Integrity Checking of Function Pointers in Kernel Pools via Virtual Machine Introspection

Irfan Ahmed, Golden G. Richard III, Aleksandar Zoranic, Vassil Roussev

Department of Computer Science, University of New Orleans
Lakefront Campus, New Orleans, LA 70148, United States
irfan.ahmed@uno.edu, golden@cs.uno.edu, azoranic@uno.edu, vassil@cs.uno.edu

Abstract. With the introduction of kernel integrity checking mechanisms in modern operating systems, such as PatchGuard on Windows OS, malware developers can no longer easily install stealthy hooks in kernel code and well-known data structures. Instead, they must target other areas of the kernel, such as the heap, which stores a large number of function pointers that are potentially prone to malicious exploits. These areas of kernel memory are currently not monitored by kernel integrity checkers.

We present a novel approach to monitoring the integrity of Windows kernel pools, based entirely on virtual machine introspection, called HookLocator. Unlike prior efforts to maintain kernel integrity, our implementation runs entirely outside the monitored system, which makes it inherently more difficult to detect and subvert. Our system also scales easily to protect multiple virtualized targets. Unlike other kernel integrity checking mechanisms, HookLocator does not require the source code of the operating system, complex reverse engineering efforts, or the debugging map files. Our empirical analysis of kernel heap behavior shows that integrity monitoring needs to focus only on a small fraction of it to be effective; this allows our prototype to provide effective real-time monitoring of the protected system.

Keywords: virtual machine introspection; malware; operating systems.

1 Introduction

Malware (especially rootkits) often targets the kernel space of an operating system (OS) for attacks [1], modifying kernel code and well-known data structures such as the system service descriptor table (SSDT), interrupt descriptor table (IDT), and import address table (IAT) to facilitate running malicious code. In other words, malware of this type installs hooks, which enables it to control the compromised system. For instance, such malware might hide the traces of infection, such as a user-level malicious process, or introduce remote surveillance functionality into the system, such as a keylogger.

This work was supported by the NSF grant CNS # 1016807.

Microsoft (MS) has introduced kernel patch protection (a.k.a. PatchGuard) in 64-bit Windows (such as Windows 7/8) to protect the integrity of kernel code and the data structures often targeted by traditional malware. It is implemented in the OS kernel and protected by using anonymization techniques such as misdirection, misnamed functions and general code obfuscation. In the presence of PatchGuard, it is hard for malware to directly install stealthy hooks in kernel code or modify the data structures monitored by PatchGuard. In order to avoid detection, modern malware has modified its targets to previously unexplored data regions. In particular, it targets function pointers in the dynamically allocated region in kernel space (a.k.a. kernel pools) [15],[16],[17]. Function pointers refer to the entry point of a function or routine and by modifying a function pointer, an attacker can cause malicious code to be executed instead of, or in addition to, the intended code. A demonstration of this type of attack appears in Yin et al. [2]. They created a keylogger by simply modifying function pointers corresponding to a keyboard driver in a kernel pool. Moreover, there are thousands of function pointers in the Windows kernel pools, which provides an attractive opportunity for an attacker to install stealthy hooks [2].

Current solutions such as SBCFI [3], Gibraltar [4], SFPD [5], and HookSafe [6] check the integrity of function pointers by generating hook detection policy and extracting information about function pointers by performing static analysis of the kernel source code. For example, they obtain the definitions of the kernel data structures that contain the pointers, and subsequently generate the traversal graph used to reach the data structures containing the function pointers. Unfortunately, these solutions are dependent on the availability of kernel source code and thus not appropriate for closed source OSs such as MS Windows.

More recently, Yin et al. presented HookScout [2] for checking the integrity of function pointers in MS Windows. It uses taint analysis [7] to trace the function pointers in kernel memory, and generates a hook detection policy from the memory objects that hold the pointers. The tool learns about the function pointers from a clean installation of the OS with typical user applications installed on it. The effectiveness of HookScout depends on how much contextual information is obtained about the function pointers during the learning phase. During the detection phase, if a target machine is attacked via modification of a function pointer not evaluated by HookScout during its learning phase, it will be unable to check the function pointer integrity.

Importantly, HookScout was developed on a 32-bit Windows XP OS and its current approach cannot be readily extended to 64-bit Windows 7:

- For detecting hooks, HookScout adds a `jmp` instruction at the entry point of each function whose function pointer is being monitored. The `jmp` redirects the execution to its own detection code. In 64-bit Windows 7, such patching of functions is not possible due to PatchGuard.
- HookScout obtains a list of function pointers for analysis by disassembling the MS Windows kernel in IDA Pro and traversing through the relocation table to identify the absolute addresses of function pointers. This works on 32-bit Windows XP since Windows uses absolute addresses in the kernel to

refer to functions and thus an entry for such addresses exists in the relocation table. However, 64-bit Windows 7 uses offsets from the current instruction to refer to functions, because the code in 64-bit Windows is guaranteed to be aligned on a 16-byte boundary [8]. Thus, the relocation table does not contain entries for all the function pointers in kernel code.
- Finally, HookScout is implemented as a kernel module, which runs in the kernel space being monitored in the target machine. Thus, it is prone to subversion like any other security solution running inside a compromised machine, a limitation acknowledged by the authors of HookScout.

In this paper, we present `HookLocator` for MS Windows, which checks the integrity of function pointers in a virtualized environment and identifies function pointers that are maliciously modified in a kernel pool. HookLocator runs in a privileged virtual machine and uses virtual machine introspection (VMI) to access the physical memory of a target virtual machine (VM) running MS Windows. Since HookLocator runs outside the target VM, it is less prone to subversion and can obtain the list of function pointers directly from reliable sources in the physical memory of the target machine (such as kernel code and data structures monitored by PatchGuard), without disassembling kernel code or traversing the relocation table. The list is then used to find the instances of function pointers in kernel pool data. HookLocator does not require hooking to obtain the kernel pool data; instead, it uses kernel data structures maintained by Windows to track memory allocations to locate appropriate dynamic allocations in kernel pools. Our tool does not require access to source code to learn contextual information about function pointers; instead, it obtains all the information directly from the physical memory of the target machine. Thus, it continues learning from the target machine even during the monitoring phase, in order to obtain new information about the pointers under scrutiny.

The main contributions of this work are as follows:

- We propose a new approach to obtain the list of function pointers to be monitored directly from physical memory. The approach takes two memory snapshots of kernel code that are loaded into two different locations in memory and uses the differences to locate candidate function pointers. The locations are marked and then used to obtain the function pointers from the in-memory kernel code of the target machine.
- We propose a VMI-based hook detection approach to check the integrity of function pointers in kernel pools. The approach obtains the list of function pointers and their context information directly from the physical memory of the target system.
- We present a proof-of-concept prototype, HookLocator, for 64-bit Windows 7 to evaluate the effectiveness and efficiency of the approach.
- We thoroughly evaluate HookLocator on Windows 7 and identify a region in kernel pool, which provides a non-pageable target-rich attack surface. HookLocator is able to perform real-time monitoring on the region with a negligible amount of memory overhead.

2 Related Work

In addition to HookScout, which represents the current state of the art, a number of other techniques have been developed to combat kernel hijacking techniques. Since the inclusion of PatchGuard within Microsoft Windows, rootkits are no longer able to hook to kernel objects by modifying kernel code. This, in turn, has changed malware hooking techniques by redirecting their subversion efforts to the previously unexploited kernel heap. Therefore, much prior work has become obsolete, and not up to the task of reliably detecting rootkits. Many of the techniques, which we outline below, have substantial practical limitations as they rely on source code availability and/or incur unacceptable performance penalties.

Riley *et al.* implemented PoKeR[18], a rootkit behavior profiling engine. It relies on the OS kernel source code for static analysis or debugging symbols and type information for binary analysis. Furthermore, certain modules in PoKeR, such as context tracking and logging, run inside the VM allowing malware to tamper with such modules. PoKeR utilizes a resource intensive profiling engine, which incurs a significant overhead and severely limits its practical deployment.

Yin *et al.* developed HookFinder [19], a method that uses dynamic analysis of kernel code to identify and analyze malware hooks. Designed mostly for analytical purposes, HookFinder does not constantly check for rootkit activity, but rather provides a controlled experimental environment to analyze rootkits in. The tool is relatively resource intensive and, therefore, potentially detectible by the malware using performance monitoring.

Carborne *et al.* [5] implements a system called KOP to perform systematic kernel integrity checking by mapping kernel objects. KOP operates in an offline manner by capturing Windows kernel memory dumps and using the Windows debugger to obtain information about the kernel objects. KOPs static analysis utilizes Vistas source code to identify relevant data types, variables and structures.

Wang *et al.* propose HookSafe [6]a solution that creates a shadow copy of non-movable function pointers, and return addresses mapped to a hardware protected page aligned area. Requests to modify this shadow copy are intercepted and forwarded to the hypervisor module that inspects the validity of these requests. It requires knowledge of OS source code, which is not always available, as in the case of Windows.

Nick *et al.*'s State Base Control Flow Integrity (SBCFI) [3] system conducts periodic kernel control flow integrity checks. To do so, it relies on knowledge of kernel source code and binary files. As stated before, OS kernel source code for commodity OSs is usually not readily available.

Baliga *et al.*'s Gibraltar [4] relies on a separate observer machine to capture periodic memory snapshots of the target machine for integrity analysis. This idea is similar in spirit to our approach; however, Gibraltar's implementation relies on constantly capturing memory snapshots via Direct Memory Access (DMA). This poses a significant performance overhead on the observer machine and increases training time to almost an hour; it also requires additional hardware just for

monitoring. Another shortcoming is that Gibraltar requires kernel source code for the initial static analysis.

In summary, each of the prior efforts surveyed exhibit at least one of the following limitations: a) reliance on source code; b) execution at the same level of privilege as the malware; c) significant performance overhead; and d) incomplete coverage of kernel space. The HookLocator overcomes most of these limitations and present a viable solution to kernel heap integrity monitoring.

3 Integrity Checking for Function Pointers

3.1 Environment

In a modern virtualized environment, a virtual machine manager (VMM) allows multiple virtual machines (VMs) to run concurrently on the same physical hardware. The virtual machines are called guest VMs, and are isolated from each other by the VMM. However, some VMMs also provide virtual machine introspection (VMI) capabilities that allow a privileged VM to monitor the system resources (such as physical memory) of guest VMs. HookLocator works in such a virtualized environment, where it runs in a privileged VM and accesses the physical memory of guest VMs through VMI. We also assume that the kernel code and the well-known kernel data structures inside the guest VMs are protected by PatchGuard or an alternate solution, such as VICE [9], System Virginity Verifier [10], ModChecker [11] and IceSword [12].

The remainder of this section describes HookLocator's architecture in more detail.

3.2 HookLocator Architecture

Figure 1 shows the architecture of HookLocator, which consists of four modules: extraction, search, learning, and pool monitor. The extraction module builds a list of function pointers from reliable sources in the physical memory of a guest VM, which is used by the search module to locate candidate function pointers in the kernel pool data. The learning module uses heuristics to identify the genuine function pointers, which are then monitored for integrity by the pool monitor. The details for each module are provided in the following sections.

Extraction Module The extraction module obtains function pointers from kernel code and well-known data structures residing in the physical memory of a guest VM and builds a list that is subsequently passed to other modules in HookLocator. The data structures are well organized and have specific fields that either contain or lead to function pointers, which are used by the module to extract the pointers. On the other hand, kernel code does not have such fields. Thus, extracting function pointers directly from the code requires a different approach. We employ two different methods to obtain function pointers from the code, which are described next.

The first method uses a cross-comparison approach and takes advantage of the relocatable property of Windows kernel code. Note that we cannot directly use the relocation table associated with kernel components, since the image

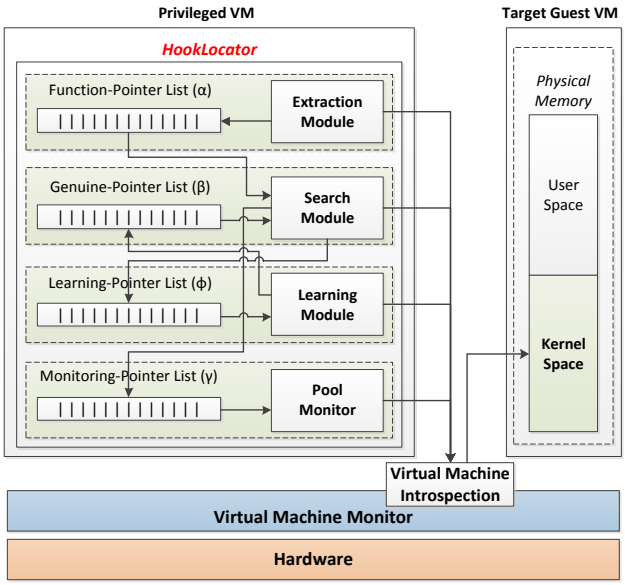


Fig. 1. HookLocator Architecture. (Arrows directed towards the lists represent write operations; otherwise, they represent read operations.)

loader often removes this table when the kernel is loaded [1]. Instead, we build a model from the kernel code, which identifies the absolute addresses in the code.

We accomplish this by comparing two snapshots of the kernel code loaded at two different locations in memory. The differences in the memory contents of the two loaded kernels are the identified absolute addresses, because the kernel has been loaded at different locations (the code itself is invariant). Figure 2 illustrates the process. If the addresses lie within the address range of kernel code, they are potentially kernel function pointers, and are tagged and stored within the protected VM. In order to extract function pointers from the kernel code of a target VM, the model is compared with the target VM's kernel code. The tags in the model identify the locations of the function pointers in the code, which are then obtained by the extraction module.

The second approach is a simple pattern matching technique, which complements the first one in that it does not count on the relocation property of the kernel. It is specifically designed for 64-bit Windows to overcome the limitation of the first approach, due to the use of offsets rather than absolute addresses [8]. We analyzed a collection of kernel functions in the 64-bit Windows 7 kernel and found a useful pattern: after return instructions (opcode `0xC3`), there are a number of `NOP` instructions (opcode `0x90`) followed by the entry point of the next

[1] While the `.reloc` section of the MS Windows kernel does contain the relocation table, the section is discardable as identified by the characteristic field in the section header.

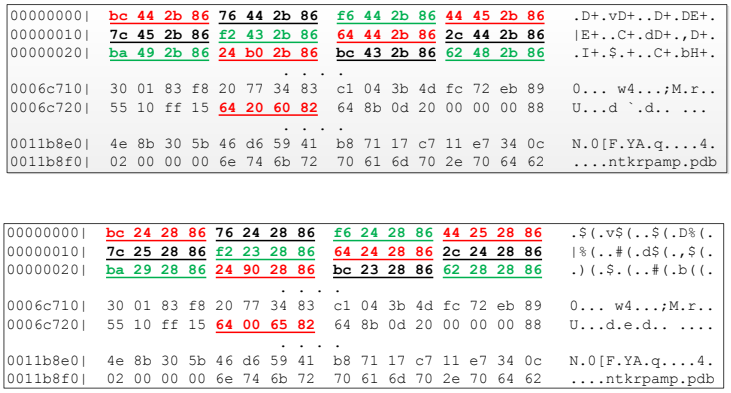


Fig. 2. Two snapshots of the .text section of the kernel code (ntoskrnl.exe) from 32-bit Windows 7. The kernel is loaded at the base addresses `00 10 60 82` and `00 f0 64 82`.

function. The extraction module uses the `0xC390` pattern to obtain a substantial list of function pointers.

Search Module The search module performs three primary tasks. First, it obtains the kernel pool data from the kernel space in the physical memory of a guest VM; this involves extracting data structures that reference kernel pools in the memory, and identifying dynamic allocations in the pools.

Second, the search module searches for function pointers in the kernel pool data. To do this, it uses the function pointer list, built by the extraction module, and locates function pointers in the pool data.

Third, the search module decides whether to pass the pointer to the learning module or the pool monitor. This decision is based on the entries in the genuine pointer list, which contains contextual information about the pointers. This includes, among others, the name of the module that made the allocation containing the pointer, the size of the allocation, and the offset of the pointer from the base address of the allocation. When the search module finds a function pointer in a pool allocation, it obtains its information and searches for it in the genuine-pointer list. If it finds it in the list, it adds the pointer to the monitoring-pointer list so that the pool monitor will check the integrity of the pointer. If the search module does not find the pointer in the genuine pointer list, it adds it to the learning-pointer list so that the learning module can appropriately identify genuine pointers and add them to the genuine-pointer list.

Learning Module A pointer located by the search module may or may not be genuine, because it can be confused with random data in the kernel pools that matches a pointer in the function pointer list. When the learning module receives a pointer via the learning-pointer list, it observes the pointer over its entire life span. Yin et al. [2] reported that 97% of function pointers in Microsoft Windows do not change over their entire life span, which provides a basis to identify genuine pointers. If a pointer in the list does not change until the de-allocation

of the memory where the pointer is located, the learning module assumes it is genuine.

HookLocator does not check the integrity of pointers that change during their life span. The rationale here is that attractive targets for attack are the pointers that: a) have a long life span; and b) do not change throughout their life span. This allows an attacker to create a persistent change in the control flow of the system. Such pointers may live in the system for long time, which apparently slows down the learning process, since the module cannot decide whether the pointer is genuine until its memory is deallocates. Thus, in order to identify such pointers efficiently, the learning module monitors the age of a candidate pointer and, if the age exceeds a given threshold, the learning module considers it to be genuine.

Pool Monitor The sole purpose of the pool monitor is to check the integrity of function pointers and raise alerts upon any detected modifications. HookLocator maintains a monitoring-pointer list, which contains the pointers that are monitored by the module. Typically, the pointers being monitored do not change; however, if a pointer value is changed to another value within the address range of the kernel-code, or it is changed to NULL or from NULL to a value within the address range of the kernel code, the module considers the change in pointer value as legitimate; in all other cases, it raises an alert.

4 Implementation

HookLocator is fully implemented within a privileged VM and does not require modifications to the underlying VMM or running any components inside a guest VM. Thus, it works on any VMM (such as Xen, KVM, etc.) that has VMI support for physical memory analysis. Our current implementation is based on Xen (version. 4.1.2) on 64-bit Fedora 16 (version. 3.1.0-7). The Windows guests are Windows 7 SP1.

We use LibVMI [13] to introspect the physical memory of the guest VM to check the integrity of function pointers. Since LibVMI simply provides access to the raw physical memory of a guest VM, we need to bridge the semantic gap between raw memory and useful kernel data structures, which we further discuss in this section.

The extraction module builds a list of function pointers from kernel code (including kernel modules) and four tables: interrupt descriptor table, system service descriptor table, import address table, and export table. To extract information about function pointers from kernel code and import and export tables, the extraction module first gathers information about the in-memory kernel modules, including the name of each module, the module's base address, and the size of the module. Windows maintains a list of kernel modules in a doubly linked list. Each node in the list is represented by the data structure `LDR_DATA_TABLE_ENTRY`. The base address of the first node in the doubly linked list is stored in a system global variable `PsLoadedModuleList`, which is used by the extraction module to reach the list. Each `LDR_DATA_TABLE_ENTRY` also has

FLINK and BLINK fields that contain pointers to the next and previous node in the list. The extraction module uses these fields to traverse the complete list of loaded modules.

For each module in the list, the extraction module uses the base address and size of the kernel module to extract the whole module from the memory. Each kernel module is in the portable executable (PE) format, which contains headers, code and data sections, and import and export tables. The extraction module parses each kernel module in the PE format to access the tables and the code. It further processes the code using the cross-comparison and pattern matching approaches discussed previously to obtain the list of function pointers. Moreover, the extraction module reads the base address and size of the IDT from the IDTR register and copies the entire IDT into a local buffer and further processes it to extract function pointers. It also obtains the system service descriptor table (SSDT) using the approach from a Volatility plugin [14]. The extraction module further processes the SSDT to obtain function pointers of system calls.

The search module locates function pointers in the Windows kernel pools by obtaining the data from each allocation in a pool and scanning it for matching function pointers. The allocations in a pool are classified into two types based on the allocation size: small chunks, and big allocations. A small chunk requires less than a page for an allocation. On the other hand, a big allocation requires more than a page for an allocation. MS Windows keeps track of these two types of allocations in separate data structures, which are also used by the search module to track and process kernel pool allocations. The big allocations are tracked through the PoolBigPageTable, with each entry represented by a POOL_TRACKER_BIG_PAGES structure. The search module finds the location of the PoolBigPageTable in the .data section of ntoskrnl.exe. A small chunk, on the other hand, resides completely within one page and an allocated page can have several small chunks (represented by POOL_HEADER structures), which are adjacently located in a sequence. The search module finds the allocated pages and further processes them to extract chunks.

The learning module observes pointers in the learning-pointer list. If a pointer does not change during its entire life span, i.e., until deallocation of the containing block, then the learning module considers it a genuine pointer. In order to discover the current validity of an allocation, the base address of the allocation can be examined, if the allocation is a big one. If the content of that address is not 1, it means it is a valid allocation. In case of a small chunk, the learning module looks at the 1^{st} bit of the PoolType field in the chunk header POOL_HEADER. If it is set, it means the allocation is valid. There are some cases, e.g., when a pointer has a long life span, which makes the first criteria less effective for learning. Thus, the learning module uses a threshold value based on the age of the pointer. If the age exceeds a certain threshold value, it considers the pointer as genuine. The learning-pointer list maintains the creation time of a pointer entry in the list, which the learning module uses to predict the age of a pointer.

The pool monitor observes the pointers from the monitoring pointer list until the allocation where the pointer is located is de-allocated. In this case,

Table 1. Number of function pointers found in the `.text` code section of kernel and its modules by HookScout's IDA plugin (H), our cross-comparison approach (C) and our pattern matching approach (P).

	Windows 7				
	32-bit		64-bit		
Module Name	**H**	**C**	**H**	**C**	**P**
ntoskrnl.exe	5,388	5,390	401	200	2,960
hal.dll	537	537	0	0	537
ntfs.sys	147	147	0	0	416
i8042prt.sys	68	68	0	0	90
http.sys	212	212	0	0	518
disk.sys	11	11	0	0	65
volmgr.sys	18	18	0	0	35
kdcom.dll	21	21	0	0	20

the module stops observing the pointer and also deletes the pointer entry from the list. If the pointer changes to `NULL` or to any other value within the address range of kernel code, the module considers such changes as legitimate and keeps observing the pointer. Otherwise, it raises an alert.

5 Evaluation

In this section we quantify the performance of each component of HookLocator. All experiments were performed on fresh installations of Windows 7 in VMs running on Xen (version. 4.1.2). We also installed several common applications: Skype, Google Chrome, MS Office 2010, Acrobat Reader, WinDbg, CFF explorer, and WinHex in order to understand the effects of user processes on kernel heap data.

5.1 Extraction Module

We use HookScout's IDA Pro plugin [2] as a baseline for our cross-comparison approach as both tools rely on the relocation property of Windows kernel. Table 1 summarizes the number of function pointers extracted from the code section of the Windows kernel by the different approaches.

In the 32-bit case, the two methods are equally effective as they extract almost exactly the same number of function pointers (the number differs by 2 – and the exact reason for this is under investigation). In 64-bit Windows 7, HookScout and the cross-comparison approach do not work well, because the kernel code in 64-bit MS Windows 7 uses offsets to address the entry point of functions in instructions, instead of absolute addresses. Despite this fact, both the approaches showed one exception: they both obtained a small number of pointers from `ntoskrnl.exe`. HookScout finds 401 pointers (in `ntoskrnl.exe`) vs. 200 found by our cross-comparison approach. We further investigated the exception and it turns out that both methods find the pointers from the SSDT,

Table 2. Number of function pointers from different sources in 64-bit Windows 7

Function Pointer Source Name	Total Number of Function Pointers
Kernel Code	28,518
IDT	271
SSDT	401
IAT	12,142
Export Table	4,887
TOTAL	46,219
Unique Pointers	30,875

Table 3. Correlation between the allocation size and the number of function pointers

Allocation type		Total size of allocations (MB)	Total number of Pointers	Number of pointers per MB
Big allocations	Non paged pool	34.09	0	0
	Paged pool	79.13	1665	12
Small Chunks	Non paged pool	4.25	3201	753
	Paged pool	172.62	2297	13

however, HookScout analyzes the file, whereas we analyze the in-memory version. The file `ntoskrnl.exe` contains 401 absolute addresses for function pointers; however, when the kernel is loaded into memory, the addresses are adjusted according to where the kernel is loaded. After the adjustment, a new SSDT is created, which overrides the original. In the new SSDT, 201 of the 8-byte absolute pointers are replaced with 401 4-byte offsets from the base address of SSDT. Thus, half of HookScout's results are, in fact, false positives.

The byte pattern matching is much more effective and obtains an additional 2,960 pointers and this is the method we used to more accurately identify function pointers in 64-bit Windows 7. Table 2 shows a breakdown of the number of function pointers obtained from different sources. Since HookScout only works on 32-bit Windows, we present only our results for the 64-bit MS Windows 7. We found that 33.2% of function pointers are duplicates; after excluding them, we are left with 30,875 distinct function pointers, which are used by the other modules of HookLocator.

5.2 Search Module

The search module goes through all memory allocations and scans for the function pointers identified by the extraction module. There are two types of allocations – small and big – in each of the paged and non-paged kernel pools. We used HookLocator to understand the number and distribution of pointers in each of these cases. To obtain them, we ran HookLocator 10 times with a five-second delay between executions.

Table 4. Effects of user process creation on small chunks in the non-paged pool.

Process Name	Total size of Allocations (in Kilo Bytes)			Total Number of Pointers		
	Before Initiation	After Initiation	%age Increase	Before Initiation	After Initiation	%age Increase
Skype	4,717.00	4,770.83	1.14	3,456	3,499	1.25
MS Word	4,751.25	4,793.33	0.89	3,491	3,512	0.61
MS Power-Point	4,769.25	4,789.33	0.42	3,503	3,515	0.34
Acrobat Reader	4,767.25	4,840.67	1.54	3,507	3,535	0.79
Windows Media Player	4,820.75	4,928.00	2.22	3,481	3,556	2.16
Chrome	4,789.75	4,863.33	1.54	3,472	3,533	1.76

As shown in Table 3, we obtained the number of function pointers in each type of allocations and calculated the pointer density in each type of allocation. It is clear that allocations in the paged pool have a lower concentration of function pointers (12 per megabyte), which makes them less attractive for attack. More importantly, these allocations are pageable; if used by an attacker, there is always a chance that the page containing the modified pointer would be swapped out, which makes retaining control of the system flow more problematic.

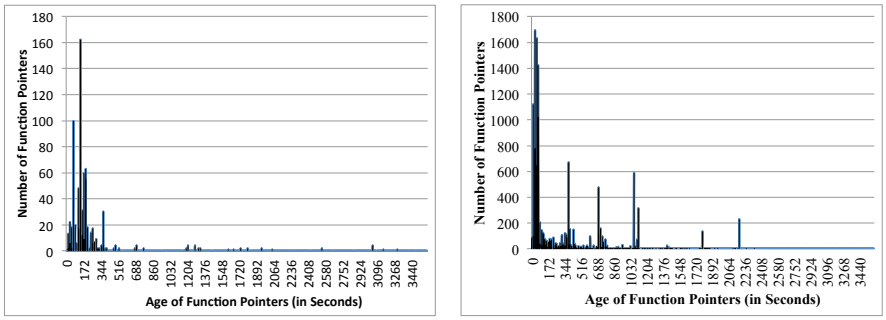
Pages in the non-paged pool always reside in physical memory, so the pointers in the pool offer a more reliable means to subvert control flow. Our experiment shows that the big allocations in the non-paged pool have no function pointers, which leaves small chunks in the pool – with relatively high concentrations of function pointers – as the most target-rich attack surface. Thus, for the rest of our discussion, we narrow our focus to the small chunks in the non-paged pool as the area to be protected.

Table 3 shows that the small chunks in the non-paged pool consist of around 4MB immediately after boot. Our first task is to understand whether there is a change in small chunk allocations and the number of pointers in the region when a user process is initiated. We obtained the total size of the small chunks in the pool before and after a user process is initiated. We ran HookLocator 10 times with a 5-second pause in between each run. The first four runs were performed before the initiation of a user process and obtained almost identical allocation sizes and number of pointers. After the fourth run, we initiated the process and obtained 6 more readings. The procedure was repeated for six different applications and their respective averages are given in Table 4.

Across the board, we see an increase of 1-2% in the size of the small chunks and the total number of function pointers. Therefore, for the rest of our evaluation experiments we consider the system in both idle and active-user state.

5.3 Learning Module and Pool Monitor

The purpose of the learning module is to validate candidate pointers identified by prior modules and present them as genuine targets for integrity monitoring. The validation is based on age of the pointer and the observation that only a small



(a) VM is not in use by a user (b) VM is actively in use by a user

Fig. 3. Life span of function pointers.

number of kernel function pointers change over their life span [2]. Therefore, to evaluate the false positives – our primary criterion for success – we study the life span of discovered function pointers and whether they are modified.

We consider two extreme cases: 1) when the machine is idle and there is no user activity; and 2) when the machine is actively in use. We use PCMark 7 [20] a performance-benchmarking tool to create a reproducible workload on the target machine, which is representative of user activities. It automatically performs several computational and IO activities on the system without any user intervention, including web browsing, image manipulation, video playback, and antivirus scanning.

We ran HookLocater for one hour to observe the function pointers in the small chunks of non-paged pool – the results are shown on Figures 3(a/b). In the idle case, 8% of function pointers end up de-allocated within the first six minutes, another 0.5% are de-allocated by the end of the hour, and 91.5% are still valid. None of the de-allocated pointers ever changed their value. In the active-user case, 69% of pointers live no longer than 20 minutes, another 3% are de-allocated by end of the experiment, and 28% are still alive. Among pointers that have completed their life cycle, none changed their values. The interesting result here is that, it is easier to identify genuine pointers when the system is actively in use.

Evidently, for a practical system we need a time frame within which to make a decision whether a pointer is genuine, or not. When a genuine pointer is identified, it is included in the genuine pointer list, which is then handed over to the pool monitor for protection. If the assigned pointer is not genuine, it will generate false alarms and render our system less useful. In other words, we need to balance two requirements – *responsiveness* (identify breaches as soon as possible) and *accuracy* (have a low false positive rate).

We use a threshold parameter that determines how long we wait before we declare a pointer genuine. Figure 4 shows the measured false positive rate as a

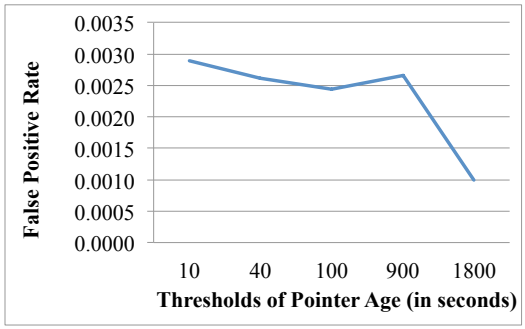


Fig. 4. False positive rate.

function of the age threshold of pointers. It is clear that even for a threshold of 10 seconds the false positive rate is 0.0028, which is a good starting point for tuning the system. Further investigation into the specific instances of false positives will, presumably, allow us to lower the rate even further but this study is outside the scope of this paper.

As a practical test of the effectiveness of the pool monitor we created a tool that synthetically mutates function pointers in the kernel pools at random. HookLocator raised an alert in all cases for suspicious changes. In the next section, we quantify the frequency at which the protection service can be run and these integrity checks be performed.

5.4 Performance Overhead of HookLocator

We built a small test environment to evaluate the performance of HookLocator. The physical machine that we used contained an Intel Core 2 Quad CPU (4 × 2.83GHz) with 4GB RAM. We used a VM running 64-bit Windows 7 over Xen (4.1.2), allocating one core and 1GB RAM to the VM. The privileged VM had Fedora 16 installed with the `3.1.0-7.fc16.x86_64` kernel.

We evaluated the HookLocator's execution performance on different types of memory allocations (big allocations/small chunks, paged/non-paged pools). We used two extreme test cases 1) when the target VM idle and 2) when it is actively in use. Moreover, in both cases, we did not run any extra services on the privileged VM in order to allow HookLocator exploit all the available resources of the VM. We obtained the HookLocator's memory overhead and the speed of scanning a type of memory allocations.

We ran the HookLocator for a minute and counted the number of scans it performed. Tables 5 and 6 summarize the results for both the test case, which show that the HookLocator is able to scan the small chunks of non-paged pool more than once per second, which makes it suitable for even real-time monitoring. HookLocator also has a low memory overhead, compared to the typical physical memory available on a modern system.

[2]Megabytes

Table 5. Performance evaluation of HookLocator when VM is idle.

Allocation Type	Memory Overhead		No. of Executions	
	MB[2]	%age	per minute	per second
Big allocations	60	2.38	31	0.52
Small chunks (Paged Pool)	44	1.74	4	0.07
Small chunks (Non-paged pool)	24	0.94	115	1.91

Table 6. Performance evaluation of HookLocator when VM is actively being used.

Allocation Type	Memory Overhead		No. of Executions	
	MB[2]	%age	per minute	per second
Big allocations	65	2.58	31	0.52
Small chunks (Paged Pool)	47	1.87	4	0.07
Small chunks (Non-paged pool)	24	0.96	101	1.68

6 Conclusion

In this paper, we argued that prior work does not provide reliable and practical protection against modern rootkits on Windows systems. Specifically, their shortcomings range from the need to analyze source code and execution in the same address space, to prohibitive overhead and no protection for the kernel heap. Also, they all focus on old, 32-bit Windows XP versions, and none is capable of working with 64-bit versions of Windows 7. To address these challenges, we developed a new approach based on VM introspection and implemented it in a tool called HookLocator. The main contributions of our work can be summarized as follows:

We address the biggest current threat to kernel hooking, which involves tampering with function pointers on the kernel heap. Microsoft's PatchGuard has effectively rendered most prior attack research ineffective, but does not protect the heap. Consequently, the kernel heap is becoming the primary vector for intercepting kernel control flow by current malware rootkits. Our approach conceptually works on both 32 and 64-bit versions of Windows. It does not rely on examining the source of the target OS; rather, it uses the relocatable property of Windows kernels as appropriate, supplemented by pattern matching to reliably identify kernel function pointers to be protected.

HookLocator works at a higher level of privilege than the potentially compromised VM, which makes it very difficult for any rootkit to cover its tracks. Further, since the tool runs in a separate VM and has a light performance footprint, it would be very difficult for a rootkit to detect that it is being monitored.

Our evaluation shows that our approach imposes minimal memory and CPU overhead, which makes it very practical. In particular, its light footprint enables real-time monitoring of the target VM, and easy path to scaling up the protection service.

References

1. G. Hoglund, and J. Butler, Rootkits: Subverting the Windows Kernel, Addison-Wesley Professional, First edition.
2. H. Yin, P. Poosankam, S. Hanna, and D. Song, HookScout: Proactive Binary-Centric Hook Detection, in Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA10), Bonn, Germany, 2010, pp. 1-20.
3. J. Nick, L. Petroni, and M. Hicks, Automated Detection of Persistent Kernel Control Flow Attacks, in Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007), Alexandria, VA, USA, 2007, pp. 103-115.
4. A. Baliga, V. Ganapathy, L. Iftode, Automatic Inference and Enforcement of Kernel Data Structure Invariants, in Pro-ceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, California, USA, 2008, pp. 77-86.
5. M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, X. Jiang, Mapping Kernel Objects to Enable Systematic Integrity Checking, in Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, IL, USA, 2009, pp. 555-565.
6. Z. Wang, X. Jiang, W. Cui, P. Ning, Countering Kernel Rootkits with Lightweight Hook Protection, in Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009), Chicago, IL, USA, 2009, pp. 545-554.
7. TEMU, http://bitblaze.cs.berkeley.edu/temu.html.
8. M. Russinovich, D. Solomon, Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition, Microsoft Press.
9. J. Butler, G. Hoglund, VICECatch the Hookers!, In Black Hat USA, July 2004, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/ bh-us-04-butler.pdf.
10. J. Rutkowska, System Virginity Verifier: Defining the Roadmap for Malware Detection on Windows Systems, In Hack In The Box Security Conference, September 2005.
11. I. Ahmed, A. Zoranic, S. Javaid, G. G. Richard III, Mod-Checker: Kernel Module Integrity Checking in the Cloud Environment, in 4th International Workshop on Security in Cloud Computing (CloudSec '12), 2012, pp. 306 313.
12. IceSword, http://www.antirootkit.com/software/IceSword.htm.
13. LibVMI, https://code.google.com/p/vmitools/.
14. SSDT Volatility, https://code.google.com/p/volatility/source/browse/trunk/volatility/plugins/ssdt.py?r=3158.
15. T. Mandt, Kernel Pool Exploitation on Windows 7, http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf
16. mxatone and ivanlef0u, Stealth hooking: Another way to subvert the Windows kernel, http://www.phrack.com/issues.html?issue=65&id=4.
17. K. Kortchinsky, Real World Kernel Pool Exploitation, http://sebug.net/paper/Meeting-Documents/syscanhk/KernelPool.pdf.
18. R. Riley, X. Jiang, D. Xu, Multi-Aspect Proling of Kernel Rootkit Behavior, In the proceedings of the 4th ACM European conference on Computer systems (EuroSys09), Nuremberg, Germany, 2009, pp. 47-60.
19. H. Yin, Z. Liang, D. Song, HookFinder: Identifying and understanding malware hooking behaviors, In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), February 2008.
20. PCMark 7, http://www.futuremark.com/benchmarks/pcmark7.