Integrity Verification of User Space Code

By

Andrew White, Bradley Schatz and Ernest Foo

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2013 USA

Monterey, CA (Aug 4th - 7th)

Digital Investigation S10 (2013) 59–S68

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Integrity verification of user space code

Andrew White*, Bradley Schatz, Ernest Foo

Queensland University of Technology, Brisbane, Australia

Keywords:
Memory forensics
User space
Windows XP
Windows 7
Malware analysis

A B S T R A C T

We present a novel approach for the construction and application of cryptographic hashes to user space memory for the purposes of verifying the provenance of code in memory images. Several key aspects of Windows behaviour which influence this process are examined in-depth. Our approach is implemented and evaluated on a selection of malware samples with user space components as well as a collection of common Windows applications. The results demonstrate that our approach is highly effective at reducing the amount of memory requiring manual analysis, highlighting the presence of malicious code in all the malware sampled.

1. Introduction

The proliferation of the success of memory forensics in detecting system compromise has led to an increase in the difficulty of performing the analysis itself. With the introduction of new techniques to detect malware and new malware techniques to subvert analysis, the act of investigating a memory image has evolved from what was once the simple application of a few Volatility (Volatile Systems, 2011) plugins into the successive application and comprehension of many. This newfound maturity has exposed memory forensics to the same issues encountered by other more established digital forensics disciplines, such as the complexity and volume problems.

For this reason, we are proposing the use of a reduction technique, similar to those used previously in disk forensics. Our reduction technique is capable verifying the provenance of the executable contents of user space memory such that all known code can be excluded from further analysis, an approach similar to that of the National Software Reference Library (NSRL) (National Institute of Standards and Technology, 2012) used in disk forensics. We apply cryptographic hashes of code from on disk in a manner that takes into consideration the unique challenges posed by code in memory, allowing the identification of known code. This approach was found to indicate the presence of the unknown user space code introduced by malware in all samples tested, highlighting the areas which require further analysis.

This paper proposes the following contributions. Firstly, a novel method of applying executable code hashes built from executable files to code in memory. Secondly, a detailed analysis of the idiosyncrasies in Windows XP and Windows 7 that would otherwise prevent the success of such an approach. Thirdly, an implementation of the above research as an open source tool, and lastly, an evaluation of this tool on both malware and common Windows applications. Only 32-bit versions of Windows are examined in all of these contributions.

The rest of this paper is structured as follows. Section 2 details the related research. Section 3 details the building and application of the code hashes. Section 4 explains the challenges to this approach introduced by typical Windows behaviour. Section 5 describes the implementation of the tool. Section 6 describes the setup of our experiments and Section 7 details the results of these experiments. Section 8 evaluates the strengths and weaknesses of this approach before the paper is concluded in Section 9.

2. Related work

Detecting malicious code in memory relies heavily on the rootkit paradox (Kornblum, 2006a); that as long as code wishes to execute on the system, it must be visible to the

* Corresponding author.
E-mail address: a13.white@qut.edu.au (A. White).

1742-2876/$ – see front matter © 2013 Andrew White, Bradley Schatz and Ernest Foo. Published by Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.diin.2013.06.007

system in some way. Such rootkit-like behaviour can be achieved under a few different generic approaches (Rutkowska, 2006). As such, the majority of memory analysis malware detection approaches rely on the location of these specific artifacts. An example of this can be seen in the Volatility Project (Volatile Systems, 2011), which contains plugins to locate specific artifacts such as Direct Kernel Object Manipulation in the list of running processes and loaded modules, or hooking call addresses in the Global Descriptor Table. While malfind (Ligh, 2012a), a more generic malware finding plugin does exist, it relies on a single permission artifact of crudely injected code. Code that correctly performs process hollowing and can circumvent malfind has been around longer than malfind itself (Keong, 2004).

The use of hashes to establish the provenance of code lying dormant on disk, such as Tripwire (Kim and Spafford, 1993) and the NSRL (National Institute of Standards and Technology, 2012), are quite common, whereas their application to code in memory is not. The difficulty of such an approach lies within the fact that code on disk and in memory are stored differently, and that various aspects of the code need to be updated upon loading into memory to accurately reflect the environment the code is executing in. A live response method of directly comparing code on disk to in memory exists (Rutkowska, 2005), however this requires trusting both the contents of the disk and operation of the system simultaneously, allowing changes in either to compromise the results of the tool.

A previous attempt to apply hashes of code to memory relied on the naive approach, using the on disk files to build an unmanageably large list of potential hashes which were blindly applied against the physical address space (Walters et al., 2008). Each page of memory would then be in a state of matching or not matching a hash, allowing only an estimate of how much of memory was devoted to code. Our approach improves on this approach by applying the hashes to pages of virtual memory while leveraging the metadata available in memory (White et al., 2012), allowing the correct contents of the page to be known in advance.

Hashes have also been used to verify code while running in memory, by reverting the code back to its state on disk and comparing stored hashes (Oerting et al., 2010). Our research differs by applying these hashes post-mortem instead of during execution, and by using hashes of the in memory state of the code, rather than converting the in memory code back to its format on disk.

Although hashing algorithms exist that are designed to handle changes, known as similarity hashes (Kornblum, 2006b; Roussev, 2010), such algorithms are not suitable for use in verifying in memory code. Code in memory will always differ to on disk, and with such an approach one cannot distinguish between legitimate and malicious changes to the in-memory code.

3. Approach

Our approach of verifying the integrity of in-memory code utilises collision-resistant cryptographic hashes to determine whether in-memory code is equivalent to code on disk, and this process is split into two distinct phases. The first is the way in which the hashes are created from code on disk, and the second is the application of these hashes to code in memory. We describe each of these steps in detail in this section, then examine the resistance of this approach to subversion. Although the approach used to build the hashes is similar to that used by Walters et al. (2008), we summarize this process here to assist in the comprehension of the remainder of the technique.

3.1. Building hashes

The Windows operating system uses the Portable Executable (PE) file format to store executable code. While code on disk and in memory are both stored according to the PE file format, their layouts differ. When on disk, a PE file is stored according to its physical layout, while in memory, a PE file is stored according to its virtual layout.

The Windows PE loader is responsible for converting a PE into its virtual layout for use in memory, and updating all code reference to reflect the run time state of the system. This involves processing internal pointers to reflect the location of where the PE file itself is loaded, known as relocations, and updating the Import Address Table (IAT), where the addresses of imported functions from other PE files are stored. These internal pointers and the IAT are instantiated with default absolute values, which are only correct if the PE file and all imported libraries are loaded at their preferred base address.

The advent of Address Space Layout Randomization, which intentionally randomizes the load address of PE files to increase the difficulty of exploitation, means that PE files are rarely loaded at their preferred base address. This causes the values of internal pointers and the IAT to require updating to reflect the current run time environment when loaded. As these values are dependent on the run time load address of other PEs, their values cannot be known when building the hashes from on disk. As such, the hashes must be constructed to take into account that parts of the code are not known until run time.

For this reason, the hashes are built with the relocations and IAT normalized with a constant value, and these locations saved, producing a tuple of normalized hash and list of normalized offsets. This allows hashes to be tested by taking the in-memory page and normalizing the same offsets before hashing. To deal with the possibility of paging, these hashes are created on a per page basis. This was the approach taken by Walters et al. (2008).

Our approach achieves this by recreating the Windows PE loader, transforming the PE into its virtual layout, and normalizing the appropriate offsets, allowing the creation of per page hashes for each PE file on disk. If any sections of the PE do not reach the end of a page, we fill the remainder with null bytes before hashing. This approach to building hashes was chosen as makes it simple to create a unique hash set for a machine, such that the introduction of any unknown code for that machine can be detected.

Building upon the existing approach, we also save other key information with this tuple, such as the filename of the PE, the offset of this page in the PE, and whether this page is in a section of the PE marked executable. This information allows the hashes to more effectively be applied to memory, as is described in the next section.

3.2. *Applying hashes*

Each process on Windows resides within its own virtual address space, isolating it from other processes. The user space portion of this virtual address space, where the programs code and data is stored, is described by the Virtual Address Descriptor (VAD) Tree. This VAD Tree contains metadata about the contents and roles of each memory allocation, such as file objects and the allocation's permissions (White et al., 2012). By using the permissions and file objects of these allocations, we can determine which of these allocations contain PE files which require verification.

Our approach leverages this metadata to cull the dataset to specific pages before hashing, such that we only hash pages known to contain code and know what code that page should contain in advance. This overcomes the two major weaknesses in the work of Walters et al. (2008), as now only one hash has to be tested on each page instead of the entire hash set, and any specific pages of code which fail hashing will be identified.

A diagram of our approach is shown in Fig. 1. The first step of applying the hashes to a memory image is to enumerate the list of processes running on the system, through one of the many available methods. Then, the layout of that processes address space is determined using the VAD Tree, and the location of all allocations are saved. Executable allocations are checked for any file objects backing them, and the names of these file objects are used to retrieve the hashes relevant for that allocation. The hashes are then applied, page by page, by replacing the mutable parts of that code page with a constant and comparing the resulting hash to the stored hash. Allocations that are not executable are checked to ensure they contain no executable pages. If such executable pages exist, they are highlighted for further analysis.

As applying the hash involves altering the contents of the page, there exists the possibility of hashes for pages with large amounts of modifications to incorrectly match the page being tested. To avoid this possibility, our approach utilises the filename and offset stored with each hash, in order to ensure that each page of memory is only ever tested against its correct hash value. This essentially makes the application of hashes a white-listing approach, matching known code and identifying unknown code for future analysis.

In addition to preventing incorrect matches, such an approach also has the benefit of reducing the number of hash comparisons required. Instead of each page requiring individual testing against every hash in the hash set, as with the approach of Walters et al. (2008), each page now only requires testing against a single hash value. This makes runtime dependent only on the memory image, rather than the size of the hash set and the memory image as with the existing approach.

3.3. *Potential for subversion*

Since in our approach we replace the contents of specific offsets with a constant, it would seem that these offsets would form an ideal place for malicious code to reside. Given that each of these locations represents a pointer that is unknown until run time, each location represents 4 bytes an attacker could potentially modify. The strategy that an attacker might employ to abuse this is dependent on the distribution of these locations, whether the numerous locations are non-consecutive or consecutive.

When there are numerous non-consecutive locations, an attacker is able to modify 4 byte pointers that are spread across the page without breaking the verification. If the attacker were to replace a pointer with a pointer to where malicious code is stored, the location of the malicious code would be flagged by our approach. If the pointer is replaced with a pointer to existing code or 4 bytes of instructions, an attacker could attempt to subvert the logic flow of a program. Achieving any useful functionality through this approach without crashing the program however would be difficult.

When there are numerous consecutive locations however, an attacker is able to insert arbitrary code. Such consecutive locations typically occur within the Import Address Table (IAT) of the PE, as it is a continuous run of pointer values that require updating. However, given that these pointers should point to valid addresses within other loaded libraries, it makes them easy to verify through import resolution. A Volatility plugin, *apihooks*, already exists with such functionality (Ligh, 2012a).

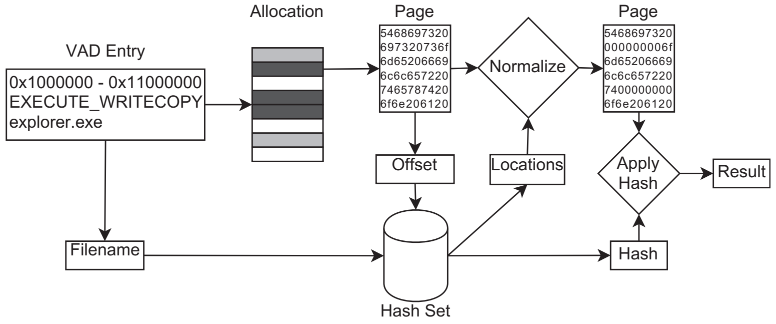

Fig. 1. The hash application process.

One complication that arises from the process of applying the hashes however is that an executable allocation containing a PE is capable of containing both code and data. This data, typically contained within the *.data* section of the PE, is data created and used by the program at run time, and generally cannot be known in advance. This prevents the verification of these pages, making them an ideal place for malicious code.

The introduction of the *Never eXecute* (NX) bit however increases the granularity of executable permissions to the page level. A single bit in the Page Table Entry of a page is used to determine whether execution is allowed, and the execution of non-executable pages is blocked at the hardware level. Windows has supported this feature since XP SP2 (Microsoft, 2006), which means that verification is only required for pages marked executable.

Of course, while this intended behaviour should match the implemented behaviour, this is not quite the case. Windows presents a number of idiosyncrasies that diverge from this behaviour, and these are detailed in the next section.

4. Windows-related challenges

While Windows has made great strides in increasing the security of its operating system, it was never built from the ground up with such security considerations in mind. This has resulted in a Windows system by default containing data allocations that are marked executable, and executable pages within non-executable allocations. As such pages are not backed by files on disk, they are not able to be verified via our approach. Since such allocations will occur by default in every Windows image however, we output these *unverifiable* allocations into a separate category, such that any non-default behaviour will become more evident.

The contents of the allocations in this section were determined through manual investigation. As an additional check, allocations suspected to only contain data were tested by removing their executable permissions on a running system, and observing the resulting state of the system. Since there may exist edge cases for which these allocations require executable permissions however, proving these allocations only ever contain data would require code coverage testing beyond the scope of this research.

The following is a comprehensive list of unverifiable allocations that occur by default on Windows XP and Windows 7 systems, organized by which operating system they affect. As each of these allocations are not able to be verified, they will require manual examination before their contents can be trusted.

4.1. Windows XP

Each process on Windows requires some default system-wide information in order to function, and Windows allocates shared memory, using *sections*, in order to achieve this. The issue however is that on Windows XP, although each of these default sections only contain data, they are marked executable by default. This gives an attacker numerous locations to hide malicious code which could then be executed from within any process.

The first of these executable shared allocations is the Read-Only Shared Heap, which is mapped read-only in each process and can be located through a pointer in the Process Environment Block (Ionescu, 2004). Although the permissions on this allocation are read-only by default, with sufficient privileges these permissions could be changed, allowing malicious code to be inserted.

The second of these executable shared allocations are the Desktop Heaps, each of which are responsible for storing the data required to draw the graphical objects of its corresponding desktop. To find the desktop heaps, one must first enumerate all the *tagDesktop* objects which each represent a desktop (Ligh, 2012b). Using these tagDesktop objects, one can then find the kernel address of the desktop heap using the *pHeapDesktop* pointer, and the user space address within a process by matching the *_SECTION_OBJECT* found using the *hsectionDesktop* pointer to a section backing an allocation. While these desktop heaps are mapped as read-only, since their contents can be modified through API calls, it is conceivable that malicious code could be inserted into the desktop heap from unprivileged code and executed.

The third and last executable shared allocation is one created by Win32k.sys, the driver that implements windowing and the desktop. This allocation is unusual in that it seems to be a composite of two data structures, the first $0 \times$ C0000 bytes appear to be a caching mechanism, while the last 0×8000 bytes are a heap, and although this heap appears to contain user interface data, it is unique from the desktop heaps. Unlike other section objects, the only reference to this section is found within the .data section of Win32k.sys, typically located at Win32k.sys$+0 \times$ 1AA6DC. While like the other shared allocations it is mapped read only, since it appears to contain desktop related data, it is possible that like the desktop heap malicious code could be inserted using standard Windows API calls.

Aside from the default executable sections, Windows XP also exhibits some default behaviour in two system processes that create unverifiable allocations. The first process is *winlogon.exe*, in which 9 executable and writable allocations are created that appear to only contain data. Each of these allocations are 4 pages long, and are sparsely populated. The locations of these allocations can be found in an array within the .data section of winlogon.exe, typically starting at winlogon.exe$+0 \times$ 72b0c.

The second process is *csrss.exe*, in which many pages in the range of $0 \times$ 00000000 – $0 \times$ 000fffff are marked executable but are within non-executable allocations. While the number of allocations and pages within this range vary depending on the system, they appear to contain boot related information such as the MBR and parts of the BIOS. Although mapped into memory, issues with imaging this memory means that much of this information is not accessible in memory images from physical machines. Although these allocations contain code, since this code does not come from PE files they are unable to be verified with our approach.

4.2. Windows 7

While Windows 7 has removed the executable permissions on the default shared allocations and the pages within winlogon.exe and csrss.exe, it has introduced a new set of issues specific to behaviour exhibited by individual PE files. The majority of these new issues are still the same case of allocations containing data being marked executable, however many of these allocations are also marked as writable.

The first of these executable allocations is caused by the use of *ole32.dll*, a library responsible for the support of Microsoft's Object Linking and Embedding (OLE) technology. While not present in every process that imports this library, the use of a particular feature results in the creation of a 2 page executable and writable allocation, containing data that appears to be consistent across invocations on the system. A pointer to this allocation can be found in the .data section of ole32.dll at ole32.dll+0 × 148A24 and ole32.dll+0 × 148A2C.

A second executable and writable allocation can be found in *searchfilterhost.exe*, part of the Windows Search service responsible for indexing files and providing fast file search results. This allocation contains a heap, that is typically for the most part unallocated. The cause of this heap is within *ntdll.dll*, and a pointer to the allocation can be found at ntdll.dll+0 × D7514.

The last executable and writable allocation is found within *explorer.exe*, the process responsible for displaying the desktop. This allocation is usually 1 page and typically contains an empty unknown linked list data structure. Unlike the other allocations, pointers to this allocation only exist within the stack and the heap, making them difficult to locate reliably. Since this allocation only appears within explorer.exe however, we can use this fact combined with the analysis of pointer values within the unknown linked list structure to locate this allocation.

In addition to these previously mentioned executable and writable data allocations, there exist four other allocations containing code that cannot be verified, as they are different in memory to on disk. The first of these is in *wmpnetwk.exe*, where *blackbox.dll*, a Windows DRM library, is loaded and contains different code pages in memory than on disk. This difference is caused by intentional obfuscation, as upon loading blackbox.dll it decrypts DRM-related resources which are stored encrypted on disk, likely in order to prevent reverse engineering.

The second of these unverifiable code allocations is the *shell32.dll* allocation within *searchindexer.exe*. Occasionally, in addition to containing shell32.dll, this allocation will also contain the PEs of *fdProxy.dll*, *pnidui.dll* and *WPDShServiceObj.dll*. While the reasons for such behaviour are unknown, by checking for the presence of PE headers at specific offsets these PEs can be located.

The last two allocations are also within *searchindexer.exe*, but differ in that they are non-executable allocations that contain executable pages. These allocations are 0 × 3f3000 and 0 × c0000 bytes in size, and when present, occur sequentially in memory. While the larger allocation contains the PEs of *vsstrace.dll.mui*, *ntprint.dll*, *searchindexer.exe* and *p2p.dll*, the smaller allocation contains no PE headers at all. Although the purpose of these allocations are not known, their unique sizes, combined with the fact the smaller allocation can be found at esent.dll+0 × 181C90, makes them easy to locate. As the exact contents of these allocations do not exist on disk, they are unable to be verified with our approach.

4.3. Common issues

A common issue within both Windows XP and Windows 7 is the presence of sections of a PE not marked executable, such as a .data section, appearing to be marked executable in memory. Such pages can be split into two categories, based on the type of PTE involved.

If the PTE is a valid PTE, this indicates that the permissions of these pages have been intentionally changed after being loaded into memory. Such pages commonly occur within the components related to DRM in Windows 7, such as blackbox.dll and wmdrmdev.dll within the address space of wmpnetwk.exe. The presence of these pages, termed *executable data* pages, indicates that a section containing data has been repurposed as code as part of some obfuscation technique. As such pages are unable to be verified, they are placed in the *unverifiable* category, and require manual analysis. These pages are identified by including the PE section permissions with the hashing information, and checking for when the permissions have changed.

When a PTE is not valid, defined as when bit 0 is cleared, this indicates that no valid physical address exists for that virtual address (Intel Corporation, 2013). However, when bit 11 is set and bit 10 is cleared, this indicates that it is in fact a *transition* PTE (Kornblum, 2007). Such PTEs are a special edge case defined by Windows, and still contain valid addressing information. However, such PTEs do not contain correct permission information, which is why these pages appear executable.

Retrieving the correct permissions for a transition PTE involves parsing the Page Frame Number (PFN) database, to find the corresponding entry for that PTE. When a PTE is in transition, the *OriginalPte* of the *_MMPFN* data structure of that entry takes the value of a *_MMPTE_TRANSITION*. This data structure has a *Protection* field, which contains the permissions that will be assigned to the PTE during the page fault that will occur upon access. As such, although these transition PTEs will always appear to be executable based on the NX bit, they may in fact not be executable upon access, allowing such non-executable pages to be excluded from the hashing process.

5. Implementation

The implementation of our tool was carried out in two parts, *hashbuild.py* and *hashtest.py*, which as their names indicate, build the hashes and test the hashes against the memory images respectively. A copy of our implementation can be found at https://github.com/a-white.

5.1. Hashbuild

Hashbuild is a standalone python script that walks through a filesystem and produces hashes for all PE files. This script takes two parameters, a filesystem, and the name of the file to output the hashes to. The filesystem is accepted as a mount point, allowing the use of disk images through FUSE or physically mounted disks, thereby supporting almost any disk format. We utilised the SHA1 hashing algorithm for our hashes due to its widespread availability, however this could be replaced with any other cryptographic hash function without issue.

Each PE file within the given filesystem is parsed through a virtual PE loader as described in Section 3. In our implementation, for each page in a PE file's virtual layout, we record the name of the PE on disk, the offset within the virtual layout, whether that particular part of the PE was marked as executable and the list of locations to normalize. Additionally, since relocations can potentially cross page boundaries, the list of locations to normalize also accepts negative offsets to allow relocations that continue from previous pages to be processed.

Our implementation only supports 32-bit Windows PEs, excluding MSDOS and NE versions of PE files. As we only implemented the loading process, our implementation does not deal with any intentional obfuscation techniques such as packing.

5.2. *Hashtest*

The application of the hashes was implemented as a plugin for the Volatility Memory Analysis Framework (Volatile Systems, 2011), allowing us to leverage existing implementations for basic functionality such as virtual address translation and process listing. Our implementation takes as options the hash file to use as the source of code hashes, and a location to dump any pages for which hash verification fails. Our plugin then follows the approach outlined in Section 3, taking into account all of the edge cases listed in Section 4.

In order to facilitate prioritization of what memory requires further analysis, we have separated the memory that is not verified into three categories. Pages for which the hash check fails are labeled as *failed* matches. Executable pages for which no hash information is available are labeled as *unknown*. Pages which cannot be verified due to standard Windows behaviour are labeled as *unverifiable*. While memory labeled with any of these categories still requires manual examination, it can be seen that memory labeled failed or unknown would be more likely to contain unknown code than memory labeled unverifiable.

An example of the output can be seen in Fig. 2. For each process in the memory image, a single summary line is displayed for that process. This summary contains the process ID, the number of allocations which were verified, unknown, unverifiable or failed verification, and the name of the process. If a process contained any allocations that were not verified for any reason, a summary of the offending allocation in question is displayed. This summary is the same as for the process, except it deals with pages, lists the allocation offset instead of the process ID and outputs any information regarding why the allocation was not verified.

An overall summary of the results is provided at the bottom, which gives the totals for the results at both the allocation and page levels. Additionally, it breaks down the unverifiable pages into two categories, those caused by PE data being marked executable, and those caused by default Windows behaviour. If more detail is required, the results for every allocation within every process can be made to output instead.

6. Experimental setup

We created four datasets of the purposes of our experiments. Each dataset consisted of a clean install of an operating system, and a collection of memory images after running various executables. For each operating system tested, Windows XP SP3 32-bit and Windows 7 SP1 32-bit, one dataset consisting of memory samples containing malicious code and one with memory samples containing common Windows programs were created. These datasets were created using a virtual machine, so each memory image is an atomic snapshot of the contents of memory at that point in time. The hard disk images of the virtual machines were included with the datasets, for the purposes of building the hashes required for verification.

The creation of the malware dataset involved creating an uninfected baseline on a virtual machine, and then infecting this baseline with a piece of malware, while directing all network traffic to a honeypot to prevent any unwanted activity. For each piece of malware examined, the virtual machine was restored to the uninfected

```
PID          Verified  Failed  Unverifiable  Unknown   Name
-----        --------  ------  ------------  -------   ------
00004               1       0             0        0   System
00268               3       0             0        0   smss.exe
00372              17       0             0        0   csrss.exe
                                ...
00764              85       0             1        0   svchost.exe
    01110000        0       0             2        0   ole32.dll executable alloc (Unverifiable)
                                ...
02376             100       0             6        0   wmpnetwk.exe
    003a0000        0       0             2        0   ole32.dll executable alloc (Unverifiable)
    6cd00000       47       0            11        0   msmpge2enc.dll (Executable Data)
    6ced0000      103       0            20        0   blackbox.dll (Unverifiable / Executable Data)
    6de80000      165       0            11        0   drmv2clt.dll (Executable Data)
    6dfa0000       57       0            11        0   wmdrmdev.dll (Executable Data)
                                ...
Totals
    Allocations  2076       0             7        0
    Pages       38788       0            57        0

Unverifiable Pages Breakdown
    43 Executable Data
    14 Default Windows Behaviour
```

Fig. 2. Sample output of hash testing process.

baseline, the malware sample executed, the machine rebooted and then a memory image taken, with a few minutes of idling in between each step to ensure the malware executed correctly. While the use of pausing a virtual machine to image memory potentially causes the loss of network related information, such information was not required for our approach. Lastly, each memory sample generated was manually examined prior to analysis with our tool, such that we could ensure the malware did in fact execute correctly and persist through the reboot process.

For the creation of the common program datasets, a similar process was taken, except that all programs were installed in advance, and the machine was not rebooted after execution. Additionally, to facilitate the use of some programs, normal internet connectivity was provided.

To run the experiments, a hash file was created from the hard disk image accompanying each data set using hashbuild, and then that hash file used in verifying the code in each memory image using hashtest. The results of these experiments are shown in Section 7. For the malware datasets, the tool was run against the entire memory image for each sample. Since in the application dataset however we were testing individual applications, the tool was only run against processes from that application.

In terms of the selection of malware and programs used in these datasets, two different approaches were taken. For the malware, selection was determined by the use of user space components, and the ability for the malware to execute correctly on both Windows XP and Windows 7. While numerous other malware samples were tested, only those included in the results were found to fulfill both of these criteria. As for the common program selection, no reputable sources of program usage were available, so what the authors believed what be the most common programs under a variety of categories were used.

7. Results

The results of our experiments are outlined in this section, with one set of results for each dataset.

7.1. Malware

Figs. 3 and 4 show the results of running the tool against the Windows XP and Windows 7 malware datasets respectively. For these datasets, each entry in the results shows the result of running the tool against the entire memory image containing the sample. This allows the impact of the introduction of the malware on the system to be seen.

The *Executable Pages* column shows the number of memory pages marked executable that were subject to the verification process, and the *Pages Failed* column shows the number of these pages that failed verification. The successfully verified pages as a percentage is shown in the next column. The remaining three columns show information about memory that was not subject to the verification process. In the *Executable Data* column the number of pages from data sections of the PE that were later marked executable, and as such, were not able to be verified, are displayed. The *Unverifiable Allocations* column shows the number of allocations that contained executable pages but were known to be unverifiable due to being caused by standard Windows behaviour. The last column, *Unknown Allocations*, displays the allocations containing executable pages that were not able to be verified due to no hash information being available.

For Windows XP, Fig. 3 shows that without any malware, the system typically has a 100% match of executable code, with 25 unverifiable allocations caused by default Windows behaviour. By comparing the infected samples to this baseline, the effect the malware had on the system can be seen. Each infected sample introduced unknown allocations, and additionally some also altered the code in existing allocations.

Windows 7, as shown in Fig. 4, on an uninfected system has a 100% match of executable code, with 43 executable data pages and 7 unverifiable allocations caused by default Windows behaviour. As with Windows XP, each malware sample introduced unknown allocations, and most also altered existing allocations. This allows the effects of the malware samples to be easily identified.

When comparing the results on Windows XP and Windows 7, one can see that default Windows behaviour causes more unverifiable allocations on Windows XP, and more executable data pages on Windows 7. Across all the samples, the number of executable data pages and unverifiable allocations remained consistent with slight fluctuations, demonstrating that they indicate typical Windows behaviour. For the same sample on different versions of Windows, most samples reported consistent results. Some malware however, such as NGRBot and Spyeye, reported

Program	Executable Pages	Pages Failed	Pages Verified	Executable Data	Unverifiable Allocations	Unknown Allocations
No Sample	18701	0	100.00%	0	25	0
Cridex.B	18808	38	99.80%	0	25	4
Cridex.E	16964	28	99.83%	0	25	3
Dexter	37506	0	100.00%	0	25	2
NGRBot	19700	332	98.31%	0	25	44
Shylock	19583	30	99.85%	0	25	7
Spyeye	18564	107	99.42%	0	25	23
TDL3	19719	14	99.93%	0	25	49
TDL4	19911	14	99.93%	0	25	52
Vobfus	18322	0	100.00%	0	25	3
ZeroAccess	19644	0	100.00%	0	25	10

Fig. 3. Windows XP SP2 Malware Dataset Results.

Program	Executable Pages	Pages Failed	Pages Verified	Executable Data	Unverifiable Allocations	Unknown Allocations
No Sample	38463	0	100.00%	43	7	0
Cridex.B	38607	27	99.93%	43	7	2
Cridex.E	39080	27	99.93%	43	7	4
Dexter	48366	0	100.00%	43	7	2
NGRBot	36799	60	99.84%	43	6	8
Shylock	40058	35	99.91%	43	7	8
Spyeye	38970	24	99.94%	44	7	7
TDL3	39063	12	99.97%	43	8	59
TDL4	41225	12	99.97%	43	9	59
Vobfus	38090	2	99.99%	43	7	2
Zeroaccess	40751	0	100.00%	43	9	16

Fig. 4. Windows 7 SP1 Malware Dataset Results.

large discrepancies in results between the two versions of Windows, indicating that they employ different techniques depending on the operating system. Dexter caused larger numbers of executable pages to be in memory, which given that it scans for credit card data, suggests that this scanning activity somehow forced more executable pages into memory.

7.2. Common applications

Figs. 5 and 6 show the results from running the tool against the Windows XP and the Windows 7 application datasets respectively. For these datasets, each entry in the results shows the result of running the tool against the processes created as a result of running that program. This allows the impact of that program on the results of the tool to be seen.

The results from Windows XP shown in Fig. 5 indicates that numerous programs exhibit behaviour that complicated the verification process. Of a total of 20 programs, only 6 can be considered to have been verified completely. All of the 14 remaining applications introduced 1 or more unknown allocations. In addition, some of these applications introduced changes into their own code upon loading, making some pages fail verification. Each application from the Microsoft Office suite exhibited this behaviour, and is likely a remanent of their anti-piracy measures. Skype and Powerpoint were the most prolific in this regard, having low page verification rates and high numbers of executable data pages. This would indicate that the programs are packed when on disk, to help prevent reverse-engineering of their proprietary formats.

For Windows 7, the results in Fig. 6 show that the number of applications that were verified were the same. The overall results between Windows XP and Windows 7 are similar, except that the results from Windows 7 have far fewer unverifiable allocations. One interesting trend however is that some applications that under Windows XP have no executable data pages have executable data pages under Windows 7.

8. Discussion

For the results outlined in the previous section, it can be seen that our research was able to detect the introduction of running malicious code to the system in all examined cases. As can be seen from the application dataset however, the introduction of other pieces of software can potentially add some noise to this process, increasing the amount of memory requiring further examination. This amount of memory that requires further examination is only a fraction of the original memory however, making our approach

Program	Executable Pages	Pages Failed	Pages Verified	Executable Data	Unverifiable Allocations	Unknown Allocations
7zip	583	0	100.00%	0	3	0
Adobe Reader	3478	42	98.79%	0	4	11
Chrome	11370	6	99.95%	0	5	25
Excel	1883	6	99.68%	0	3	1
Firefox	5606	5	99.91%	0	3	5
Google Talk	1560	0	100.00%	0	3	1
Internet Explorer	3151	0	100.00%	0	3	1
iTunes	13869	0	100.00%	0	3	0
Notepad++	1288	0	100.00%	0	3	0
Outlook	4148	12	99.71%	1	3	3
Pidgin	2334	0	100.00%	0	3	0
Powerpoint	2276	519	77.20%	1192	3	3
Skype	6571	4216	35.84%	262	3	2
Thunderbird	5651	6	99.89%	0	3	7
VLC	1645	0	100.00%	0	3	0
Winamp	2626	0	100.00%	0	3	18
Windows Media Player	833	0	100.00%	0	3	1
Winrar	835	0	100.00%	0	3	11
Wordpad	899	0	100.00%	0	3	0
Word	3113	8	99.74%	0	3	1

Fig. 5. Windows XP SP3 Application Dataset Results.

Program	Executable Pages	Pages Failed	Pages Verified	Executable Data	Unverifiable Allocations	Unknown Allocations
7zip	583	0	100.00%	0	0	0
Adobe Reader	3478	42	98.79%	0	0	17
Chrome	10867	9	99.92%	32	0	25
Excel	2419	6	99.75%	0	0	2
Firefox	4480	5	99.89%	0	0	5
Google Talk	2951	0	100.00%	0	0	0
Internet Explorer	3794	27	99.29%	0	1	1
iTunes	5991	0	100.00%	11	0	0
Notepad++	1651	0	100.00%	0	0	0
Outlook	6981	11	99.84%	1	0	4
Pidgin	2720	0	100.00%	0	0	0
Powerpoint	3558	2023	43.14%	972	0	10
Skype	7320	4216	42.40%	262	0	2
Thunderbird	4247	5	99.88%	0	0	5
VLC	2073	0	100.00%	0	0	0
Winamp	3810	0	100.00%	0	0	18
Windows Media Player	3160	1	99.97%	0	0	1
Winrar	1457	0	100.00%	11	0	11
Wordpad	1545	0	100.00%	0	0	1
Word	3403	9	99.74%	0	0	2

Fig. 6. Windows 7 SP1 Application Dataset Results.

highly suitable as a reduction technique to direct further analysis.

From the malware results, it can be seen that all samples examined were able to be detected due to the fact that the code must exist in memory somewhere in order to execute. As discussed in Section 3, even if the pointers are maliciously altered, they have to point somewhere else to where the code resides. If malicious code is attempted to be entered in place of the pointers, it would have to somehow perform its own malicious tasks while at the same time not crashing the original process, making such an attack unlikely.

One possible method of circumvention would be what is considered a "Type-2" rootkit (Rutkowska, 2006), that is, a rootkit which only modifies data structures. Such a rootkit would under our approach be able to freely modify pointers and other data without detection. However, the malicious payload of such a rootkit would still have to exist somewhere in memory, meaning the rootkit would need to be able hide its executable memory. Another possibility would be a piece of malware that is completely implemented using Return-Orientated Programming (ROP), as this would mean the malware's "code" only ever exists as data, however such malware has yet to be seen.

If the malware is not concerned with allowing the process it inhabits to continue running however, a PE file's Import Address Table (IAT) provides an ideal place to conceal malicious code. This is due to the fact that the IAT is typically a continuous run of pointer values which are changed at run time, making them unverifiable under our approach. Tools that can correctly validate the IAT however already exist (Ligh, 2012a), and should be used alongside our research.

While it would be possible to investigate the unknown allocations caused by the applications examined, every application likely causes such allocations in its own unique way. For example, many of the unknown allocations caused by Adobe Reader are due to ".api" plugin files being loaded. These plugins are in fact simply PE files with a different extension, making it trivial to include them in the hashset and verify them correctly. However, there are simply too many applications to attempt to support in this manner. For this reason, we chose to only support such special considerations for built-in default Windows behaviour, which will apply to every system being examined.

Our approach suffers from two main limitations that hamper its effectiveness. The first is that it only scans user space memory, meaning that any malware which hides in the kernel space via a driver or similar cannot be detected under our approach. While a similar approach could be applied to kernel memory, it would require the mapping of all locations in kernel memory in which code could potentially execute, as has been done for user space memory, which is beyond the scope of this research.

The other limitation in our approach is that it relies on being able to accurately replicate what the code will look like in memory for hashing purposes. This means that any PE files that have been packed or obfuscated will not be able to be verified correctly, nor any PE files that are simply wrappers around interpreted code. Being able to successfully overcome such issues would require the implementation of a virtual machine capable of loading and running such PE files, or a system of building hashes of the code from in memory rather than from disk.

9. Conclusion

Our research has provided a novel approach to validating the contents of in-memory code. We have developed an approach for building and applying hashes to validate the contents of user space memory, described methods of overcoming the numerous complications to this approach provided by Windows, and demonstrated that it is highly effective at reducing the amount of memory that requires analysis. Our implementation of the aforementioned approach has been made available, and has been shown to be capable of detecting malware introduced into memory images.

For future work, we plan to investigate the applicability of a similar approach to verifying the contents of kernel memory, as well as extend our current approach to more versions of the Windows operating system. Alternative hash building methods to improve the handling of obfuscated files will also be investigated.

References

Intel Corporation. Intel 64 and IA-32 architectures software developer manuals, In: http://www.intel.com/products/processor/manuals/index.htm; 2013.

Ionescu A. Introduction to NT internals. http://www.alex-ionescu.com/part1.pdf; 2004.

Keong TC. Dynamic forking of Win32 EXE. http://www.security.org.sg/code/loadexe.html; 2004.

Kim G, Spafford E. The design and implementation of tripwire: a file system integrity checker. In: Proceedings of the 2nd ACM conference on computer and communications security 1993.

Kornblum J. Exploiting the Rootkit Paradox with Windows memory analysis. International Journal of Digital Evidence 2006a;5(1):1–5.

Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digital Investigation 2006b;3:91–7.

Kornblum J. Using every part of the buffalo in Windows memory analysis. Digital Investigation 2007;4(1):24–9.

Ligh MH. Malfind plugin. http://code.google.com/p/volatility/wiki/CommandReferenceMal22; 2012a.

Ligh MH. MoVP 1.3 desktops, heaps, and Ransomware. http://volatility-labs.blogspot.com.au/2012/09/movp-13-desktops-heaps-and-ransomware.html; 2012b.

Microsoft. A detailed description of the data execution prevention (DEP) feature. http://support.microsoft.com/kb/875352; 2006.

National Institute of Standards and Technology. National software reference library. http://www.nsrl.nist.gov/; 2012.

Oerting T, Lafornara P, Oliver R, Brender S, Marr M. Portion-level in-memory module authentication 2010.

Roussev V. Data fingerprinting with similarity digests. Advances in Digital Forensics VI 2010:207–26.

Rutkowska J. System virginity verifier. In: Hack in the Box security Conference 2005.

Rutkowska J. Rootkit hunting vs. compromise detection. Black Hat Federal; 2006.

Volatile Systems. The volatility framework. http://code.google.com/p/volatility/; 2011.

Walters A, Matheny B, White D. Using hashing to improve volatile memory forensic analysis. In: American Acadaemy of forensic sciences annual meeting 2008.

White A, Schatz B, Foo E. Surveying the user space through user allocations. Digital Investigation 2012;9:S3–12.