Membrane: A Posteriori Detection of Malicious Code Loading by Memory Paging Analysis

Gábor Pék, Zsombor Lázár, Zoltán Várnagy,
Márk Félegyházi, and Levente Buttyán

CrySyS Lab, Budapest University of Technology and Economics, Hungary

Abstract. In this paper, we design and implement Membrane, a memory forensics tool to detect code loading behavior by stealthy malware. Instead of trying to detect the code loading itself, we focus on the changes it causes on the memory paging of the Windows operating system. As our method focuses on the anomalies caused by code loading, we are able to detect a wide range of code loading techniques. Our results indicate that we can detect code loading malware behavior with 86-98% success in most cases, including advanced targeted attacks. Our method is generic enough and hence could significantly raise the bar for attackers to remain stealthy and persist for an extended period of time.

Keywords: Code loading, memory paging, Windows, memory forensics

1 Introduction

Recent years' targeted attack have shown that even the most advanced systems can be compromised. Some of these targeted attacks used sophisticated intrusion techniques [3] and others were quite simple [1]. These malware attacks typically employed a sequence of steps to compromise a target system. A majority of these malware codes have information gathering and information stealing capabilities. Once they have access to their target, these malware codes typically perform a number of operations to cover their traces and remain undetected. Intuitively, the more the malware can persist in the target system, the more information it can collect. Often, the attacks persist for years in the target systems and the attackers get access to a substantial amount of confidential information (as reported for example in [11]). It is reasonable to assume that these long-term operations leave a noticeable trace, yet many examples show that the complexity and rich features of contemporary operating systems leaves ample space for the attackers to hide their operation.

Code loading is one of the key techniques that malware employs to achieve persistence. Code loading happens when the malware adds to or replaces the functionality of existing code to execute its own components. It is typically possible as, for example, the Windows operating systems provides various methods (e.g., legitimate API functions, registry entries) to support this. Thus, code loading usually exploits the conditions given by a legitimate process. That is

why code loading is used to achieve evasion of detection or bypass restrictions enforced on a process level.

There has been efforts to develop various memory and disk forensics techniques to pinpoint system anomalies caused by such malware infections [5]. One of the biggest problem with current memory forensics techniques is that they only utilize memory locations that were actively used by the OS at the moment of acquisition. That is, important information about code loading can be lost if the malware or part of it was inactive when the memory was grabbed. Furthermore, the rich feature set of Windows allows miscreants to build unique code loading techniques (e.g., Flame's code loading mechanism) that evades signature-based protections.

In this paper, we explore the realm of Windows memory management, systematically identify key paging states and build upon the details of these paging states to detect malicious behavior after a successful system compromise. We design and implement *Membrane*, an anomaly-based memory forensics tool to detect code loading attacks by malicious software. Membrane is based on the popular memory forensics framework Volatility [18]. Membrane performs detection by analyzing the number of memory paging states. Our approach is different from approaches in related work because we focus on detecting anomalies (symptoms) concerning paging states of malware code loading behavior instead of the code loading actions themselves. While code loading actions can only be pinpointed by live analysis, our approach focuses on the consequences of these actions that may persist in the system for an extended period of time.

Our cardinality-based analysis is advantageous for several reasons: (i) it is generic, (ii) this technique is less researched and can provide new insights into malware detection, (iii) it can be used in combination with other detection techniques to maximize the confidence of detection, and (iv) arguably, evasion against this method is difficult (see Sect. 6). These characteristics allow us to provide a solution that can detect a wide variety of code loading attacks. This could significantly raise the bar for miscreants to implement memory-resident and stealthy malware attacks.

This paper is organized as follows. In Sect. 2, we give an overview of code loading techniques employed by malware and mitigation techniques designed against code loading attacks. Sect. 3 gives a short motivation and a high-level overview about our memory paging based approach to detect the traces of malware code loading after infection. In Sect. 4, we present Membrane and give details of our methodology including the implementation of Membrane. Section 5 shows the efficiency of Membrane to detect a posteriori code loading attacks. We discuss evasion techniques and other concerns in Sect. 6. Finally, Sect. 7 summarizes our work.

2 Background and Related Work

2.1 Code Loading Techniques and Types

Malware can use a wide range of means to take control over a system and hide its presence. Due to its versatility, code loading is commonly used by contemporary malware in spite of the effort that has been spent on developing defenses. Recent advanced targeted attacks, most prominently infostealers, still used this technique to surpass defenses and ensure an extended operation. To achieve this goal, the malware employs more complex code loading techniques that go beyond the simple use of operating system functions (e.g., the Flame malware described in 3.1).

First and quite surprisingly, Windows provides a wide range of user mode (e.g., `VirtualAllocEx`, `WriteProcessMemory`) and kernel mode functions (e.g., `KeAttachProcess`, `ZwAllocateVirtualMemory`) that can be used to inject code into processes. DLL loading can be achieved by using, for example `VirtualAllocEx` and `WriteProcessMemory` to allocate *private* memory in a target process and to store the name of the DLL to load. Then, `CreateRemoteThread` is called to start a new thread in the virtual address space of the target process which loads the corresponding DLL.

The *detours library* was implemented by Microsoft back in 1999 which provides a rich feature set to load DLL into running processes or program binaries. Miscreants can also insert various DLL paths into the *AppInit_DLLs* registry value, which forces newly created processes to load those DLLs. *DLL preloading* is another notorious technique, which enforces legitimate binaries to load illegitimate DLLs when the program starts. Another technique is called *process replacement* where the malware launches a legitimate process, but replaces its memory content when the image file is mapped into memory. For completeness, we mention *pure kernel rootkits* which load a new kernel driver or hijack an existing one. They operate without using any user-space component. Writing reliable kernel rootkits is challenging, however, as it is difficult to control simple functionalities from the kernel space such as network communication. Another challenge for malware writers is that the kernel is quite volatile and this requires that the kernel space malware is well-written. Additionally, recent Windows operating systems enforce code signing for kernel drivers, whose evasion requires an extra effort from attackers (e.g., Flame, Duqu [2]).

In this work, we focus on detecting code loading techniques in the user space. As discussed above, malware manipulating only the kernel functionality requires more substantial effort from attackers. We note, however, that our method does detect kernel-space malware with user-space components.

2.2 Mitigating Code Loading Attacks

Windows implements various protection mechanisms, for example, a private virtual address space for processes to mitigate code loading attacks. But, it also allows to use simple API functions to circumvent such protections. For instance

`WriteProcessMemory` can write into a target process when called with appropriate privileges. As `WriteProcessMemory` invokes kernel mode code in the context of the target process no address space restriction is violated. In this section, we present various techniques preventing and detecting code loading attacks and discuss their merits and limitations.

Prevention. Various preventive methods have been designed in recent years to thwart the loading of illegitimate codes. These techniques include the use of whitelisting, checksums or enforcing signed code loading. Unfortunately all of these techniques can be circumvented by determined attackers. An example for this is Flame, which created fake, but valid certificates for its component to deceit Windows.

Recent targeted attacks have demonstrated that preventive approaches might not be able to block attackers. Thus, we assume that attackers succeed in compromising the target system and focus on detecting their activity.

Detection. There are various approaches suggested over the years to detect the integrity violations of operating system structures [7,17] using either virtualization or a new architecture design [13]. In memory forensics, integrity protection of user-space code using cryptographic hashes has been proposed by White *et al.* [19] to detect in-memory code placed by malware. The proposed hashing algorithm matches the in-memory code with its binary counterpart. The problem with integrity checking is that dynamically allocated memory locations (e.g., heap) cannot be verified in this way due to its alternating nature. Srivastava and Giffin [17] design and develop a hypervisor-based system using virtual machine introspection, Pyrenée. This aggregates and correlates information from sensors at the network level, the network-to-host boundary, and the host level to identify the true origin of malicious behavior. While Pyrenée's host-based sensor checks code loading mechanisms from the perspective of processes, Membrane is a system-centric approach which makes our detection mechanism more generic. For this reason, Membrane includes the detection of widely-used code loading techniques such as AppInit_DLLs or DLL preloading previously overlooked by related work.

The Volatility memory forensics tool [18] offers the *malfind* plugin to find malicious or hidden code segments from memory dumps. Malfind crawls process VADs (see Sect. 3.2) and looks for entries with suspicious protection bits (i.e., RWX) and type. However, these permission bits can be manipulated by the malware at a later point of time, which may turn such solutions ineffective. In contrast to malfind which checks only VAD entries, Membrane tracks per process memory paging modifications from the perspective of the OS. Various free anti-rootkit tools are also available to hunt for hidden threads and processes. Unfortunately, *none* of the tested 33 anti-rootkit tools could detect hidden processes or threads in case of Flame [2]. Thus, we need another approach which tackles the problem of code loading from another aspect. We believe, that Membrane can be successfully used in combination with previous solutions to build more effective detectors.

3 Approach

3.1 Motivation: the Case Study of Flame

For motivation, we quickly show a case study about the Flame [2] targeted attack. Flame employs an entirely unique, but sophisticated thread injection method to hide its malicious activity via a chain of system processes. This unique technique *allows Flame to completely mimic the behavior of a normal Windows update process* by the runtime patching of `services.exe`, `explorer.exe` and `iexplore.exe`. As Flame uses `iexplore.exe` for network connection [3], it can evade many malware scanners by default.

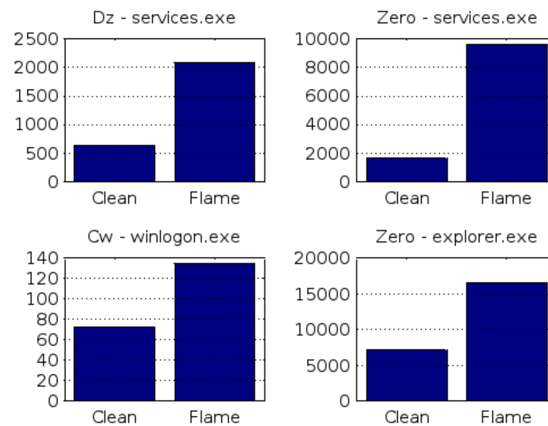


Fig. 1: Cardinality of different paging states in `services.exe`,`winlogon.exe` and `explorer.exe` on a clean and a Flame-infected 32-bit Windows 7 system.

For demonstration purposes, we infected a 32-bit Windows 7 machine with Flame malware to see what kind of page type modifications it causes. Surprisingly, the fine-grained process injection method it used to stay silent, entirely distorts the normal memory usage of system processes. Figure 1 shows that the number of certain page types (see Sect. 4.2 for more details) increases significantly when Flame is present.

This observation started us on a quest to use memory usage anomalies to detect code loading attacks.

3.2 Windows Memory Management

In order to understand, how different code loading attacks modify the state of memory, we introduce some important details about Windows' internal memory management. This will lead us to create our code loading detection technique in Sect. 4.

The virtual memory management unit of an operating system offers a transparent access method to physical addresses located in the host physical memory or the hard disk (e.g., in a pagefile). For Windows operating systems [16], this functionality is maintained by the hardware Memory Management Unit (i.e., *hardware MMU*) and a proprietary memory management kernel component which is called *memory manager* in [16]. In this work, we refer Windows' internal memory manager as the software Memory Management Unit (i.e., *software MMU*). The hardware MMU uses multi-level translation tables (i.e., page tables) to resolve virtual memory addresses pointing to pages which are loaded into the physical memory (i.e., valid pages). The software MMU, however, is used for invalid pages (e.g., memory is allocated, but never used), thus extra operation needs to be performed by the OS to, for example, bring them into the physical memory. In other words, Windows defers to build hardware page tables, until a page is first accessed by a thread. This approach of the OS is also called *demand paging*.

Hardware Supported Memory Management. When a virtual address points to a valid page, the hardware MMU uses multi-level translation tables, where the number of levels is influenced by the architecture. For the clarity of presentation, in this section we only discuss the 32-bit non-PAE case where each process owns a single page directory and multiple page tables. When a context switch occurs, the OS consults a kernel structure (i.e., `KPROCESS`) to retrieve the physical address of the process page directory. This address is loaded into a privileged CPU register called `CR3`, which allows the hardware MMU and software MMU to start the translation for process private virtual addresses. *Page directory entries (PDEs)* store the state and location of all the page tables belonging to a process. Due to Windows' lazy memory management, page-tables are built on demand, so only a small set of page tables are created in reality for processes. Similarly to page directories, page tables are built up from *page table entries (PTEs)* containing the location (i.e., page frame number) of the referenced page and certain flags which indicate the state and protection of the corresponding page [16]. We refer to the state of PTEs and PDEs as *paging entry types*.

Software Supported Memory Management. When the hardware MMU intends to access an invalid page, indicated by a status flag in the corresponding PTE, a page fault occurs which has to be handled by the OS. More precisely, when a hardware PTE is invalid it is referred as a *software* PTE, because the software MMU takes care of it by checking other PTE status flags to determine how to evaluate this software PTE. This evaluation process will be discussed later in Sect. 4.2. As Windows defers to build hardware PTEs until, for example, an allocated memory is first accessed, it has to record the allocation information internally. To achieve this, the software MMU maintains a self-balanced tree called *Virtual Address Descriptors (VAD)* tree [4] for each user-space process and stores the root of it (i.e., *VadRoot*) to help its traversal. Thus, whenever a memory is allocated, for example, by the `NtVirtualAllocate` native function, Windows adds a new VAD entry to the VAD tree of the allocator process. Each

VAD entry describes a virtually contiguous non-free virtual address range including protection information and the type of this range (e.g., copy-on-write, readable, writable). Windows makes a difference between (i) dynamically allocated memory and (ii) memory which contains data from files. The former is called *private memory* and reflects allocations on the heap or stack, the the latter one is called *file-backed or mapped memory* as it is associated with files, such as data files, executables, or page files, which can be mapped into the address space [20].

3.3 Root Cause Analysis

In order to understand why memory paging analysis can be a prominent method to detect malicious code loading, we explored the details of Windows internals by using various resources [16,20,14] as well as the static and dynamic analysis of the Windows 7 kernel. For dynamic analysis, we used WinDbg. In this way, we could understand how certain WinAPI, and thus, kernel functions influence per process paging entry types. Please see Table 1 for more details on the paging entry types.

A typical code loading attack first calls `VirtualAllocEx` to allocate a *private* memory in the target process to store the name of the DLL we intend to load into that process. It creates and initializes a single VAD entry with *RW* protection bits set, *zeroes out the PTE* (i.e., zero type software PTE is created) of the allocated memory range and sets up the page fault handler (#PF). The `NtProtectVirtualMemory` kernel function is also called after this, which sets software PTE protection bits, thus the software PTE is changed to be Demand Zero (Dz). When `WriteProcessMemory` tries to access the page, the page fault handler is invoked. At this point, the page fault is caught, hardware PTE is created and initialized (i.e., Valid and Write hardware PTE bits are set) using the previously created VAD entry as a template. This lazy memory allocation mechanism allows Windows to build page tables for only accessed pages. When #PF finished, the execution is handed to the `NtWriteVirtualMemory` kernel function, which writes the name of the malicious DLL to be loaded by the target process. As the allocated pages were accessed and modified, the hardware MMU now sets the accessed (A) and dirty (D) bits on the PTE, respectively. In the next step, `CreateRemoteThread` is called by setting the newly created thread's start address to the `LoadLibrary` WinAPI function.

At this point, `LoadLibrary` is invoked by the newly created thread with the name of the malicious DLL to be loaded. As `LoadLibrary` first invokes `NtOpenFile` to return a file handle, that is handed to `NtCreateSection` to create a *Section* object for the the corresponding DLL. Due to the lazy memory management of Windows no hardware PTE is created at this point, however, only Prototype PTE kernel structures are initialized. This structure is used by the OS to handle shared memory between different processes. We emphasize here, that PPTEs are initialized with `EXECUTE_WRITECOPY` protection masks to enable Copy-On-Write operations. To support memory sharing the DLL is now mapped as view of the section object (i.e., file-backed-image is mapped) by the

`NtMapViewOfSection` kernel function, which also creates a VAD entry by using the Prototype PTEs and sets up the page fault handler to build hardware PTEs when the mapped DLL is first accessed. Finally the opened file handler was closed by `NtClose`.

If certain parts of the loaded DLL is not used currently, the OS swaps the corresponding pages and creates **Prototype PTE* entries which point to the original *Prototype PTEs*. In this way, the memory manager can quickly load back the page when it is referenced again, and uses the Prototype PTE as a template to set the protection masks to build a new hardware PTE. The presented code loading technique creates *Zero software PTEs* and *hardware PTEs with Valid, Writable, Accessed, Dirty and Copy-on-Write bits set*. This example shows fairly well, how some simple WinAPI functions influence the management of memory in Windows OSs. This rich behavior allows us to observe how different code loaders manipulate per process PTEs.

It is important to emphasize that not only the fact of code loading diverts memory paging states, but also do those *malicious functionalities* (e.g., data exfiltration, keylogging) that the loaded code executes.

3.4 Overview of Membrane

Membrane is a Volatility extension and works on virtual machine memory snapshots. Membrane detects all the active processes when the analyzed snapshot was taken and starts to dissect these processes to determine their memory paging states. By doing so, Membrane first restores a wide range of memory paging states as Table 1 shows it. To the best of our knowledge, we are the first, who restore memory paging states in that detail and use them for a posteriori malware detection as detailed in Sect. 4.2. Then, Membrane calculates different paging state cardinalities (e.g., the number of zeroed out pages present in the address space of a given process) and creates features from them. These feature candidates are accumulated into a feature set, and out of these candidates the prominent features are selected by a machine learning algorithm. These feature are later used for decision making as described in Sect. 4.3.

4 Implementation

First, we select key features of paging states and train a *random forest classifier* to detect code loading behavior. These candidate features are later evaluated and filtered to pick the most relevant ones suitable to detect code loading. In the testing phase, we repeat the training phase, but samples are classified (i.e., detected as malicious or not) only with the chosen features.

4.1 Collecting Samples

To train our system, we first collect samples for our benign and malicious datasets. Thus, we prepare our malware dataset with prudent practices [15]

to have a ground truth consisting of malicious samples using malicious code loading. Considering malicious samples, we work with one dataset of generic malware and one dataset of targeted malware samples. We collect the *generic* dataset from a large set of generic malware samples retrieved from public and private malware data feeds. Our initial *generic* dataset comprised of 9905 samples, however, our thorough dataset preparation process (see Sect. 2.2 in [12] for details) resulted in a balanced dataset of 1095 binaries adequate to start and analyze their runtime behavior. Out of these, we have found 194 active malware samples that exhibited code loading. Similarly, we prepare goodware programs to have a balanced dataset for machine learning, we perform later. Then, we install and start multiple benign applications and a single malicious sample into a VM under analysis. Next, snapshots are retrieved from this running VM, which are handed to our tool, Membrane, for analysis. For testing *targeted* malware, we manually selected 7 malware samples from active targeted campaigns excluding malware whose C&C servers have been sinkholed or shut down.

4.2 Feature Creation

The observations in Fig. 1 strongly suggest that code loading attacks have a significant effect on the frequency of different paging states. Thus, we use the frequency of various paging state to set up detection *features*. We summarize all the paging states we handled during our feature creation process in Table 1.

Membrane internals. We carefully investigated Windows-specific page types and their corresponding translation process to restore our features from paging states. Albeit, most of these page types are documented with a certain level of granularity [8,20,16], some of them are still not well understood. Our implementation currently supports both x86 PAE enabled and x64 images. While we tested the former case on 32-bit Windows XP and 7 OSs, the latter was verified on 64-bit Windows 7 and 8.1 snapshots. As a result of our translation process we are capable of restoring 28 different paging states as shown in Table 1.

Our translation process works as follows. We first natively open the memory snapshot file (i.e., .vmsn) with Volatility. We then locate *VadRoot* values for running processes by iterating through the *EPROCESS* kernel structure. VadRoot stores the virtual address of the process VAD tree [4]. At this point, we traverse the VAD tree by locating VAD entries which describe all the valid virtual addresses belonging to the process virtual address space. From this point on, we need to resolve all these virtual addresses by slicing them into page-sized buffers.

As a next step, we modified Volatility's internal address translation process as follows. Whenever a new process virtual address is retrieved by Volatility, it starts to resolve the address by traversing the process page tables. Similarly to the hardware MMU, it returns the corresponding physical address if all the page table entries (e.g., PDE, PTE) were valid. If it encountered an invalid entry (e.g., Software PTE), it raises an exception or returns with zero values. Thus, we had to implement the software MMU here to retrieve exact paging entry types.

Table 1: Paging entries we restored for feature creation.

Name	Description
RESTORED FEATURES FOR NON-PAE 32-BIT WINDOWS XP	
Valid	The PTE points to a page residing in the physical memory
Zero PDE	The Page Directory Entry contains all zeroes
Zero PTE	The Page Table Entry contains all zeroes
Pagefile	The PTE points to a page which is swapped out
Large	The PDE maps a 4-MB large page (or 2MB on PAE systems)
Demand Zero	The memory manager needs to provide a zeroed out page.
Valid Prototype	The *PPTE points to (directly or via VAD) a PPTE which points to a valid page
Invalid Prototype	The PPTE points to an unresolvable page.
Pagefile Prototype	The *PPTE points to a PPTE which points into a pagefile.
Demand Zero Prototype	The *PPTE points to a PPTE which translates to a zeroed out page
Mapped File	The PTE is Zero, thus we resolve it from VAD which points to the mapped file
Mapped File Prototype	The *PPTE points to a PPTE where the prototype bit is set and has *Subsection* type.
Unknown Prototype	The same as Mapped File Prototype, but PPTE has no *Subsection* type.
Writable	The PTE points to a writable page
Owner (Kernel mode)	The corresponding page can be accessed in user- or kernel mode
Cache Disabled	The PTE points to a page where CPU caching is disabled
Writethrough	The PTE points to a write-through or write combined page
Accessed	The PTE points to an accessed page
Dirty	The PTE points to a written page
Transition	A shared page is no longer referenced and moved to the transition list
Copy-on-write	The PTE points to a copy-on-write page
Global PDE	The address translation applies to all processes
Unknown	The PTE is not empty, but does not belong to any of the cases.
ADDITIONAL FEATURES FOR PAE-ENABLED 32-BIT WINDOWS 7	
Non-executable	The PTE points to a non-executable page
VAD Prototype	The *PPTE points to a location described by the VAD tree
Modified-no-write	The requested page is in the physical memory and on the modified-no-write list
Zero PDPTE	The Page-directory Pointer Table Entry contains all zeroes
ADDITIONAL FEATURES FOR 64-BIT WINDOWS 7	
Zero PML4E	The Pagetable Level 4 Entry contains all zeroes

To do that, we resolve invalid entries using the VAD tree as Fig. 2 shows it. Furthermore, Table 1 details how we resolved different paging entry types and restored paging entry states from them.

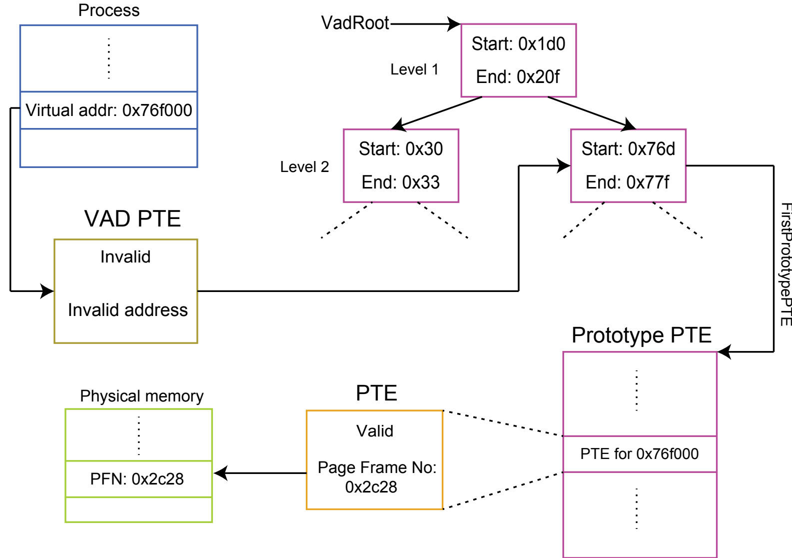


Fig. 2: Crawling VAD tree by Membrane to resolve invalid paging entries and restore the corresponding paging states.

4.3 Feature Selection and Classification

After evaluating possible paging states, we defined 28 different features under Windows 7 and 23 features under Windows XP as shown in Table 1. We selected these features to train our system. Thus, we had an X matrix with m rows and 28/23 columns, where m is associated with the number of different code loader malware samples. As our training set is relatively small, we used various methods for better evaluation. First, we used feature scaling to represent our features on the same scale (i.e., we used the interval of $[0-1]$). Then, we applied feature reduction to avoid overfitting by calculating the Gini index of our normalized features. The Gini index shows the inequality among the values of a frequency distribution. In our case, per process frequency distributions are defined by the cardinality of given per process paging states in n consecutive memory snapshots. The lower this index is, the more equal the values are in a frequency distribution.

For later classification, we selected features that had smaller index than the mean of all indexes. This resulted in using features *Zero PTE, Demand Zero, Valid Prototype, Mapped File Prototype* and *Copy-on-write* under Windows XP. The selected features under Windows 7 are *Zero PTE, Accessed, Demand Zero,*

Writable, Owner, Writethrough, Cache disabled, Pagefile, Valid Prototype, Valid, Dirty and *Mapped file*. The process noise under Windows 7 is higher than in Windows XP, thus we needed more features for classification.

4.4 Detecting Code Loading

After selecting the prominent features we used *random forest* classification to detect code loading. Random forest is a bag of randomly split decision trees. We can collect the trees with this method and the majority voting of the trees will classify a given sample.

Initially, we followed the hint obtained from the execution of the Flame malware shown in Fig. 1. We first took single snapshots of malware infected VMs to see if we detect some of the features we identified in Sect. 4.3.

We then created multiple snapshots periodically from the same infected VM to understand in detail how code loading attacks modify the number of paging states and how the OS manipulates them over time. There are various reasons to perform this measurement. First, single snapshots could miss the activation of execution-stalling malware which defer their parasitic behavior until a certain event occurs (e.g., no keyboard press for 5 minutes for Duqu). Second, malware could retrieve further components and load them into other system processes as well. This behavior can only be detected if we repeatedly take snapshots. Third, Windows OSs use various memory optimization methods, for example, to conserve physical memory. These optimization routines could also affect our features.

To measure the *accuracy (ACC)* of our classification process, we calculate the ratio of all the correctly classified machines per all the machines we tested. More precisely, $ACC = \frac{TP+TN}{P+N}$, where TP, TN refer to the number of successfully identified infected and clean VMs, and P, N indicates the number of infected and benign machines, respectively.

Furthermore, we calculate *true positive rates (TPR)* to denote the ratio of correctly classified infected machines compared to all the infected machines we tested (i.e., $TPR = \frac{TP}{P}$). Finally, the false positive rate ($FPR = \frac{FP}{N}$) is used to denote misclassification of benign machines, where FP means that an infected machine bypasses detection and is marked as clean.

5 Analysis

We now present our result detecting code loading behavior in contemporary malware. We then apply our code loading detection method for a dataset composed of generic malware samples as well as hand-picked samples from recent targeted attacks.

5.1 Analysis Setup

We built a dynamic execution environment for Membrane to systematically detect code loading by contemporary malware. We instrument virtual machines

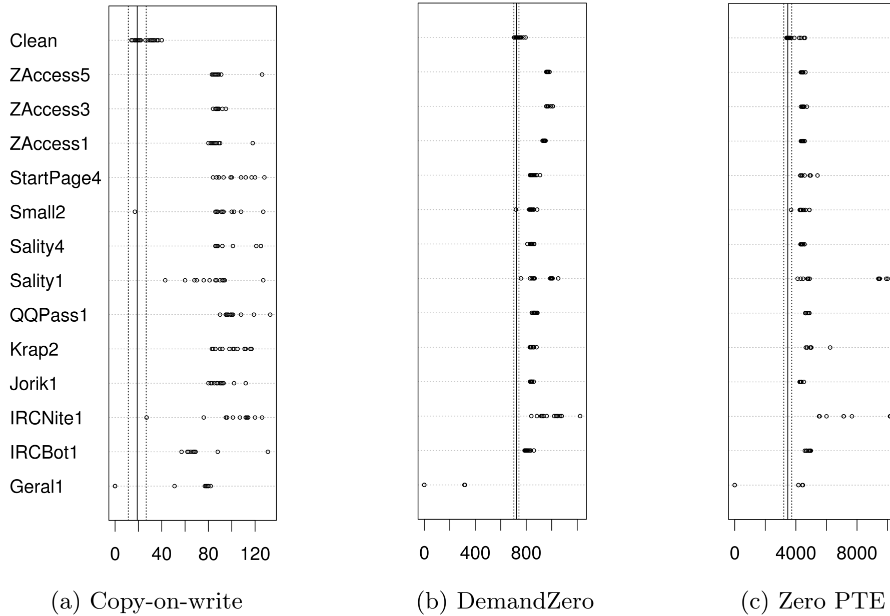
(VMs) using VMware's ESXi to run malware samples with various operating system and network settings. Designing a containment policy for prudent malware experiments is essential and we used the guidelines of [9] and [15] to design our containment policy.

We implemented our execution environment for Membrane on *two* identical physical machines with Intel Core i7-4770 CPUs, 32 GB of memory and VMware ESXi 5.5 OSs. Host machine (*ESX1*) runs our VMs and takes snapshots, and the other one (*ESX2*) is responsible to retrieve information from these snapshots and perform analysis by Membrane which enables detection. We used a controller VM (*CTRL*) to invoke VMware vSphere API via the *psphere* python module. This module is a native binding to control VMware ESXi environments. We executed our tests on multiple VMs with different OS configurations (*WinXP, Win7_1, Win7_2*) to test the robustness of our approach against system changes. While WinXP and Win7_1 are 32-bit Windows XP and 7 installations, Win7_2 is a 64-bit Windows 7 OS. Our exact system configuration is detailed in Table 2.

Table 2: Machine configurations we used in our malware detection and analysis system. Virtual machines WinXP, Win7_1 and Win7_2 are running on the ESX1 host machine and are used for snapshot based malware detection. Note that Win7_2 was configured with 1 and 4 GBs of memory for more complete evaluation. Note that the default setup comes with 1 GB of memory, so the other case is explicitely mentioned when used.

Type	**Name**	**Memory size**	**OS**
Host	ESX1	32 GB	ESXi 5.5 U1
	ESX2	32 GB	ESXi 5.5 U1
VM	CTRL	4 GB	64-bit Ubuntu 14.04
	WinXP	512 MB	32-bit Win XP SP3
	Win7_1	1 GB	32-bit Windows 7
	Win7_2	1/4 GB	64-bit Windows 7

We instrumented our experiments with two different network configurations to study the activation behavior of various malware samples. The two configurations are the following: (i) no Internet connection is enabled, (ii) real Internet connection is enabled with a carefully crafted containment policy following the suggestions of Rossow *et al.* [15]. When Internet connection was enabled, we used NAT with the following containment policy: a) we enabled known C&C TCP ports (e.g., HTTP, HTTPS, DNS, IRC) b) we redirected TCP ports with supposedly harmful traffic (e.g., SMTP, ports not registered by IANA) to our INetSim network simulator [6] we also configured, and c) we set rate-limitation on analyzed VMs to mitigate DoS attacks.



(a) Copy-on-write (b) DemandZero (c) Zero PTE

Fig. 3: Recorded number of paging states in `explorer.exe` running in our WinXP environment. Each dot on horizontal lines shows the number of paging state in a given snapshot. The vertical straight line corresponds to the median of clean executions, while the dashed lines depict the standard deviation.

5.2 Detecting Malware

First, we created a cross-validation dataset from generic malware samples to test our tool with the features selected in Sect. 4.3. To make sure that the feature selection is not biased towards our training dataset, we also compiled a test-only dataset from a mix of generic malware and confirmed targeted attack samples.

Birds-eye-view Analysis. We first created sample runs on our test machine WinXP VM with only legitimate applications installed (see Sect. 2.2 in [12]) as a benchmark. We also infected these systems with samples from selected malware families performing code loading.

We repeated our evaluation process 13 times in case of malware samples and 54 times for goodware. In this way, we estimated certain statistics (i.e., median, standard deviation) to see whether these features can be used to detect the of malware. Figure 3 shows how the cardinality of certain paging state changes in `explorer.exe` on Win XP for different generic malware executions.

Detailed Analysis. Then, we executed an extensive set of experiments with the two network containment configurations described in Sect. 5.1. Out of the 194 generic code injector malware, 128 samples targeted `explorer.exe` which turned

to be the most popular injection target. That is why our classification algorithm works with these samples. Our evaluation process comprises three parts: (i) code loaders are evaluated on VMs with preinstalled and started benign applications with K-fold cross-validation in accordance with our methodology detailed in Sect. 4, (ii) same as the previous point, but no benign processes are installed and started, (iii) the best performing random forest classificator is chosen from the cross-validation process to evaluate our test-only dataset.

In cases (i) and (ii), we infected our prepared Windows XP (i.e., WinXP) and Windows 7 (i.e., Win7_1) environments in each network containment configuration. After retrieving snapshots with Membrane, we run our classification algorithm to find malicious infections. Note that we put equal number of benign and infected snapshots into each cross-validation process. Table 3 summarizes our results and strengthens some of our key observations. According to our observations running processes raise the noise of certain system processes. This observation manifests in lower detection ratio as the (i) and (ii) versions of Win7_1 show it. Our next observation is that the execution environment also affects detection accuracy as the *TPR* of Win7_1 and WinXP setups show it.

While the detection rates are fairly good in case of Windows XP, the results are worse for Windows 7 snapshots due to the increased baseline memory paging of `explorer.exe` in Win7_1. Interested readers can read more about our measured baseline noises in Table 2.15 in [12]. We can increase the detection accuracy by stopping legitimate processes and thus offloading `explorer.exe`. We see this increase when the TPR in Table 3 increases from 75.92% to 86%.

Considering case (iii), we could detect all the *generic* samples in our test-only dataset in both environments by using the best classificator chosen by the cross-validation process. Note that we also detected all the *targeted* samples of our test-only dataset (i.e., Flame, Shale, Snake, Epic and Turlae where the latter four belongs to the Uroburos campaign revealed by G Data in 2014) on Win7_1, all injecting into `explorer.exe`.

Interestingly, however, the state of network connection slightly increased the detection accuracy on Win7_1 setups, and decreased it on WinXP. One of the explanations could be that injected payloads on Windows 7 downloaded extra components, which added extra process noise that was easier to detect.

While `explorer.exe` is the most popular target of code loading according to our dataset, there are many other system processes with lower noise (e.g., `services.exe` or `winlogon.exe`) that are known to be also preferred by miscreants.

A combined detection over several processes could further increase accuracy for malware that injects into multiple processes (eg. Flame). We did not do this extensive study as it was not our focus.

6 Discussion

Systems Under Heavy Load. Our results showed that our experiments on the *WinXP* operating system were fairly accurate with a very low false positive

Table 3: Detecting generic and targeted code loading malware on our WinXP and Win7_1 VMs with different network connections in `explorer.exe`. We used a $K = 5$ cross-validation iteration for the measurements. Note that the targeted samples of our test-only dataset were executed only under Win7_1.

Internet	VM	ACC %	TPR %	FPR %
CROSS-VALIDATION DATASET				
no	WinXP	98.67	98.82	1.18
	Win7_1	73.59	75.92	28.46
yes	WinXP	92.81	90.88	5.45
	Win7_1	79.45	82.69	23.59
CROSS-VALIDATION DATASET WITH NO ADDITIONAL PROCESSES				
no	Win7_1	77.16	86.00	34.73
TEST-ONLY DATASET				
no	WinXP	100	100	-
	Win7_1	100	100	-

rate. We noticed, however, that occasionally on *Win7* the baseline operation of the system was dependent on the realistic system load and the target process. This meant that heavy system use (e.g. watching Youtube videos, installing and starting of various applications or stress-testing the memory) caused a significant increase in the number of paging states (for example, the `explorer.exe` process produced a high number of Copy-on-Write and Demand Zero paging states on a *Win7_1*). This somewhat reduced the efficiency of detection in these cases. As `explorer.exe` is taking care of the graphic shell, every newly spawned GUI application can raise process noise. That is a key observation especially for Windows 7 systems, where our detection ratio was lower.

Clearly this increase in the number of paging states only appeared when the system was under a heavy load. One can circumvent this anomaly by establishing baseline cardinality of paging states under normal system load (e.g., in a nightly operation). Assuming that the system is not under a permanent load, we can wait until it returns to normal load and perform the detection then.

Countering Evasion. Legitimate software sometimes uses code loading to achieve more functionalities [17]. For example, the Google toolbar uses DLL loading for legitimate purposes. Debuggers, such as the Microsoft Visual Studio Debugger also exhibit this behavior. Our tool detects the fact of code loading, but it cannot judge if it was for legitimate purposes. To filter out the false positives caused by legitimate applications, we can mark them and compile a whitelist. This whitelist can be constructed for example by using cryptographic hashes to user space memory allowing the identification of known code [19]. These legitimately injecting applications could then be excluded from analysis.

Unfortunately, this solution would open the door for malware to evade detection by loading code into the whitelisted applications. We can counter this option by applying our memory paging analysis on the application process instead of the system processes. We can first establish a baseline for a clean operating system running the code injecting legitimate application. Then, we proceed as follows. First, we run Membrane comparing the baseline behavior without code loading with the system behavior. If Membrane indicates an alarm, we check if any of the legitimate code loading mechanisms are running on the system. If such an application exists, then we run Membrane against the baseline behavior established earlier for this application. If Membrane still raises an alarm that is a good indicator that a malware infected the system, otherwise the previous alarm was a false alarm caused by the legitimate code injecting application.

Performance Issues. As snapshot creation can be a heavy-duty operation, we further designed and implemented a live monitoring version of Membrane called Membrane Live by extending a virtual machine introspection-based malware analysis tool called DRAKVUF [10]. This tool is used solely for evaluation purposes, and to compare the performance of snapshot and live monitoring-based approaches. More information about Membrane Live can be found in Sect 2.2 in [12].

7 Conclusion

In this paper, we present a memory forensics tool called Membrane to detect code loading behavior of malware programs. Instead of detecting various code loading actions, our tool focuses on identifying the cardinality of paging states caused by code loading. With this more generic technique, we are able to detect a wide range of malware code loading techniques. Our results show that we are able to identify 86-98% of code loading malware samples on Windows machines, depending on the system load and the targeted process. In summary, we show that Membrane significantly raises the bar for miscreants to employ code loading attacks.

While for certain conditions (e.g., explorer.exe on Windows 7 under heavy load) our approach may seem less effective, we do emphasize that this condition is one of the worst cases according to our measurements. In future work, we will study how detection can be further improved. We will also investigate if Membrane can be extended to detect kernel-only code loading attacks. All in all, we believe that our approach tackles the problem of code loading from another, interesting aspect which can even be combined with existing methods effectively.

References

1. AlienVault. Batchwiper: Just another wiping malware. https://www.alienvault.com/open-threat-exchange/blog/batchwiper-just-another-wiping-malware, accessed on Nov 13, 2014.

2. Bencsáth, B., Pék, G., Buttyán, L., and Felegyhazi, M. The cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet 4*, 4 (2012), 971–1003.
3. CERT.PL. More human than human - Flame's code injection techniques. http://www.cert.pl/news/5874/langswitch_lang/en, accessed on Nov 13, 2014.
4. Hand, S., Lin, Z., Gu, G., and Thuraisingham, B. The vad tree: A process-eye view of physical memory. *Digital Investigation 4* (2007), 62–64.
5. Idika, N., and Mathur, A. P. A survey of malware detection techniques. Tech. rep., Purdue University, 2007.
6. INetSim. http://www.inetsim.org/, accessed on Nov 10, 2014.
7. Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 128–138.
8. Kornblum, J. D. Using every part of the buffalo in windows memory analysis. *Digital Investigation 4*, 1 (2007), 24–29.
9. Kreibich, C., Weaver, N., Kanich, C., Cui, W., and Paxson, V. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the 2011 ACM SIGCOMM Internet Measurement Conference (IMC)* (2011), ACM, pp. 397–412.
10. Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., and Kiayias, A. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference* (Dec. 2014). To Appear.
11. Mandiant. APT1: Exposing One of China's Cyber Espionage Units. http://intelreport.mandiant.com/, 2013.
12. Pék, G. *New Methods for Detecting Malware Infections and New Attacks against Hardware Virtualization.* PhD thesis, Budapest University of Technology and Economics, 2015.
13. Petroni, Jr., N. L., Fraser, T., Walters, A., and Arbaugh, W. A. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.
14. ReactOS. A free open source operating system based on the best design principles found in the Windows NT architecture. http://doxygen.reactos.org, accessed on Nov 8, 2014.
15. Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., and Van Steen, M. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 65–79.
16. Russinovich, M., Solomon, D. A., and Ionescu, A. *Windows Internals, Sixth Edition*, 6th ed. Microsoft Press, 2012.
17. Srivastava, A., and Giffin, J. Automatic discovery of parasitic malware. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection* (2010), pp. 97–117.
18. Volatility. The Volatility Framework. https://code.google.com/p/volatility/, accessed on Nov 13, 2014.
19. White, A., Schatz, B., and Foo, E. Integrity verification of user space code. *Digital Investigation 10* (2013), 59–S68.
20. Willems, C. Internals of windows memory management (not only) for malware analysis. Tech. rep., Ruhr Universität Bochum, 2011.