POSTER: HookLocator: Function Pointer Integrity Checking in Kernel Pools via Virtual Machine Introspection

Irfan Ahmed, Aleksandar Zoranic
Computer Science Department
University of New Orleans, LA USA
iahmed, azoranic@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.

Our novel approach to monitoring the integrity of Windows kernel pools called HookLocator is based entirely on virtual machine introspection and is the only system of its kind to allow both 32 and 64-bit versions of the Windows kernel to be monitored for function pointer integrity. 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.

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.

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) [9-11]. 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. Unfortunately, these solutions are dependent on the availability of kernel source code and thus not appropriate for closed source OS's such as MS Windows.

More recently, Yin *et al.* presented HookScout [2] for checking the integrity of function pointers in MS Windows. 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.

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.

2. Our Proposed Solution: HookLocator

HookLocator is fully implemented within a privileged VM and does not require modifications to the underlying VMM or run-

ning 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.

We use a customized version of LibVMI [8] 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 along with the low-level implementation details. Our implementation also takes into account important differences between 32-bit Windows XP and 64-bit Windows 7 internals, which are relevant to our objectives.

HookLocator consists of *extraction*, *search*, *learning*, and *pool monitor* modules. 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 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. The first method uses a cross-comparison approach and takes advantage of the relocatable property of Windows kernel 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). 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 [7]. 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 function. The extraction module uses the `0xC390` pattern to obtain a substantial list of function pointers.

We recognize the weakness of the pattern matching approach in that we may not find all the function pointers using this technique. This is why our current work in progress is to further analyze disassembled kernel code to find stronger and more generalized patterns that would extract most, if not all function pointers.

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 analyze relevant Windows 7 internals, including the data structures related to function pointers and the kernel pool, and show how these data structures can be used to extract function pointers and pool data through VMI.
- 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.

3. 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 (DIMVA'10), 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 Proceedings 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] M. Russinovich, D. Solomon, "Windows Internals: Including Windows Server 2008 and Windows Vista", Fifth Edition, Microsoft Press.

[8] LibVMI, https://code.google.com/p/vmitools/.

[9] T. Mandt, "Kernel Pool Exploitation on Windows 7", http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf

[10] mxatone and ivanlef0u, "Stealth hooking: Another way to subvert the Windows kernel", http://www.phrack.com/issues.html?issue=65&id=4.

[11] K. Kortchinsky, "Real World Kernel Pool Exploitation", http://sebug.net/paper/Meeting-Documents/syscanhk/KernelPool.pdf.