On Challenges in Verifying Trusted Executable Files in Memory Forensics

Daniel Uroz, Ricardo J. Rodríguez*

Dept. of Computer Science and Systems Engineering,
University of Zaragoza, Spain

Abstract

Memory forensics is a fundamental step in any security incident response process, especially in computer systems where malware may be present. The memory of the system is acquired and then analyzed, looking for facts about the security incident. To remain stealthy and undetected in computer systems, malware are abusing the code signing technology, which helps to establish trust in computer software. Intuitively, a memory forensic analyst can think of code signing as a preliminary step to prioritize the list of processes to analyze. However, a memory dump does not contain an exact copy of an executable file (the file as stored in disk) and thus code signing may be useless in this context. In this paper, we investigate the limitations that memory forensics imposes to the digital signature verification process of Windows PE signed files obtained from a memory dump. These limitations are data incompleteness, data changes caused by relocation, catalog-signed files, and executable file and process inconsistencies. We also discuss solutions to these limitations. Moreover, we have developed a Volatility plugin named `sigcheck` that recovers executable files from a memory dump and computes its digital signature (if feasible). We tested it on Windows 7 x86 and x64 memory dumps. Our experiments showed that the success rate is low, especially when the memory is acquired from a system that has been running for a long time.

Keywords: memory forensics, Authenticode, digital signature verification, code signing, Volatility

1. Introduction

A common kind of security incident is caused by the presence of malicious software (*malware*) in a system. As part of a security incident response process, computer and network forensics becomes a fundamental step during the detection and analysis stage. Anomalous or unauthorized activity performed by malware in a compromised system can be detected through the analysis of both the device drive and the memory of the system. Disk forensics is the part of forensics focused on the analysis of device drives. On the other hand, memory forensics focuses on the analysis of the data contained in the memory of the system under study (Ligh et al., 2014).

Although we normally have both possibilities, there are situations where the access to the physical device drives is difficult to accomplish (for instance, in Cloud computing). Furthermore, the current storage capacity in disk devices make the analysis of memory content (in the order of magnitude of gigabytes) a more affordable task that the analysis of disk content (in the order of magnitude of terabytes), as well as facilitate the initial triage. Considering these facts, in this paper we focus on memory forensics.

Memory forensics is usually carried out capturing the current state of the system's memory and dumping it into disk as a snapshot file. This file is also known as *memory dump*. A memory dump can then be taken offsite to analyze it with dedicated software such as Volatility (Walters and Petroni, 2007), searching for facts about the security incident.

A memory dump contains tons of data that might be of interest for analysis. Among other things, it contains a snapshot of the processes in execution, as well as other system information such as logged users, open files, or open network connections at the time of memory acquisition. Note that the memory state can be inconsistent if it was acquired in a live system, since the system itself evolves over time and system objects might be created or destroyed during the acquisition process. To assess the reliability of analysis results, the use of the temporal dimension in memory forensics has been recently proposed (Pagani et al., 2019).

Code signing helps to establish trust in computer software, since it allows to authenticate the software publisher and to guarantee code integrity through the validation of the digital signature shipped within the software (Parno et al., 2010). Several operating systems rely on code signing to warn the users about the potentially harmful actions that a piece of software may perform. For instance, the execution of a properly signed application in Windows avoids any alert box informing the user about the possible harmful consequences of its execution. Under this premise, malware developers use digital signatures to deceive users to execute their malware and thus compromise their systems, thus subverting the trust in digitally signed software.

Although the use of digital signatures in malware is not a growing trend (Ugarte-Pedrero et al., 2019; Rivera et al., 2019),

*Corresponding author
Email address: `rjrodriguez@unizar.es` (Ricardo J. Rodríguez)

there are documented cases of signed malware samples in the wild (such as Stuxnet (Langner, 2011), Duqu, or Flame, to name a few). Malware developers use trusted certificates that were either compromised or issued directly to them to sign their software. As the primary defense against these threats, we rely mainly on the revocation process of the abused certificates done by the certification authorities (CAs).

Intuitively, a forensic analyst can think of code signing as a preliminary step to prioritize the list of suspicious processes that need further analysis. The rationale for this thought is correct, but unfortunately is not very fruitful when inspecting a memory dump: a process is an inaccurate representation of the executable files[1] in memory, since parts of the binary code may be paged out of memory or may change at acquisition time (Case and Richard, 2017). Furthermore, software defenses such as address space layout randomization (Bhatkar et al., 2003; PaX Team) or position-independent code may change the memory references of certain binary code instructions. But, to what extent can these issues negatively affect the signature computation? Is there any other issues affecting it? Are there any ways to overcome these problems? These questions have motivated our research. Our main research goal in this paper is to explore whether code signing brings any benefit to memory forensics.

Contributions. In this paper, we describe the limitations that memory forensics imposes to the digital signature verification process of Windows PE signed files. In particular, Authenticode is the code signing standard designed to digitally sign files in Windows, introduced in Windows 2000 (Microsoft Corporation, 2008). We focused on Windows since it is still the preferred target of malware authors (AV-TEST, 2019). We have also developed a Volatility plugin to verify digital signatures in a memory dump, named `sigcheck` (as the tool provided by Microsoft for verifying digital signatures on binary files (Russinovich, 2019)). When feasible, our plugin works on kernel-space file objects that represent executable files, computing the signature and verifying the certificate chain attached to the digital signature. To assess the reliability of our plugin, we tested it in different scenarios and in signed malware samples. We concluded that the longer a system runs, the fewer file objects can be acquired. Hence, given the current limitations (data incompleteness, data changes caused by relocation, catalog-signed files, and executable file and process inconsistencies), the verification of digitally-signed files does not bring any benefit to memory forensics.

Structure of the paper. The structure of this paper is as follows. Section 2 gives background on previous concepts. In Section 3, we describe in detail the plugin `sigcheck` that we have developed. Section 4 details the experimentation and discussion of results. Section 5 is devoted to explaining the limitations imposed by memory forensics and possible solutions. Section 6 presents related work. Finally, Section 7 concludes the paper and states plans for future work.

[1] In this paper, we refer as *executable file* to the representation of a file as in disk and as *image file* to the representation of a file as in memory.

2. Previous Concepts

In this section, we briefly describe first the Microsoft Authenticode code-signing technology and then we explain the internal structures used by the Windows operating system (OS) to represent files.

2.1. Microsoft Authenticode

Microsoft Authenticode is the code-signing standard used by Windows to digitally sign files that adopt the Windows portable executable (PE) format (Microsoft Corporation, 2008), which includes executables (files with extension `.exe`), dynamically loaded libraries (`.dll`), and drivers (`.sys`), among other files.

Based on Public-Key Cryptography Standard (PKCS) #7 (Kaliski, 1998) and X.509 version 3 certificates (code-signing certificates issued by certificate authorities), it allows to bind an Authenticode-signed binary to the identity of a software publisher and to guarantee the integrity of the file content.

The Authenticode signature of a PE file follows the PKCS#7 structure that includes the signature (the hash value of the PE file), a timestamp (optional) and the certificate chain. Roughly speaking, the Authenticode signature is a binary data blob consisting of a certificate and a signed hash of the file (which excludes certain parts of the PE header in the hash calculation). As hash algorithms, Authenticode supports MD5 (currently supported only for backwards-compatibility), SHA-1, and SHA-256 hashes. Let us note that a PE file can be dual-signed by applying multiple signatures, which is strongly recommended when using deprecated hashing algorithms such as MD5. The certificate chain is built to a trusted root certificate by using X.509 chain-building rules. This trusted root certificate is mandatory as long as the root certificate is not present in the users' root stores.

An Authenticode-signed Windows file can be shipped in two different ways, either through an embedded signature within the PE file structure or through a digitally-signed catalog file. Whether embedded signed or catalog signed, both signatures are stored as PKCS #7 signed data, which follows the Abstract Syntax Notation One (ASN.1) format (ITU-T, 1984).

ASN.1 is a joint standard of the International Telecommunication Union Telecommunication Standardization Sector and ISO/IEC broadly used in telecommunications, computer networking, and cryptography (for instance, ASN.1 is used in X.509 certificates). It states how to store binary data of different data structures and includes a set of encoding rules that specify how to represent a data structure as a series of bytes to facilitate their serialization. In particular, Authenticode signatures follows the DER-encoded ASN.1 format.

The digital signature data in an embedded Authenticode-signed Windows PE file is appended to the file. The offset and size of the embedded signature is stored in the `Security directory` entry within the `Data directories` array of the PE optional header. The `Data directories` array contains offsets and sizes of different structures within the PE file, such as the export, import, or relocation directories, among others. All directories but the security directory store their offsets as relative virtual address (RVA) offsets, which means that they are the

virtual addresses from the PE file once it is loaded into memory. On the contrary, the security directory stores its offset as a file offset.

The binary data stored at the security directory file offset is a `WIN_CERTIFICATE` structure (Microsoft Corporation, c), which defines the signature length (in bytes), the certificate revision, the type of certificate, and the certificate list. In the case of Authenticode signatures, the certificate type matches a PKCS#7 `SignedData` structure.

A catalog file (extension `.cat`) collects digital signatures for an arbitrary number of files. It contains a collection of cryptographic hashes, each one corresponding to a file that is included in the collection. To prevent unauthorized modifications, catalog files are also Authenticode-signed files. Catalog files are located in the `system32/catroot` path within the Windows directory (normally, `C:\Windows`). The database of catalog files in a Windows system is maintained in a separate file, named `catdb`, which is located in the `system32/catroot2` path within the Windows directory. This database follows the Extensible Storage Engine (ESE) format, an indexed and sequential access method storage technology developed by Microsoft.

Signature Verification of Authenticode-Signed Windows Files

Microsoft designed the subject interface package (SIP) architecture to support the creation, retrieval, and hash calculation and validation of digital signatures, abstracting software developers about any inner behavior. The verification of signed code is performed using a trust provider, which is a piece of software that decides whether a given file is trusted based on the certificate associated to that file. Every trust provider establishes its own criteria to validate a signed file. A more detailed explanation of SIP and trust providers architectures, as well as their intrinsic relationships, is given in (Graeber, 2018). In a nutshell, a series of Windows APIs of distinct system's dynamic link libraries (DLLs) are invoked to verify the digital signature of a file.

We have manually checked the execution flow followed by the Windows OS to verify an Authenticode-signed file. To do so, we have extracted and then analyzed the Windows API trace of an execution of the Sysinternals' `sigcheck` (Russinovich, 2019) tool when verifying an embedded-signed file. The main core steps of the verification process is done by the `WINTRUST` and `CRYPT32` DLLs. The (summarized) execution flow is as follows:

1. First, the function `WinVerifyTrust` (in `WINTRUST`) is invoked (Microsoft Corporation, d). This function performs a trust verification action on a specified object, inquiring to a trust provider that supports the action identifier passed by parameter. We are interested in the Authenticode trust provider, specified by the `WINTRUST_ACTION_GENERIC_VERIFY_V2` action identifier. Furthermore, this function accepts a structure named `WINTRUST_DATA` where we can explicitly indicate if we want to perform the certificate revocation process, among other verification options.
2. Then, the function `CryptSIPRetrieveSubjectGuid` (in `CRYPT32`) is invoked to retrieve the globally unique identifier (GUID) –a 128-bit integer number used to identify resources– of the SIP implementation. This GUID is passed to the function `CryptSIPLoad`, which loads the DLL that implements the specified SIP and sets pointers to SIP-related functions appropriately. The Windows' Authenticode trust provider of PE files is implemented in `WINTRUST`.
3. The first of these called functions is `CryptSIPGetSignedDataMsg`, which retrieves the Authenticode signature from a file. Internally, this function eventually invokes to the function `ImageGetCertificateData` (in `imagehlp DLL`) to retrieve the Authenticode signature from the given file.
4. Once the signature is retrieved from the file, the function `CryptSIPVerifyIndirectData` is invoked to verify the hash of the file. When computing the Authenticode hash of a PE file, certain parts of the PE file are skipped: the Authenticode signature itself, the pointer reference to the location of the Authenticode signature (located in an entry of the data directories), and the PE file checksum (located in the optional header). This functions returns a boolean value indicating whether the Authenticode hash verification was succeeded.
5. Once the Authenticode hash is validated, the process continues invoking the function `CryptVerifyTimeStampSignature` to validate the timestamp signature on the Authenticode data though a Time Stamping Authority, which certifies that it observed the signed data at a specific time. This step is optional since a signature may optionally be timestamped.
6. When the certificate chain needs also to be validated, the verification process continues after the validation of the Authenticode signature. The function `CertGetCertificateChain` (in `CRYPT32`) is eventually invoked, which builds a certificate chain context starting from an end certificate and going back (if possible) to a trusted root certificate. This function accepts some flags to indicate whether extra processing is required. In our experiments, we have observed that the Sysinternals' `sigcheck` tool uses the value `CERT_CHAIN_CACHE_END_CERT | CERT_CHAIN_REVOCATION_ACCUMULATIVE_TIMEOUT | CERT_CHAIN_REVOCATION_CHECK_CHAIN_EXCLUDE_ROOT`. A full description of these flags is given in (Microsoft Corporation, a).

The verification process of a catalog-signed file is done in a programmatic way. The functions needed are defined also in `WINTRUST` and have `CryptCAT` as prefix in the function name. A correct way to implement this is to first acquire a handle to a catalog administrator context (function `CryptCATAdminAcquireContext` –it requires to later invoke the function `CryptCATAdminReleaseContext`), then to compute the Authenticode hash of the file using the function `CryptCATAdminCalcHashFromFileHandle`, and finally get the handle to the catalog that contains the given hash using the function `Crypt-`

CATAdminEnumCatalogFromHash. Once the specific catalog is retrieved, the function WinVerifyTrust can be invoked to verify the file signature against that given catalog. Internally, since the catalog file is also an Authenticode-signed file, the function WinVerifyTrust may perform as indicated before to validate the signature of the catalog file.

2.2. File Objects

File Objects represent the memory mapped files into the kernel memory, acting as the logical interface between kernel and user-mode processes and the corresponding file data stored in the physical disk (Microsoft Corporation, b). This object is used by Windows to track a single open instance of a file (that is, if you open multiple times a file, you have multiple file objects) (Russinovich et al., 2012). The representation of this internal memory structure is depicted in Figure 1. Apart from the data written to the file, the file object contains a set of kernel-maintained attributes, such as a pointer to the Unicode-encoded file name and a pointer to a structure named Device object that represents the device which contains the file.

A file object also stores a pointer to a SECTION_OBJECT_POINTERS structure, used by the memory manager and cache manager to store file-mapping and cache-related information for a file stream. This structure is conformed by three opaque pointers (DataSectionObject, SharedCacheMap, and ImageSectionObject). An opaque pointer points to a data structure whose contents are not exposed at the time of its definition.

The pointer DataSectionObject is used to track state information for a data file stream. That is, it may pointer to data files such as those used by a word processor o a spreadsheet. Similarly, the pointer ImageSectionObject is used to track state information for an executable file stream. Roughly speaking, it points to executable files mapped into memory (images). The pointer SharedCacheMap points to a structure used by the Windows OS to maintain the cache.

Only DataSectionObject and ImageSectionObject are of interest to our research, since both structures are useful to look for executable files. In a memory dump, we can look for the file objects handled by the Windows OS at the time the memory dump was acquired. We may find a DataSectionObject pointing to the memory zone where the executable file was mapped as data file –although it may contain end padding bytes by memory alignment issues. Similarly, we may also find an ImageSectionObject pointing to the memory zone where the executable file was mapped prior to be initialized by the Windows PE loader – that is, a memory zone that contains the image file after the relocation process was done to prepare the image for execution. As before, it may contain also end padding bytes. Unlike DataSectionObject, an ImageSectionObject is missing the data related to the Authenticode signature contained in the PE header. We argue that the PE loader strips this data before mapping the file, since it is used to verify the file prior execution and thus it is not longer needed. Surprisingly, though, this internal structure contains the relocation information even it is neither longer needed.

3. Description of the plugin sigcheck

This section describes the Volatility plugin that we developed to verify digital signatures of executable files (namely, .exe, .dll, and .sys files) in memory dumps. We named the plugin sigcheck, as the Microsoft's tool that verifies digital signatures on binary files (Russinovich, 2019). We have selected Volatility as memory forensics framework since it has become a *de facto* standard for analyzing memory dumps in computer forensics. Initially released in 2007 (Walters and Petroni, 2007), Volatility is currently an open source project that supports the analysis of memory dumps from the main-streams operating systems (namely, Windows, Linux, and macOS), and in both 32-bit and 64-bit architectures. Every supported OS has its own *profile*, which helps Volatility to find the references to internal OS structures within the memory dump. Volatility is implemented in Python and incorporates an API that allows developers to implement their own plugins or to extend the features of the official tools very easily.

Our plugin relies also on a set of plugins shipped with Volatility. In particular, it uses tasks (to retrieve the list of processes in execution), modules (to retrieve the list of drivers), devicetree (to retrieve the driver objects for a given module), filescan (to retrieve the list of file objects), and dumpfiles (to obtain the list of memory addresses associated to a FileObject structure. The content of these addresses is later read in a programmatic way).

Algorithm 1 shows a pseudo-algorithm of the workflow followed by sigcheck to verify the digital signature of an executable module e, contained in a memory dump $\mathcal{M}$, both given as inputs. As output, it returns a status code to indicate whether or not the digital signature of e was verified. This status code is then parsed to a user-friendly message to report the user appropriately.

As first step, the plugin obtains the file object f associated to the executable module e (line 1). Then the boolean flag *pe_rebuilt* is initialized to false value (line 2). If f does not exist or does not contain any valid data, it returns *FILEOBJECT_ERROR* (lines 2–3). In our experiments, every file object that we found contained either a DataSectionObject or an ImageSectionObject valid pointer (cf. Section 2.2). These memory zones are then checked for completeness.

When data is complete (lines 5–43), first the end padding is removed from the file object f. Then, f is interpreted as a PE file and stored in p (line 7). If f is in fact an ImageSectionObject structure, then we look for an image address $a \in \mathcal{A}$ such that the computation of the PE checksum is correct after undoing any relocation, considering that value as image address (line 9). Note that our tool needs a list $\mathcal{A}$ of frequent image base addresses. We have populated this list after analyzing all system executable files in the Windows OS flavors used in the experiments (cf. Section 4). In case that no valid address is found, it returns *PE_REBUILT_FAILED* (lines 10–11). Otherwise, the flag *pe_rebuilt* is set to true value. Then, the validation of the PE checksum is performed only if *pe_rebuilt* is false, and if there is a mismatch, it returns *PE_CHECKSUM_MISMATCH* (line 17). Otherwise, the verification of the

```
Input: A memory dump M and an executable module e
Output: Returns a status code indicating whether the verification of digital signature was successful or failed otherwise
   /* Let A be the list of frequent image base addresses */
1  Extract from M the file object f associated to e
2  Initialize the flag pe_rebuilt to false value
3  if f does not exists or does not contain data then
4  |   return FILEOBJECT_ERROR
5  else if data content of f is complete then
6  |   Delete end padding of f (if any)
7  |   Let p be the content of f, interpreted as a PE file
8  |   if data content of f is an ImageSectionObject then
9  |   |   Select an image address a ∈ A such that the PE checksum of p is correct, after undoing the relocation process
10 |   |   if a is not found then
11 |   |   |   return PE_REBUILT_FAILED
12 |   |   else
13 |   |   |   Set the flag pe_rebuilt to true value
14 |   |   end
15 |   end
16 |   if not pe_rebuilt and PE checksum of p is incorrect then
17 |   |   return PE_CHECKSUM_MISMATCH
18 |   else
   |   |   /* cf. Algorithm 2 */
19 |   |   return validate_signed_PE_file(p, pe_rebuilt)
20 |   end
21 else
22 |   Let p be the content of f, interpreted as a PE file
23 |   if p cannot be read as a PE file then
24 |   |   return PARTIAL_CONTENT_PE_DATA_ERROR
25 |   end
26 |   if p contains an embedded Authenticode signature then
27 |   |   if data content of f is an ImageSectionObject then
28 |   |   |   return SIGNED_FILE_NOT_VERIFIED
29 |   |   else
30 |   |   |   if the certificate chain c, contained in p header, exists then
31 |   |   |   |   return Verification of the certificate chain c, prefixing it with PARTIAL_CONTENT
32 |   |   |   else
33 |   |   |   |   return CONTENT_SIGNED_NOT_VERIFIED
34 |   |   |   end
35 |   |   end
36 |   else
37 |   |   if f is located in Windows system root directory then
38 |   |   |   return PARTIAL_CONTENT_MAYBE_CATALOG_SIGNED
39 |   |   else
40 |   |   |   return PARTIAL_CONTENT_NOT_SIGNED
41 |   |   end
42 |   end
43 end
```

Algorithm 1: Pseudo-algorithm of the analysis performed by the plugin `sigcheck`.

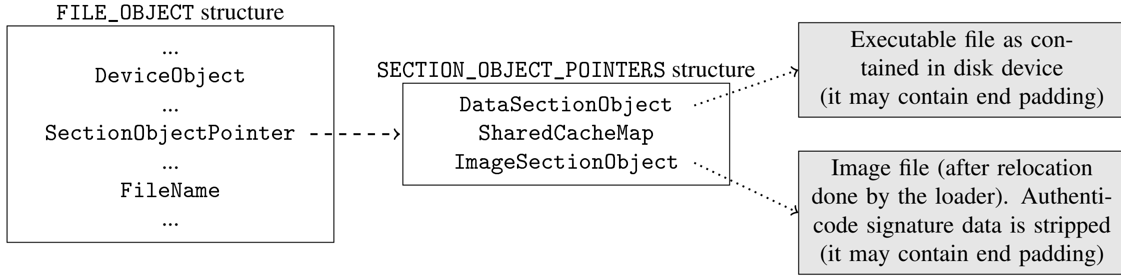


Figure 1: Representation of navigation from a `FILE_OBJECT` structure to `DataSectionObject` and `ImageSectionObject` structures.

Input: A PE file p and a boolean value *pe_rebuilt*
Output: Returns a status code indicating whether the verification of digital signature was successful or failed otherwise

```
   /* Let H be the list of hash values contained in every catalog file */
 1 Compute h_p as the Authenticode hash of p
 2 if the certificate chain c, contained in p header, exists then
 3 |   Retrieve the Authenticode hash value from c as h_c
 4 |   if h_p is equal to h_c then
 5 |   |   return Verification of the certificate chain c
 6 |   else
 7 |   |   if pe_rebuilt then
 8 |   |   |   return AUTHENTICODE_SIGNATURE_MISMATCH_OR_INCORRECT_IMAGEBASE
 9 |   |   else
10 |   |   |   return AUTHENTICODE_SIGNATURE_MISMATCH
11 |   |   end
12 |   end
13 else
14 |   if h_A ∈ H then
15 |   |   return CATALOG_SIGNED
16 |   else
17 |   |   if f is located in Windows system root directory then
18 |   |   |   return MAYBE_CATALOG_SIGNED
19 |   |   else
20 |   |   |   if pe_rebuilt then
21 |   |   |   |   return NOT_SIGNED_OR_INCORRECT_IMAGEBASE
22 |   |   |   else
23 |   |   |   |   return NOT_SIGNED
24 |   |   |   end
25 |   |   end
26 |   end
27 end
```

Algorithm 2: Pseudo-algorithm of the signature verification process.

PE signed file p is validated through the steps shown in Algorithm 2 (explained below). When data is incomplete (lines 21–39), first f is interpreted as a PE file and stored in p (line 22). If there is some error with the PE data content (e.g., important data is missing), it returns *PARTIAL_CONTENT_PE_DATA_-ERROR*. Then, it is checked if p contains an embedded Authenticode signature. If so, if the data content of f is an `ImageSectionObject` structure, the tool returns *SIGNED_FILE_NOT_-VERIFIED* (line 28). Otherwise, if the certificate chain c exists, it is verified (line 30). We rely on OpenSSL 1.1.0l binary package for this purpose, following a strict certificate validity time. Unfortunately, we have empirically tested that this software does not verify the revocation of certificates. We plan to solve this issue programmatically as future work. Our plugin returns the result of the certificate chain verification (explained below), prefixing it with *PARTIAL_CONTENT*. If c does not exists, the tool returns *CONTENT_SIGNED_NOT_VERIFIED*. If the Authenticode header is missing, it is checked whether the file object f is located in the Windows directory or its subdirectories (line 34). In that case, the tool returns *PARTIAL_-CONTENT_MAYBE_CATALOG_SIGNED* or *PARTIAL_CONTENT_NOT_SIGNED* otherwise (lines 38 and 40, respectively).

The verification of an Authenticode-signed file has been implemented as an independent Python function (named `sigvalidator`), and thus it can be used with image files as well as with executable files. Its workflow is detailed in Algorithm 2. As input, it needs a PE file p and a boolean value to indicate whether the PE file was rebuilt.

First, the Authenticode hash h_p of p is computed (line 1). The steps of the Auhtenticode algorithm are published in (Microsoft Corporation, 2008). If p contains an Authenticode header, it is retrieved to extract its Authenticode signature h_c and the certificate chain c. When the Authenticode hashes h_c and h_p mismatch (line 4), it returns either *AUTHENTICODE_SIGNA-*

TURE_MISMATCH or *AUTHENTICODE_SIGNATURE_MISMATCH_OR_INCORRECT_IMAGEBASE*, depending on the flag *pe_rebuilt*. Note that by probability of collisions, it may happen that an image address value was selected such that the PE checksum is correct when rebuilding the PE file, but however, it was not the original image address value (in line 9 of Algorithm 1). Hence, the Authenticode hash computed would be surely different from the real one. When both Authenticode hashes match, the certificate chain c is verified (line 5) with the OpenSSL binary package. If the Authenticode header is missing (i.e., the offset value in the PE header is zero), it is checked if $h_p \in \mathcal{H}$, where $\mathcal{H}$ is the list of known hash values extracted from catalog files (line 14). Let us note that our tool needs a list $\mathcal{H}$ of known hash values. Our tool directly extracts these hash values from the catalog files stored in a given directory. If the hash is found, it returns *CATALOG_SIGNED*. Otherwise, it is checked if the file object f is located in the Windows directory or in any of its subdirectories. In that case, we assume that the file is cataloged-signed but $h_A \notin \mathcal{H}$, and the tool returns *MAYBE_CATALOG_SIGNED* (line 18). Otherwise, the tool returns either *NOT_SIGNED* or *NOT_SIGNED_OR_INCORRECT_IMAGEBASE*, depending of the value of the input flag *pe_rebuilt*.

Finally, let us describe the possible return values of the certificate chain verification process. After the verification process done by OpenSSL, our plugin may return the following: *CERT_EXPIRED*, when the certificate validity time passed; *CERT_UNTRUSTED*, when the certificate comes from an unknown publisher; *CERT_FORMAT_ERROR*, when there is an issue with the certificate structure as missing fields or malformed certificate; *CERT_VERIFICATION_SUCCESS*, when the certificate chain verification was successful. As explained above, currently OpenSSL does not verify the revocation of certificates. We are currently extending our plugin to include also this feature, adding *CERT_REVOKED* into the set of possible return values to report the user if any certificate of the chain is revoked.

To foster research in this area and enable the reproducibility of experiments, both the plugin `sigcheck` and the auxiliary tool `sigvalidator` have been released under GNU/GPL version 3 license and are freely available at `https://github.com/reversea-me/sigcheck`.

4. Experiments and Discussion

In this section, we detail the experiments performed with `sigcheck`. We have considered two different experiments. In the first experiment, we measure the effectiveness of `sigcheck` analyzing a set of memory dumps from different machines. In particular, we compute how many file objects are retrieved from memory dumps under different scenarios. The second experiment is focused on the analysis of signed malware samples.

4.1. *Effectiveness of* `sigcheck`

For experiments, we have considered a Windows 7 SP1 OS Enterprise edition Build 7601 in both 32 and 64 bits, running on top of VMware Workstation 15 Pro 15.0.2 build-10952284. Although we initially also considered Windows 8.1 and Windows 10, we discarded them from the experimentation since one of the Volatility plugins on which `sigcheck` relies (in particular, `dumpfiles`) was not working appropriately. We plan to investigate this particular problem further.

Apart from a freshly installed OS, we have installed additional software commonly used by users on both virtual machines, such as Mozilla Firefox 69.0.3, Google Chrome 77.0.3865.120, Libre Office 6.3.1, Notepad++ 7.8, and Adobe Reader DC 2019.012.20036. Every virtual machine was executed 10 times and to provide an effectiveness measure of `sigcheck` over time, the memory of these machines was acquired in four different time moments: at a fresh boot and after 10, 20, and 30 minutes of user activity (with data documents opened and viewing YouTube content with Mozilla Firefox). The processes captured in each of these dumps correspond to the ones related to the normal Windows OS activity plus the ones related to the additional software installed.

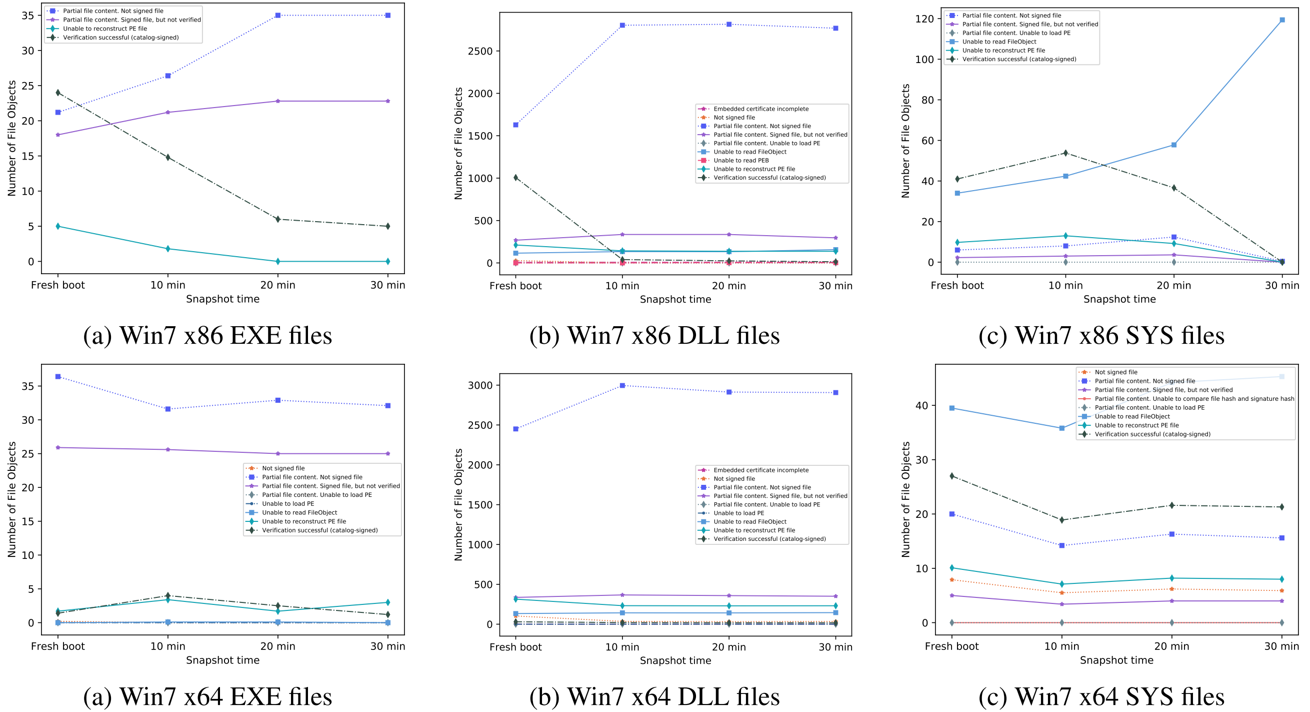
Figure 2 shows the average of `FileObject` types retrieved by `sigcheck` in 10 memory dumps of Windows 7 SP1 x86 (top figure) and Windows 7 SP1 x64 (bottom figure), obtained at different time moments. Recall that our plugin allows to verify the signature of executable files (extension `.exe`), dynamic-link library files (extension `.dll`), and driver files (extension `.sys`).

Discussion. The results show that there are more chances of retrieving file objects with complete data at fresh boot. As expected by the way of working of Windows' memory subsystem (page smearing, demand paging, and swap pages), the number of file objects with full content quickly drops as the time evolves. Furthermore, the results in a 32-bit OS are better than in 64-bit OS, for all scenarios. In Windows 7 x86, almost half of driver files are successfully verified as catalog-signed files, while executable and DLL files reach more than 30%. It is remarkable that the content of a huge percentage of file objects is partial, seeming not signed or with an incorrect image base address. In 64-bit DLL files, this percentage increases remarkably. Finally, let us also remark that none of the file objects retrieved in both scenarios contained the Authenticode signature as full content. Only 13 32-bit DLL files contained the certificate header, but it was incomplete due to memory paging issues.

4.2. *Analysis of Signed Malware Samples*

In this part, we selected a number of signed malware samples (Niemelä, 2010) from public repositories. Table 1 shows the SHA-256 of the selected samples for experimentation. When preparing this paper, we found a recent malware threat that was distributed also as a signed file (*malware04*). We also extract the most likely family names from VirusTotal reports to label every malware sample using the `AVClass` malware labeling tool (Sebastián et al., 2016).

We have analyzed every malware sample with `sigcheck` through the auxiliary tool `sigvalidator` (cf. Section 3, Algorithm 2). Table 2 collects the experiment results. For every



(a) Win7 x86 EXE files (b) Win7 x86 DLL files (c) Win7 x86 SYS files

(a) Win7 x64 EXE files (b) Win7 x64 DLL files (c) Win7 x64 SYS files

Figure 2: Average of `FileObject` types retrieved by `sigcheck` in 10 memory dumps of Windows 7 SP1 x86 (top figure) and Windows 7 SP1 x64 (bottom figure) obtained at different time moments.

Label	SHA-256 hash	AVClass result
malware01	`012503199e73d10c3f9f1b3db4876100bf1aebaea53c5da9b8b746a78cf65276`	`polycrypt`
malware02	`0da7bf4ee9884242fa2b413ecfd9b7efca2e5ccc9a87a308328c7e8279b24d20`	`perfectdefender`
malware03	`178c6f7ad9a2fa6b10e0df89f2d0893afab2215684fffb2ffa4db07f19e95833`	`trustedzone`
malware04	`2fc0512083ca44f2669815a8ce8fdcf1eaac63a282fbbc4c1c0892422816251f`	`megacortex`
malware05	`49dbf0e4f7aebb512fcd081ff5f1dbb2ec5ded98a3fbd5882e2c4c2f1a5c25e4`	`sobit`
malware06	`6910f947b942a0484097e0b95d54e49582801a93bf2ab62382f9ef45f8f2b5ac`	`pernefed`
malware07	`850c4554528c82d12a1a31f308f5b7a9b8d15eb453e65748dd5c65555ab6ba98`	`perfectdefender`
malware08	`a8bcc4f6564a96a4602584a06c5e17d0fab2f15a07a49c208657d6a37162ef24`	`geral`
malware09	`b057f90eac6a3427e95ffd8bc502659cad08206b0b560f30ab5db9e4b90bb9fd`	`perfectdefender`

Table 1: Signed malware samples used in the experiments.

malware sample, we show the problem of the certificate (*expired and revoked by issuer*; *only expired* or *only revoked by the issuer*; and *self-signed certificate*), the warning window shown to the user by the Windows UAC in a Windows 10 x64 Stable 1809 OS when executing the sample (from more trust to less trust: asking for permission to make changes, verified publisher [blue color]; asking for permission to make changes, unknown publisher [yellow color]; and execution blocked [red color]), and the results of execution of the SysInternal's `sigcheck` tool (Russinovich, 2019) and our plugin `sigcheck`. Let us remark that we have selected this version of Windows OS instead of Windows 7 (as in experimentation) since we are interested in the most updated messages of Windows UAC.

Discussion. The results indicate that both Windows UAC and SysInternal's `sigcheck` are more focused on the publisher trustfulness rather than other aspects of the certificate, such as the certificate validity time. We believe that apart from the publisher trustfulness, a certificate expired should always be reported, as our plugin does. Moreover, the messages shown by the Windows UAC are less intuitive for the users. The case of self-signed certificate (*malware08*) is detected by the three tools, although text messages differ slightly. The most interesting case is *malware04*, a malware threat associated to the ransomware MegaCortex (Brandt, 2019). Although SysInternal's `sigcheck` warns that the certificate was revoked by the issuer, surprisingly the Windows UAC tells the user that the file comes from a verified publisher. This difference may be caused by the parameter settings when calling to `WinVerifyTrust`. Similarly, our plugin also returns that the verification process was successful. Recall that as discussed in Section 3, `sigcheck` does not perform any certificate revoking checking, since it is not supported by the OpenSSL binary package on which our plugin relies. We are currently implementing the certificate re-

Sample	Real issue with the certificate	Windows UAC	SysInternal's Sigcheck	Plugin `sigcheck`
malware01	Certificate expired and revoked by issuer	Verified publisher (blue color)	Signed	Certificate expired
malware02	Certificate expired	Verified publisher (blue color)	Signed	Certificate expired
malware03	Certificate expired and revoked by issuer	Execution blocked (red color)	Certificate revoked by issuer	Certificate expired
malware04	Certificate revoked by issuer	Verified publisher (blue color)	Certificate revoked by issuer	Verification successful
malware05	Certificate expired	Verified publisher (blue color)	Signed	Certificate expired
malware06	Certificate expired	Verified publisher (blue color)	Signed	Certificate expired
malware07	Certificate expired	Verified publisher (blue color)	Signed	Certificate expired
malware08	Self-signed certificate	Unknown publisher (yellow color)	Root certificate untrusted (under current security policies)	Issuer certificate missing
malware09	Certificate expired	Verified publisher (blue color)	Signed	Certificate expired

Table 2: Experiments with signed malware samples.

vocation process using the OpenSSL library to fix this issue.

5. Limitations

This section reviews the main limitations imposed by memory forensics to the verification of digitally signed Windows PE files that we have identified in our study. We also discuss some possible solutions to these limitations.

Data incompleteness. Although during our experiments we have always found file objects corresponding to different mapped files, in most of the cases data were incomplete. The Windows memory manager swaps physical memory pages when it needs to use more memory than actually available in RAM. Apart from this, page smearing and demand paging are also obstacles to obtain an image file that exactly matches with its corresponding executable file. Page smearing refers to the inconsistency in the page tables and the content of physical memory pages due to changes during the acquisition process, while demand paging is an optimization mechanism used by the OS to load data into memory only when it is absolutely needed (Case and Richard, 2017). Note that the calculation of the Authenticode signature needs the full PE file, excluding only certain parts (cf. Section 2.1). Hence, if only one memory page of an image file is missed, the signature of that file cannot be computed correctly.

A possible solution to overcome this issue is to guarantee that the Windows PE loader locks memory pages that contain file objects (Russinovich et al., 2012). In this regard, it should always maintain an open reference to every mapped file while the corresponding process is on execution. Of course, this solution imposes a performance penalty, since physical memory pages are "wasted" containing data that are unused and thus the memory pages could be paged out or even freed. However, the memory of current computer systems is in the order of gigabytes and thus this penalty may be negligible. This is an open question that deserves further investigation. Other solution might be a combined analysis of a memory dump and their corresponding swap files (up to 16 paging files of different sizes in Windows OS (Russinovich et al., 2012)).

Data changes caused by PE relocation. As discussed in Section 2.2, a `DataSectionObject` is a perfect copy of the executable file when we eliminate the end padding added due to page alignment issues. However, an `ImageSectionObject` is created after the Windows PE loader makes the relocation process, and thus apart from the end padding there are also bytes that change, such as the image base address (located in the PE header) and every absolute address defined in the relocation section or other function tables of the executable file. This issue imposes two major limitations to our approach:

- In 64-bit files, there are a total of 2^{52} possibilities for an image address value (considering an offset of `0x1000` between memory addresses). Therefore, it is computationally infeasible to brute-force the image address. To overcome this limitation, our plugin considers the most common image base addresses of every version of Windows OS, empirically obtained. This implies that `sigcheck` may reconstruct the executable file and consider it as wrongly signed if a Windows update changes the image address of that file to a value not considered by `sigcheck`.
- To verify a successful reconstruction of the executable file, we have opted for using the value of the `CheckSum` attribute, located at the PE optional header. Although stored in a 32-bit value, it is in fact a 24 bit checksum (the higher nibble is discarded) computed using an additive checksum algorithm which serves as redundancy check. Thus, since the number of possible collisions is high ($1/2^{24}$), it may happen that `sigcheck` finds an image address value such that the PE checksum is correct, but the computed Auhtenticode hash value is invalid.

Furthermore, let us recall that the Authenticode signature data, which is contained in the security directory of the PE header, is not mapped into the `ImageSectionObject` structure. Therefore, if the executable file is an embedded Authenticode-signed file, using this internal structure we can only tell that the file is signed but its signature cannot be verified (since we are unable to retrieve the signature data from memory).

When the relocation data exist in the `ImageSectionObject` structure, we can easily revert the relocation process performed

by the Windows PE loader. However, since we are unaware of the original image base address of the executable file, it is difficult to generate a valid Authenticode signature. A solution that might overcome this problem is the use of a database storing the version information, Authenticode hash signature, and image address values for every system file of each Windows OS flavor (and associated updates). Of course, the database would quickly grow up given the vast amount of system files in all Windows OS flavors.

Catalog-signed files. As shown in Section 4, many system files in Windows are catalog-signed PE files. To verify every signature, we need the catalog database that contains that hash signature. This implies that we need to have every possible catalog file for every Windows OS flavor, as well as any of their updates. For instance, the database of catalogs that we used in our experiments is about 200MB, distributed over 8784 files. However, this database only contains data from the versions of Windows 7 and Windows 10 used in experimentation (cf. Section 4), as well as a version of Windows 8.1.

At the moment of this writing, there is not a centralized database of catalog data. State-sponsored CERTs (or university-related CERTs) could lead this initiative and maintain this database, providing a public web service to check the database for other incident response teams.

Executable file and process inconsistencies. Our plugin relies on the file object structure to verify the Authenticode signature. However, it may happen that although the mapped file is legitimate (i.e., the file contained in the device drive was not manipulated), its corresponding process is malicious. For instance, malware can inject malicious code in a legitimate process or perform a process hollowing (it occurs when a process is created in a suspended state and then its process memory is replaced with malicious code) to evade detection and analysis. Currently, our plugin does not detect this kind of evasion techniques. We have empirically tested that these evasion techniques are also undetected by other forensic programs that verify Authenticode signatures from running processes in live systems, such as `Process Explorer` and `Process Hacker`.

In our opinion, this is one of the biggest challenges in memory forensics. To be sure that the binary code of an image file was unmodified, we need to unmap its corresponding process, reverting all the work performed by the Windows PE loader, and then compare the unmapped image file with the executable file. At the moment of this writing, we are unaware of any tool capable of doing this backward process and comparison reliably. We could also check the virtual address descriptor (VAD) tree of a process, looking for strange changes in the permissions of every VAD node. VAD is a tree data structure used by the Windows OS to store information about process memory regions (Dolan-Gavitt, 2007). A similar idea is used in (Srivastava and Jones, 2017), where a Volatility plugin is introduced to locate injected code using cross-validation with stack information even when VAD nodes have been modified by the malware to remain stealthy. Of course, we can also rely on other Volatility plugins more focused on malware detection such as `malfind` or `clamav`, or more focused on process similarity as `ProcessFuzzyHash` (Rodríguez et al., 2018).

6. Related Work

There are works regarding the analysis and the presence of Authenticode-signed files in malware landscape, as well as highlighting the weaknesses of code signing in Windows.

One of the first works showing the abuse of software digital certificates and the presence of malware samples signed with rogue certificates was (Wood, 2010). The common pitfalls of certificate authorities that allow the prevalence of signed malware samples are also introduced in this paper. The presence of signed malware samples was also confirmed by (Kotzias et al., 2015), where a malware dataset of 356K samples from 2006 to 2015 was analyzed. They concluded that most signed samples in the dataset were potentially unwanted programs (between 88% – 95%), while only a few were signed samples of malware (5% – 12%).

(Kim et al., 2017) proposed a threat model which highlights some weaknesses in the use of code-signing technology, after detecting that signed malware may cause anti-virus (AVs) products to stop analyzing them. The highlighted weaknesses are related to inadequate client-side protections (such as the use of AVs that skip analysis of signed binaries), publisher-side key mismanagement (an inadequate security management in the publisher side may facilitate the stealing of private keys that corresponds to the publisher's certificate or the infection of developer machines), and CA-side verification failures (verification process may fail, resulting in certificates issued to shell companies or malicious publishers with stolen identities). They also analyzed a total of 325 signed malware samples, having a 58.2% of samples properly signed (the rest signed with malformed certificates) and classifying the certificate abuse as a compromised certificate or issued to a fraudulent or shell company.

In a similar work, (Kozák et al., 2018) studied the underground market of code signing certificates, identifying 4 leading vendors of Authenticode certificates. Furthermore, they collected 14,221 signed malware samples and analyzed the certificate attributes to graphically map the relationships among malware families and certificates.

The malware dataset focused on Windows built in (Ugarte-Pedrero et al., 2019) showed that close to 18% of the 172,612 collected samples are signed binaries, with very few (only 11 samples) using revoked certificates. Signed binaries are also collected and analyzed in Rivera et al. (2019), where 39% of 75,615 malware samples analyzed are signed – three out of four samples being correctly validated as Authenticode-signed files.

All of the aforementioned works focus on executable files instead on binary data extracted from memory dumps, as we do in this paper. To the best of our knowledge, we are the first to put a spotlight on the analysis of Authenticode-signed files extracted from memory dumps and to highlight the limitations and possible solutions.

7. Conclusions and Future Work

In this paper, we have investigated whether code signing brings any benefit to memory forensics. In particular, we have focused on Windows OS. However, due to how Windows works and the current limitations of memory forensics (such as paging memory, page smearing, and demand paging), the verification of digital signatures of processes captured in a memory dump is a difficult task. We have highlighted the limitations imposed by memory forensics (data incompleteness, data changes caused by relocation, catalog-signed files, and executable file and process inconsistencies), as well as proposed some solutions. Furthermore, we have developed `sigcheck`, a Volatility plugin that verifies a digitally-signed file (if feasible) retrieved from a memory dump through internal OS structures (in particular, file objects). We tested it on Windows 7 x86 and x64. Our findings indicate that the success rate is low, especially when the memory is acquired from a system that was running for a long time.

As future work, we aim to complete the verification process of our plugin with certificate revocation and to further study the lifetime of file objects. We also aim to explore the solutions proposed for the current limitations.

Acknowledgements

This work was supported in part by the Spanish Ministry of Science, Innovation and Universities under grant MEDRESE-RTI2018-098543-B-I00 and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-17R).

References

AV-TEST, 2019. Malware statistics. [Online; https://www.av-test.org/en/statistics/malware/].

Bhatkar, E., Duvarney, D.C., Sekar, R., 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits, in: Proceedings of the 12th USENIX Security Symposium, pp. 105–120.

Brandt, A., 2019. "MegaCortex" ransomware wants to be The One. [online; https://news.sophos.com/en-us/2019/05/03/megacortex-ransomware-wants-to-be-the-one/]. Accessed on September 30, 2019.

Case, A., Richard, G.G., 2017. Memory forensics: The path forward. Digital Investigation 20, 23–33. doi:10.1016/j.diin.2016.12.004. special Issue on Volatile Memory Analysis.

Dolan-Gavitt, B., 2007. The VAD tree: A process-eye view of physical memory. Digital Investigation 4, 62–64. doi:https://doi.org/10.1016/j.diin.2007.06.008.

Graeber, M., 2018. Subverting Trust in Windows. Technical Report. Specter Ops, Inc.

ITU-T, 1984. Introduction to ASN.1. [online; https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx]. Accesed on September 25, 2019.

Kaliski, B., 1998. PKCS #7: Cryptographic Message Syntax Version 1.5. [online; https://tools.ietf.org/html/rfc2315] ed. IETF Network Working Group. Accessed on September 25, 2019.

Kim, D., Kwon, B.J., Dumitraş, T., 2017. Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA. pp. 1435–1448. doi:10.1145/3133956.3133958.

Kotzias, P., Matic, S., Rivera, R., Caballero, J., 2015. Certified PUP: Abuse in Authenticode Code Signing, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA. pp. 465–478. doi:10.1145/2810103.2813665.

Kozák, K., Kwon, B.J., Kim, D., Gates, C., Dumitras, T., 2018. Issued for Abuse: Measuring the Underground Trade in Code Signing Certificate. CoRR abs/1803.02931.

Langner, R., 2011. Stuxnet: Dissecting a Cyberwarfare Weapon. IEEE Security & Privacy 9, 49–51. doi:10.1109/MSP.2011.67.

Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.

Microsoft Corporation, a. CertGetCertificateChain- function. [online; https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-certgetcertificatechain]. Accessed on September 26, 2019.

Microsoft Corporation, b. FILE_OBJECT structure. [online; https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ns-wdm-_file_object]. Accessed on September 30, 2019.

Microsoft Corporation, c. WIN_CERTIFICATE structure. [online; https://docs.microsoft.com/en-us/windows/win32/api/wintrust/ns-wintrust-win_certificate?redirectedfrom=MSDN]. Accessed on September 25, 2019.

Microsoft Corporation, d. WinVerifyTrust function. [online; https://docs.microsoft.com/en-us/windows/win32/api/wintrust/nf-wintrust-winverifytrust]. Accessed on September 26, 2019.

Microsoft Corporation, 2008. Windows Authenticode Portable Executable Signature Format. [online; http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_pe.docx]. Accessed on September 25, 2019.

Niemelä, J., 2010. It's Signed, Therefore It's Clean, Right?, in: CARO 2010 Technical Workshop.

Pagani, F., Fedorov, O., Balzarotti, D., 2019. Introducing the Temporal Dimension to Memory Forensics. ACM Trans. Priv. Secur. 22, 9:1–9:21. doi:10.1145/3310355.

Parno, B., McCune, J.M., Perrig, A., 2010. Bootstrapping Trust in Commodity Computers, in: 2010 IEEE Symposium on Security and Privacy, pp. 414–429. doi:10.1109/SP.2010.32.

PaX Team, . PaX address space layout randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

Rivera, R., Kotzias, P., Sudhodanan, A., Caballero, J., 2019. Costly freeware: a systematic analysis of abuse in download portals. IET Information Security 13, 27–35. doi:10.1049/iet-ifs.2017.0585.

Rodríguez, R.J., Martín-Pérez, M., Abadía, I., 2018. A Tool to Compute Approximation Matching between Windows Processes, in: Proceedings of the 2018 6th International Symposium on Digital Forensic and Security (ISDFS), pp. 313–318. URL: http://webdiis.unizar.es/~ricardo/files/papers/RMA-ISDFS-18.pdf, doi:10.1109/ISDFS.2018.8355372.

Russinovich, M., 2019. Sigcheck v2.73. [online; https://docs.microsoft.com/en-us/sysinternals/downloads/sigcheck]. Accessed on September 25, 2019.

Russinovich, M.E., Solomon, D.A., Ionescu, A., 2012. Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7. 6th ed., Microsoft Press, Redmond, WA, USA.

Sebastián, M., Rivera, R., Kotzias, P., Caballero, J., 2016. AVclass: A Tool for Massive Malware Labeling, in: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (Eds.), Research in Attacks, Intrusions, and Defenses, Springer International Publishing, Cham. pp. 230–253.

Srivastava, A., Jones, J.H., 2017. Detecting Code Injection by Cross-Validating Stack and VAD Information in Windows Physical Memory, in: 2017 IEEE Conference on Open Systems (ICOS), pp. 83–89.

Ugarte-Pedrero, X., Graziano, M., Balzarotti, D., 2019. A Close Look at a Daily Dataset of Malware Samples. ACM Trans. Priv. Secur. 22, 6:1–6:30. doi:10.1145/3291061.

Walters, A., Petroni, N., 2007. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process, in: BlackHat DC.

Wood, M., 2010. Want My Autograph? The Use and Abuse of Digital Signatures by Malware. Technical Report. Virus Bulletin Conference.