can be efficiently achieved even

Felix Halim, Yongzheng Wu, Roland H.C. Yap
School of Computing, National University of Singapore, e-mail: {halim,wuyongzh,ryap}@comp.nus.edu.sg

Rajiv Ramnath, Sufatrio
Temasek Laboratories, National University of Singapore, e-mail: {tslrr,tslsufat@nus.edu.sg}

Please use the following format when citing this chapter:

Halim, F., Ramnath, R., Sufatrio, Wu, Y. and Yap, R.H.C., 2008, in IFIP International Federation for Information Processing, Volume 263; *Trust Management II*; Yücel Karabulut, John Mitchell, Peter Herrmann, Christian Damsgaard Jensen; (Boston: Springer), pp. 295–310.

on complex operating systems such as Windows. Our system provides two guarantees: (i) we only allow the execution of binaries (in the rest of the paper, we refer to any executable code stored in the file system as a *binary*) whose contents are already known and trusted — we call this *authenticating binary integrity*; and (ii) as binaries are kept in files, the pathname of the file must match its content — we call this *authenticating binary location*. Binary integrity authentication ensures that the binary has not been tampered with, e.g. `cmd.exe` is not a trojan. Binary location authentication ensures that we are executing the correct executable content. A following extreme example illustrates location authentication. Suppose the binary integrity of the shell and the file system format executables are verified. If an attacker swaps their pathnames, then running a shell would cause the file system to be formatted. In this paper, we refer to *binary authentication* to mean when both the binary's data integrity and location are verified.

Most operating systems can prevent execution of code on the stack due to buffer overflow, e.g. NX protection. Combining stack protection with binary authentication makes the remaining avenues for attack smaller and more difficult. Binary authentication is also beneficial because it is even more important for the operating system to be protected against malicious drivers and the loading of malware into the kernel.

Most work on binary integrity authentication is on Unix/Linux [1, 2, 3]. However, the problem of malware is more acute in Windows. There are also many types of executable code, e.g. executables (.exe), dynamic linked libraries (.dll), ActiveX controls, control panel applets, and drivers. In this paper, we focus on mandatory binary authentication of all forms of executables in Windows. Binaries which fail authentication cannot be loaded, thus, cannot be executed. We argue that binary authentication together with execute protection of memory regions (e.g. Windows data execution prevention) provides protection against most of the malware on Windows.

A binary authentication needs to be flexible to operate under different scenarios. Our prototype signs binaries using a HMAC [4] which is more lightweight than having to rely on PKI infrastructure, although it can also make use of it. The authentication scheme additionally allows for other security benefits. Not only is it important to authenticate software on a system but one also needs to deal with the maintenance of the software over time. Nowadays, the number of discovered vulnerabilities grows rapidly [5]. This means that binaries on a system (even if they are authenticated) may be vulnerable. This leads to a vulnerability management and patching problems. We propose a simple software ID system leveraging on the binary authentication infrastructure and existing infrastructures such as DNS and certificae authorities to handle this problem.

Windows has the Authenticode mechanism [6]. In Windows XP version and earlier, it alerts the users of the results of signature verification under a few situations. However, it is not mandatory, and can be bypassed. The Windows Vista UAC mechanism makes use of signed binaries but it only deals with EXE binaries. It is also limited to privilege escalation situations. One common drawback of existing Windows mechanisms is that they do not authenticate the binary location. Moreover, requiring PKI infrastructure and certificates, we believe, is too heavy for a general purpose mechanism.

The main contribution of this paper is that we believe that it provides the first comprehensive infrastructure for trusted binaries for Windows. This is significant given that much of the problems of security on Windows stems from inability to distinguish between trusted and untrusted software. It provides mandatory authentication for the full range of binaries under Windows, and goes beyond authenticated code in XP and Vista. We also protect driver loading which gives increased kernel protection. Our scheme provides mandatory driver authentication which 32-bit Windows does not, and can be integrated with more flexible policies which 64-bit Windows does not support. We also analyze the security of our system. Our benchmarking shows that the overhead of comprehensive binary authentication can be quite low, around 2%, with a caching strategy.

2 Windows Issues

We discuss below the complexities and special problems of Windows which make it more difficult to implement binary authentication than in other operating systems such as Unix. Windows NT (Server 2000, XP, Server 2003, Vista) is a microkernel-like operating system. Programs are usually written for the Win32 API but these are decomposed into microkernel operations. However, Windows is closed source — only the Win32 API is documented and not the microkernel API. Our prototype makes use of both the documented and undocumented kernel infrastructure. However, it is not possible to make any guarantees on the completeness of the security mechanisms (which would also be a challenge even if Windows was open source). Some of the specific issues in Windows which we deal with are:

- **Proliferation of Binary Types:** It is not sufficient to ensure the integrity of `EXE` files. In Windows, binaries can have any file name extension, or even no extension. Some of the most common extensions include `EXE` (regular executables), `DLL` (dynamic linked libraries), `OCX` (ActiveX controls), `SYS` (drivers) and `CPL` (control panel applets). Unlike Unix, binaries cannot be distinguised by an execution flag. Thus, without reading its contents, it is not possible to distinguish a binary from any other file.
- **Complex Process Execution:** A process is created using `CreateProcess()` which is a Win32 library function. However, this is not a system call since Windows is a microkernel, and in reality this is broken up at the native API into: `NTCreateFile()`, `NTCreateSection()`, `NTMapViewOfSection()`, `NTCreateProcess()`,`NTCreateThread()`. Notice that `NTCreateProcess()` at the microkernel level performs only a small part of what is needed to run a process. Due to this, it is more complex to incorporate mandatory authentication in Windows.
- **DLL loading:** To load a DLL, a process usually uses the Win32 API, `LoadLibrary()`. However, this is broken up in a similar way to process execution above.
- **Execute Permissions:** Many code signing systems, particularly those on Linux [1, 3], implement binary loading by examining the execute permission bit in the

access mode of file open system-call. The same mechanism, however, does not work in Windows. Windows programs often set their file modes in a more permissive manner. Simply denying a file opening with execute mode set when its authentication fails, will cause many programs to fail which are otherwise correct on Windows. Instead, we need to properly intercept the right API(s) with correctly intended operation semantics to respect Windows behavior.

Compared to other open platforms, Windows potentially also makes the issue of locating vulnerable software components more complicated. A great deal of binaries created by Microsoft contain an internal file version, which is stored as the file's meta-data. The Windows update process does not indicate to the user which files are modified. Moreover, meta data of the modified file might still be kept the same. Thus, it is difficult to keep track of files changes in Windows. More precisely, one cannot ensure whether a version of a program P_i remain vulnerable to an attack A. It is rather difficult for a typical administrator who examines vulnerability information from public advisories to trace through the system and pinpoint the exact affected components. Our software naming scheme, associates binaries with their version and simplifies software vulnerability management.

3 Related Work

Tripwire [9] is one of the first to do file integrity protection but is limited as it in user-mode program and checks file integrity off-line. It does not provide any mandatory form of integrity checking and there are many known attacks such as: file modification in between authentication times, and attacks on system daemons (e.g. cron and sendmail) and system files that it depends on [10, 11].

There a number of kernel level binary level authentication implementations. These are mainly for Unix such as DigSig [1], Trojanproof [2] and SignedExec [3], which modify the Unix kernel to verify the executable's digital signature before program execution. DigSig and SignedExec embed signatures within the the elf binaries. For efficiency, DigSig employs a caching mechanism to avoid checking binaries which have been verified already. The mechanism is similar to ours here but we need to handle the problems of Windows. It appears that DigSig provides binary integrity authentication but not binary location authentication.[1] In this paper, we examine the implementation issues and tradeoffs for Windows which is more complex and difficult than in Unix.

Authenticode [6] is Microsoft infrastructure for digitally signing binaries. In Windows versions prior to Vista, such XP with SP2, it is used as follows:

1. During ActiveX installation: Internet Explorer uses Authenticode to examine the ActiveX plugin and shows a prompt which contains the publisher's information including the result of the signature check.

[1] Mechanisms based solely on signatures embedded in the binaries do not have sufficient information for binary location authentication.

2. A user downloads a file using Internet Explorer: If this file is executed using the Windows Explorer shell, a prompt is displayed giving the signed the publisher's information. Internet Explorer uses an NTFS feature called Alternate Data Streams to embed the Internet zone information –in this case, the Internet– into the file. The Windows Explorer shell detects the zone information and displays the prompt. This mechanism is not mandatory and relies on the use of zone-aware programs, the browser and GUI shell cooperating with each other. Thus, it can be bypassed.

Since Authenticode runs in user space, it can be bypassed in a number of ways, e.g. from the command shell. It is also limited to files downloaded using Internet Explorer. Only the EXE binary is examined by Authenticode, but DLLs are ignored. One possible attack is then to put malware into a DLL and then execute it, e.g. with `rundll32.exe`. Furthermore, Authenticode relies heavily on digital certificates. Checking Certificate Revocation Lists (CRL) may add extra delay including timeouts due to the need to contact CA. In some cases, this causes significant slowdown.

The latest Windows Vista improves on signed checking because User Account Control (UAC) can be configured for mandatory checking of signed executables. However, this is quite limited since the UAC mechanism only kicks in when a process requests privileged elevation, and for certain operations on protected resources. UAC is not user friendly since there is a need for constant interactive user approval. Vista does not seem to prevent the loading of unsigned DLLs and other non EXE binaries. The 32-bit versions of Windows (including Vista) do not checked whether drivers are signed. However, the 64 bit versions (XP, Server 2003 and Vista) require all drivers to be signed (this may be too strict and restrict hardware choices).

The closest work on binary authentication in Windows is the Emu system in by Schmid et al. [13]. They intercept process creation by intercepting the NtCreateProcess system call. It is unclear whether they are able authenticate all binary code since trapping at NtCreateProcess is not sufficient to deal with DLLs. No performance benchmarks are given, so it is unclear how if their system is efficient.

4 Binary Authentication and Software IDs

We want a lightweight binary authentication scheme which can work under many settings without too much reliance on other infrastructure. Furthermore, it should help in the management of binaries, and incurs low overhead. Management of binaries includes determining which binaries should be authentic, dealing with issues arising from disclosed vulnerabilities, and software patching.

4.1 Software ID Scheme

We complement binary authentication with a software ID scheme meant to simplify binary management issues. The idea is that a *software ID* associates a unique string to a particular binary of a software product. The software ID should come either from the software developer or alternatively be assigned by the system administrator. The key to ensuring unique software_ID, even among different software developers, lies on the standardized format of the ID. We can define software_ID as follows:

$$\text{Software\_ID} ::= \langle\ \text{opcode\_tag} \,||\, \text{vendor\_ID} \,||\, \text{product\_ID} \,||\, \text{module\_ID} \,||\, \text{version\_ID}\rangle.\ ^{2}$$

Here, || denotes string concatenation. Opcode_tag distinguishes different naming convention, eg. *Software_ID* and *Custom_ID* defined below.

Ideally, we want to be able to uniquely assigning vendor_IDs to producers of software which can make the software_ID unique. This problem in practice might not be as difficult as it sounds since it is similar to domain name registration or the assignment of Medium Access Control (MAC) addresses by network card manufacturers. One can leverage on existing trust infrastructures to do this. For example, the responsibility for unique and well known software_IDs can be assigned to a Certificate Authority (CA), which then define the vendor_ID as $\langle$ CA_ID || vendor_name $\rangle$. Alternatively, one might be able to use the domain name of the software developer as a proxy for the vendor_ID.

A software_ID gives a one to one mapping between the binary and its ID string. This is useful for dealing with vulnerability management problems [15]. Suppose a new vulnerability is known for a particular version of a software. This means that certain binaries, providing that they correspond to that software version may be vulnerable. However, there is no simple and standard way of automatically determining this version information. Once we have software_IDs associated with binaries then one can check the software_ID against vulnerability alerts. The advisory may already contain the software_ID. Automatic scanners can then be used to tie-in this checking with the dissemination of vulnerability alerts to automatically monitor/manage/patch the software in an operating system. General management of patches in an operating system can also be done in much the same way.

In the case where no software_ID comes with a software product, one can alternatively derive one. It can be constructed, for instance, using the following (coarse-grained) string naming:

$$\textit{Custom\_ID} ::= \langle\ \text{opcode\_tag} \,||\, \textit{hash}(\text{vendor\_URL} + \text{product\_name} + \text{file\_name} + \text{salt})\ \rangle.$$

The salt expands the name space to reduce the risk of a hash function collision.

[2] Module_ID suffices to deal with software versioning. Having a separate version_ID, however, is useful to easily track different versions (or patched versions) of the same program.

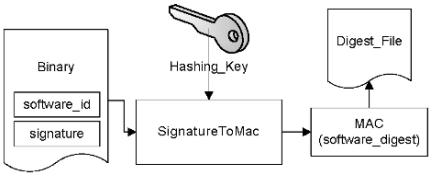

Fig. 1 SignatureToMac: Deriving the MAC

4.2 Binary Authentication System Architecture

In the following discussion, we assume here that binaries already come tagged with a software_ID. During the binary authentication set-up, preferably done immediately after the targeted binary installation, we generate the MAC values for each binary. In the case where binaries are digitally signed by its developer, then we verify the signature and then generate the MAC for each binary. Thus, only one public-key operation needs to be done at install time. We choose to use a keyed hash, the HMAC algorithm [4], so there is a secret key for the administrator. This is mainly to increase the security of the stored hashes. To authenticate binary integrity for any future execution of the code, only the generated HMAC needs to be checked. In what follows, we mostly write MAC which already covers the choice of HMAC.

One way of storing the generated MAC is by embedding it into the binary. However, doing so may interfere with file format of the signed binaries and may also have other complications. We instead use an authentication repository file which stores all the MAC values of authenticated binaries with their pathnames. During the boot-up process, the kernel creates its own in-memory data structures for binary authentication from this file. We ensure that the repository file is protected from further modification except under the control of the authentication system to add/remove binaries.[3] We can also customize binary authentication on a per user basis rather than system-wide which is a white-list of binaries approved for execution. In the case, when the initial binary does not have a digital signature, then the administrator can still choose to approve the binary and generate a MAC for it.

There are two main components of the system: the **SignatureToMac** and **Verifier**. The SignatureToMac maintains the authentication repository, *Digest_file*, consisting of ⟨path, MAC⟩ tuples. The Verifier is a kernel driver which makes use of *Digest_file* and decides whether an execution is to be allowed.

4.2.1 SignatureToMac

Once software is installed on the system, Fig. 1 shows how SignatureToMac processes the binaries:

[3] Further security can be achieved by integrating binary authentication with a TPM infrastructure. We do not do so in the prototype as that is somewhat orthogonal.

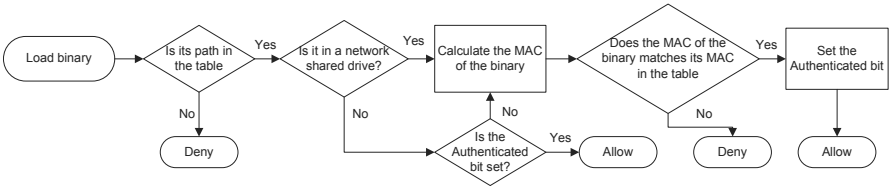

Fig. 2 Verifier: The Verifier in-kernel authentication process

1. Checks the validity of the binary's digital signature. If the signature is invalid, then report failure.
2. It consults the user or system administrator whether the software is to be trusted or not (this is similar to the Vista UAC dialog but only happens once). Other policies (possibly mandatory) can also be implemented.
3. It generates the MAC of the binary (including software_ID string) using a secret key, *Hashing_key*, to produce *software_digest*. The *Hashing_key* is only accessible by the authentication system, e.g. obtained on bootup.
4. It adds an entry for the binary as a tuple ⟨*path name*, *software_digest*⟩ into the *Digest_file* repository and informs the Verifier. The repository is protected against modification. Note that because the entries are signed, the repository can be read for other uses, e.g. version control and vulnerability management.

4.2.2 Verifier

The Verifier performs mandatory binary authentication — it denies the execution of any kind of Windows binary which fails to match the MAC and pathname. There are two general approaches for the checking. One is *cached MAC* which avoids generating the MAC for previously authenticated binaries. The other is *uncached MAC* which always checks the MAC. As we will see, they have various tradeoffs. The cached MAC implementation needs to ensure that binaries are unmodified. Hence, the Verifier monitors the usage of previously authenticated files on the cache, and removes them from the cache if it can be potentially modified.

The core data structure of the Verifier component can be viewed as a table of tuples in the form ⟨*Kernel_path*, *FileID*, *MAC*, *Authenticated_bit*⟩ representing the allowed binaries. It is indexed on *Kernel_path* and *FileID* for fast lookup. The fields are as follows:

- The *Kernel_path* is Windows kernel (internal) pathname representation of a file. In Window's user space, a file can have multiple absolute pathnames, due to: (i) 8.3 file naming format, e.g. "`C:\Program Files\`" and "`C:\progra~1\`" are the same; (ii) symbolic links (reparse points is similar); (iii) hard links; (iv) volume mount points; or (v) the `SUBST` and `APPEND` DOS commands. The *Kernel_path* is a unique representation for all the possible pathnames. When the

system loads ⟨*path name*, *software_digest*⟩ from *Digest_file* during the startup, *path name* is converted to *Kernel_path* since all subsequent checks by Verifier in the kernel all use the latter.

- The *FileID* is a pair of ⟨*device_name*, *NTFS_object_ID*⟩. The *device_name* is a Windows internal name to identify a disk or partition volume. For instance, the device name `HarddiskVolume1` usually refers to `C:\`. The *NTFS_object_ID* is a 128-bit length number uniquely identifying a file in the file system volume (this is not the same as Unix inode numbers) The Verifier uses the *FileID* to identify the same file given more than one hard link. This prevents an attacker from creating a hard link for modifying a binary without invalidating the binary cache. The *FileID* values will be queried from the system and filled into the table during system boot.
- The *MAC* is same as a *software_digest* entry in *Digest_file*. Our prototype implements the HMAC-MD5 [4], HMAC-SHA-1 and HMAC-SHA-256 [16] hash algorithms.[4]
- The *Authenticated_bit* remembers whether the binary has been previously authenticated. It is initially set to false, and set to true after successful authentication.

Fig. 2 shows the authentication process when a binary executes/loads:

1. It checks if the binary's *Kernel_pathname* exists in the table. If not, then deny the execution and optionally log the event. A notification is accordingly sent to the user.
2. It the file is on a network shared drive, goto step 4. The MAC is always recomputed as we cannot keep track of modification to files on network shares.
3. If the *Authenticated_bit* is set go to step 7.
4. It performs MAC algorithm operation on the binary.
5. If the resulting MAC doesn't match with the *MAC* stored in the table, execution is denied.
6. It sets the *Authenticated_bit* of the binary.
7. It passes the control to the kernel to continue the execution.

To control binary execution, we intercept the section creation action (`NtCreate-Section()` system call) which is better than:

- Intercepting file opening (`NtCreateFile()` and `NtOpenFile()`). This would need to authenticate any file opened with with execute access mode. As discussed in Sec. 2, however, this introduces unnecessary overheads and can cause some correct programs to fail if the files do not pass authentication. There are also technical difficulties to distinguish between process creation and regular file IO operations, which is not always easy given its microkernel nature.
- Intercepting process creation (`NtCreateProcess()`). This method is not effective for our purpose. Firstly, we cannot use it to control `DLL` loading. Secondly, it is more difficult to get the pathname of the binary because process creating is broken down into microkernel operations.

[4] Due to recent concerns which show weaknesses and attacks against MD5 [17], we also have stronger hash functions, namely SHA-1 and the stronger SHA-256.

It turns out that since all code from any kind of binary needs to have a memory section to execute, it suffices to intercept `NTCreateSection()`.

The cached MAC verifier needs to ensure that binaries which have been already authenticated are not modified. However, the uncached verifier will not need to perform file monitoring. A binary with pathname *P* is considered modified, if the following occurs:

- *P* is created: Hence, we monitor system call `NtCreateFile()` and `NtOpenFile()`.
- *P* is opened with write access mode: the previous two system calls are also intercepted for this purpose. [5]
- Another file is renamed to *P*: We monitor file renaming (`NtSetInformationFile(FileRenameInformation)`) system call.
- A drive containing *P* is mounted: We monitor drive mounting `IRP_MJ_VOLUME_MOUNT`.

Note that we do not need to monitor file deletion since we only care about executing correct files but not missing files. The details of file modification monitoring are given in Fig. 3.

Upon modification of *P*, we reset the *Authenticated_bit* of binary *P*, and update the *FileID* in the table if it is changed. Should FAT file system be used, pathname is used to identify the binary as *FileID* is not supported but neither are hard and soft links. Since *FileID* is optional and can be removed, we monitor *FileID* removal (`NtFsControlFile(FSCTL_DELETE_OBJECT_ID)`) and deny the removal if the *FileID* is in the table. Dut to the semantics of NTFS, our use of *FileID* can coexist with other applications using it.

If additional hardware and infrastructure is available to support secure booting, such as the Trusted Platform Module (TPM) initiative, the system can benefit from increased security. Offline attacks would have to first attack the TPM. The Hashing_key can also be stored securely by the TPM.

4.3 Security Analysis

The security of binary authentication relies on the strength of the chosen hash functions (MD5, SHA-1, SHA-256) as well as the HMAC algorithm. Thus, we assume that any change in a binary can be detected through a changed MAC.

In our authentication on binary with digital signature, the subsequent invocations using MAC verification is sufficient to ensure the authenticity of the binary. In other words, MAC authentications "*preserve*" the previously established properties of binary authentication derived from digital signature. A subtlety comes when the

[5] An alternative way is to monitor the file (block) writing operation (`NtWriteFile()`). However, it is less efficient because file block writings take place more frequently than file openings as one opened file for modification might be subject to multiple block writings. Furthermore, it cannot capture file-memory mapping.

```
procedure UponModification (FilePath)
   if (FS is NTFS)
      FileID := GetFileID(FilePath) # FileID can be NULL
      if (FilePath is in the table)
         Entry := LookupTableByPath(FilePath)
         if (FileID == NULL)
            # this can happen when the file is deleted and created again.
            # generate a new FileID and update the table
            Entry.FileID := CreateFileID(FilePath)
         else if ( FileID != Entry.FileID in the table)
            # this can happen when the drive is unmounted,
            # id changed off-line and re-mounted
            Entry.FileID := FileID
         end if
         Entry.Authenticated := false
      else if ((FileID != NULL) AND (FileID is in the table))
         Entry := LookupTableByID(FileID)
         Entry.Authenticated := false
      end if
   else if ((FS is FAT) AND (FilePath is in table))
      Entry := LookupTableByPath(FilePath)
      Entry.Authenticated := false
   end if
end procedure
```

Fig. 3 Pseudo code of file modification monitor

certificate expires or is revoked at some point in time after SignatureToMac. We view that the question of whether one should keep trusting the binary for execution depends on one's level of trust on certificate expiration/revocation. If the certificate expiration or revocation means that the public key must no longer be used, but the fact that *previously established* goodness binary properties still hold, then we can keep trusting the binary for execution (as long as we still believe the issuer).

Here we discuss some possible attacks to the authentication system. All the attacks except the last two target the caching system. More precisely, the attacker attempts to modify an already authenticated binary without causing the *Authenticated_bit* to be set to false.

- **Manipulating symbolic links:** The attacker can use the path S which is a symbolic link of an authenticated P to indirectly modify P and subsequently execute P. However, the modified file will not be executed successfully, because Windows kernel resolves symbolic links to real paths. More precisely, the symbolic link S is resolved to the real path P. As a result, the *Authenticated_bit* of P will be set to false. When P is executed, its MAC will be recalculated and it will not pass the authentication.
- **Manipulating hard links:** The attacker can create a hard link H on an already authenticated file P and then modifies the file using path H. This attack will not succeed because we use *FileID* to identify files. H has the same *FileID* as P, thus the *Authenticated_bit* will be set to false. Note that this attack will not succeed in FAT file system either, even though we cannot use *FileID*. This is because hard link is not supported in FAT.
- **Manipulating FileID:** Recall that *FileID* consists of *device_name* and *NTFS_object_ID*. The latter is optional and thus can be removed. The attacker can re-

move the *NTFS_object_ID*, and then performs the previous attack. We handle this attack by denying *NTFS_object_ID* removal on authenticated files. This is implemented by monitoring the file system control event `FSCTL_DELETE_OBJECT_ID`

- **Remote File Systems:** Since we cannot keep track of modification on a network shared file system, we do not cache the authentication. More precisely, the MAC of the binary is always calculated upon loading. Same applies to removable media such as floppy in which we can not keep track of modification of files.
- **TOCTTOU:** TOCTTOU stands for Time-Of-Check-To-Time-Of-Use. It refers to a race condition bug of an access control system where the resource is changed during the time of checking the resource to the time of using the resource. In the binary authentication context, the binary may be modified after the time it is authenticated and before the time it is executed. However, we observed that all binaries are exclusive-write-locked when it is opened. That means binaries cannot be modified from the time it is opened to the time it is closed. Also note that the file is authenticated after it is opened and before it is executed. As a result, binaries cannot be modified during TOCTTOU.
  When the binary is in a network shared volume, i.e. SMB share, and the write-lock is not properly implemented in the SMB server, an attacker is able to modify the binary after authentication. However, we have observed that both Windows and Samba implement write-lock properly. Thus the attack is only possible when the SMB server is compromised. One way to prevent this is to disallow binary loading from SMB share.
- **Driver Loading:** The binary authentication system authenticates all binaries including kernel driver. This means all drivers are authenticated thus driver attacks such as kernel rootkits and malware drivers can be prevented.
- **Offline Attack:** Offline attack means modification of the file system when Windows is not in control. For example, boot another OS or remove the disk drive for modification elsewhere. Such an attack will require physical access to the machine. Offline attack can corrupt data or change programs/files and affect the general functioning and we cannot prevent that. What we can do is to ensure the integrity of executable code and other data loaded in memory for processes.
  We assume that the kernel is still secure, i.e. authentication occurs early in the boot. We also assume that kernel functioning is not impaired, e.g. deleting some system files does not cause the kernel to have an exploitable vulnerability.
  Since the hashing_key is not stored in the machine, it is not available to the attacker. The attacker can still change the digest file and the binaries, however, MACs of modified binaries cannot be produced without the hashing_key. Thus modified binaries cannot be executed when the system is online.

5 Empirical Results

Our authentication system can detect when a modified binary is loaded or run from the wrong pathname. In this section, we examine the three factors which impact on

system performance: (i) Verifier checking upon binary loading (execution); (ii) file modification monitoring; and (iii) binary set-up during the SignatureToMac process. The first two above are the most important as they directly affect user's waiting time for process execution and affect overall system operation. The tests here are meant to determine the worst case overhead as well as average overheads.

The benchmarks are run on a Core 2 Duo with 2GB of ram running Windows XP with SP2. Each benchmark is run five times. As we want to investigate the effect of the cached Verifier, each benchmark is run with caching and without caching. When caching is enabled, we ignore the result of the first run because the overhead of authentication overhead is already shown in the uncached case. Even if we count the first run, its impact will be very small because some of the microbenchmarks run for 10K times, so the authentication overhead becomes negligible.

To see the difference of using different hashing algorithms, we implement and benchmark three algorithms: MD5, SHA-1 and SHA-256. Only MD5 and SHA-256 are shown in Table 1 as the results of SHA-1 are always between these. When caching is enabled, results of different hashing algorithms are not distinguished (shown as Cached-MAC in the table), because binaries are not require MAC checking during the benchmark. The reason is that the first run is ignored, and the binaries are not modified during the benchmark.

To see the difference with digital signature based authentication system, we also compare the performances of our scheme against the Microsoft official Authenticode utility called `Sign Tool` [18], and another Sysinternals (now acquired by Microsoft) Authenticode utility `Sigcheck` [19]. Note that two tools are user-mode programs. They are there to illustrate the difference between non-mandatory strategies used with Authenticode with our in-kernel mandatory authentication.

The first two benchmarks investigate system performance under two scenarios:

1. **Micro-benchmark:** The micro-benchmark aims to measure the worst case performance overhead incurred by the scheme. Note that this is primarily intended to measure the authentication cost but not other overhead, which is done by the last file modification microbenchmark. Here, we have two micro-benchmark scenarios.

   a. **EXE Loading:** This executes the `noop.exe` program, a dummy program that immediately exits, for 10K times. This scenario measures the overhead for authenticating the `EXE` file. The benchmark program first calls `CreateProcess()`, and waits for the child process' termination using the `WaitForSingleObject()` function. We use different binary sizes (40KB, 400KB, 4MB and 40MB, only the 40K and 40MB results are displayed) for noop.exe to see how executable size impacts performance.
   b. **Loading DLL:** The second scenario executes the `load-dll.exe` program for 100 times. This scenario is used to find out how the number of loaded DLLs impacts the performance. Program `load-dll.exe` loads 278 standard Microsoft DLLs with a total file size of ∼75MB. The size of the `load-dll.exe` itself is 60KB. Note that in Windows, the bulk of code is

Authentication System	Micro-Benchmark: noop 40K time	noop 40K slowdown	noop 40M time	noop 40M slowdown	load-dll time	load-dll slowdown	Macro-Benchmark: build time	build slowdown
Clean	22.76	–	30.07	–	45.32	–	66.26	–
EXE Only:								
Signtool	2822	11637%	4850	16033%	73.49	62.16%	97.00	46.39%
Sigcheck	1720	7457%	5629	18623%	62.82	38.62%	110.5	66.72%
Uncached-MD5	25.96	14.08%	2150	7052%	45.34	0.05%	70.85	6.93%
Uncached-SHA256	30.29	33.07%	9005	29851%	45.34	0.05%	71.79	8.35%
Cached-MAC	23.20	**1.93%**	30.63	**1.88%**	45.33	**0.02%**	67.62	**2.06%**
All Binaries:								
Signtool	11867	52043%	14030	46565%	16018	35244%	–	–
Sigcheck	4283	18772%	6186	20478%	12548	27587%	–	–
Uncached-MD5	26.10	14.67%	3881	12811%	128.8	184.1%	79.31	19.69%
Uncached-SHA256	30.42	33.67%	9302	30839%	201.3	344.0%	91.80	38.55%
Cached-MAC	23.25	**2.14%**	30.58	**1.72%**	45.35	**0.07%**	67.88	**2.45%**

Table 1 Benchmark results showing times (in seconds) and slowdown factors. The worst slowdown factors for each benchmark scenario are shown with underline, whereas the best are in bold. We define $slowdown_x = (time_x - time_{clean})/time_{clean}$.

often in DLLs which is why the EXE file may be small, e.g. Open Office has over 300 DLLs.

2. **Macro-benchmark:** The macro-benchmark measures overhead under a typical usage scenario. Our benchmark is to create the Windows DDK sample projects using the build command. In each test run, 482 C/C++ source files in 43 projects are built. This benchmark is chosen as it is deterministic, non-interactive, creates many processes and uses many files.

We benchmark Sign Tool and Sigcheck in the following fashion. We first sign noop.exe and load-dll.exe using Sign Tool's signing operation. We then measure the execution time of authenticating and executing the two programs. For the macro-benchmark, we replace each development tool in the DDK (i.e. *build.exe*, *nmake.exe*, *cl.exe* and *link.exe*) with a wrapper program which first authenticates the actual development tool and then invokes it. For the micro-benchmark, we consider two settings: (i) EXE only; and (ii) all binaries (EXE + DLL). The macro-benchmark, however, only tests the EXE case. This is because, during the macro-benchmark, many programs are invoked, and each program each invocation may dynamically load a different set of DLLs. Thus, it is hard to keep track of what DLLs are loaded, and it is unfair to simulate with all DLLs used.

The results are given in Table 1 but we have not shown "noop 400K" and "noop 4M" because they are bounded by the results of "noop 40K" and "noop 40M". Other results not shown are that the overhead is approximately linear with respect to the file size, e.g. the results of the All-binaries/Uncached/SHA-256 benchmarks are 40K:30.42s, 400K:85.43s, 4M:598.0s, 40M:9302s.

We can see that the overhead of Signtool and Sigcheck makes it unusable if DLLs are to be checked (352x slower on load-dll). If only EXE are checked, then

at least 40% overhead and based on the `load-dll` benchmark, one could expect about an order of magnitude worse if all DLLS are checked. Of course, using these tools would incur additional overhead from creating a process and the main purpose is just to show the difference between what can be done in user-mode versus in-kernel. We can see that all the uncached-MD5/SHA256 are considerably faster than `Signtool` and `Sigcheck`.

Authenticating only `EXE`, the difference between uncached has overheads around 8% while cached brings this down to very small, around 2%x and almost neglible in the `load-dll` benchmark (0.02%). Note that as uncached overhead is quite small, the results are dominated by non-determinism in timing measurements. Moving to all DLLs (`EXE + DLL`), we can see the effect of Windows programs using many DLLs (more code in `DLL` than `EXE`). The overhead incurred by caching is still small while uncached can grow to between 20-40% depending on the hash algorithm. Note that the uncached overhead is applicable for files which cannot be cached.

The final microbenchmark investigates the tradeoffs between cached and uncached verification. Caching means that MAC verification is amortized over executions but has added overhead from monitoring file modification, while uncached is the opposite. Our micro-benchmark opens a file for writing 100K times to measure the worst case overhead incurred by file modification monitoring. We have 3 experiments: (i) a clean system without binary authentication; (ii) binary authentication with cache and the modified file is a binary; and (iii) binary authentication with cache and the modified file is not a binary.

The results for the file modification micro-benchmark show that for binary authentication with a cache, it doesn't matter whether the file being written to is a binary or not. Both cases incur about 60% overhead compared to a clean system. Since binary authentication with no-cache has no overheads for file modifications, this means that under some usage scenarios where file modification is very high, the uncached strategy may be preferable over cached even when Verifier overhead is higher.

6 Conclusion

We have shown a comprehensive system which authenticates both content and pathname for Windows to ensure that only trusted binaries are executed. Unlike other operating systems, Windows poses significant challenges. We show that it is possible to ensure that only trusted binaries can be loaded from files for execution. This can also be combined with a simple software ID scheme which simplifies binary version management, and dealing with vulnerability alerts and patches. Our system is lightweight and integrates well with PKI and trust mechanisms without having to rely on them. The overheads of our prototype are quite low when caching is is used. In the case of workloads with heavy file modifications, an uncached strategy might be preferable. The overheads are still low in this case, since the system overhead will be dominated by I/O rather than binary authentication, so the overall binary authen-

tication would still be low as a percentage of overall system overhead. In summary, although this is a prototype, it significantly adds to the security of any Windows system but at the same time is sufficiently flexible so that it can be tailored for different usage scenarios.

References

1. A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi and V. Roy, "DigSig: Run-time Authentication of Binaries at Kernel Level", *Usenix LISA*, 2004.
2. M.A. Williams, "Anti-Trojan and Trojan Detection with In-Kernel Digital Signature testing of Executables", NetXSecure NZ Ltd.. http://www.netxsecure.net/downloads/sigexec.pdf, 2002.
3. L. v. Doorn, G. Ballintijn, and W. A. Arbaugh, "Signed Executables for Linux", *Technical Report CS-TR-4256* University of Maryland, 2001.
4. H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", *RFC 2104*, 1997.
5. CERT, "Vulnerability Remediation Statistics", http://www.cert.org/stats/vulnerability_remediation.html, 2007.
6. R. Grimes, "Authenticode", Microsoft Technet, http://www.microsoft.com/technet/archive/security/topics/secaps/authcode.mspx?mfr=true.
7. Microsoft TechNet, "KnownDLLs", http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/regentry/29908.mspx.
8. "Microsoft Security Advisor Program: Microsoft Security Bulletin (MS99-006)", http://www.microsoft.com/technet/security/bulletin/ms99-006.mspx.
9. G.H. Kim and E.H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker", *ACM CCS*, 1993.
10. E.R. Arnold, "The Trouble With Tripwire", http://www.securityfocus.com/infocus/1398, 2001.
11. M. Slaviero, J. Kroon and M. S. Olivier, "Attacking Signed Binaries", *Proc. of the 5th Annual Information Security South Africa Conference (ISSA)*, 2005.
12. B. Acohido, "Security feature in Microsoft's new Windows could drive users nuts", USA Today, http://www.usatoday.com/tech/products/2006-05-15-vista-security_x.htm?POE=TECISVA, 2006.
13. M. Schmid, F. Hill, A.K. Ghosh, and J.T. Bloch, "Preventing the Execution of Unauthorized Win32 Applications", *DARPA Information Survivability Conf. & Exposition II (DISCEX)*, 2001.
14. S. Patil, A. Kashyap, G. Sivathanu, E. Zadok, "I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System", *USENIX LISA*, 2004.
15. Sufatrio, R. Yap and L. Zhong, "A Machine-Oriented Integrated Vulnerability Database for Automated Vulnerability Detection and Processing", *USENIX LISA*, 2004.
16. D.E. Eastlake and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, 2006.
17. X. Wang, H. Yu, "How to Break MD5 and Other Hash Functions", Eurocrypt '05, LNCS 3494, Springer, 2005.
18. "Sign Tool", http://msdn2.microsoft.com/en-us/library/8s9b9yaz(vs.80).aspx.
19. "Sigcheck", http://www.microsoft.com/technet/sysinternals/Security/Sigcheck.mspx.