# HookScout: Proactive Binary-Centric Hook Detection*

Heng Yin[1], Pongsin Poosankam[2,3], Steve Hanna[2], and Dawn Song[2]

[1] Syracuse University, Syracuse NY 13104
heyin@syr.edu
[2] UC Berkeley, Berkeley CA 94720
{sch,dawnsong}@cs.berkeley.edu
[3] Carnegie Mellon University, Pittsburgh PA 15213
ppoosank@cs.cmu.edu

**Abstract.** In order to obtain and maintain control, kernel malware usually makes persistent control flow modifications (i.e., installing hooks). To avoid detection, malware developers have started to target function pointers in kernel data structures, especially those dynamically allocated from heaps and memory pools. Function pointer modification is stealthy and the attack surface is large; thus, this type of attacks is appealing to malware developers. In this paper, we first conduct a systematic study of this problem, and show that the attack surface is vast, with over 18,000 function pointers (most of them long-lived) existing within the Windows kernel. Moreover, to demonstrate this threat is realistic for closed-source operating systems, we implement two new attacks for Windows by exploiting two function pointers individually. Then, we propose a new *proactive* hook detection technique, and develop a prototype, called *HookScout*. Our approach is *binary-centric*, and thus can generate hook detection policy without access to the OS kernel source code. Our approach is also *context-sensitive*, and thus can deal with polymorphic data structures. We evaluated HookScout with a set of rootkits which use advanced hooking techniques and show that it detects all of the stealth techniques utilized (including our new attacks). Additionally, we show that our approach is easily deployable, has wide coverage and minimal performance overhead.

* This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, by the Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, or the Army Research Office.

C. Kreibich and M. Jahnke (Eds.): DIMVA 2010, LNCS 6201, pp. 1–20, 2010.
© Springer-Verlag Berlin Heidelberg 2010

## 1 Introduction

As malware evolves to be increasingly sophisticated and stealthy the operating system kernel has become a popular target for attacks [10]. Once the OS kernel is compromised, attackers control every aspect of the victim's system: they can implement illicit functionality directly, hide malicious user-level components, and make themselves and their components difficult to be detected and removed. To achieve these malicious goals, malware tends to make *persistent* control flow modifications, and in other words, *hooks* are installed in the victim's system. A previous study shows that the 24 out of 25 kernel rootkits in the survey make persistent control flow modifications [16].

Old-fashioned malware installs hooks by either tampering with certain kernel code regions or overwriting entries in well-known data regions. These well-known data regions include SSDT (System Service Descriptor Table), IAT (Import Address Table) and IDT (Interrupt Descriptor Table). Current hook detection tools, such as VICE [3], System Virginity Verifier [23] and IceSword [12], verify the integrity of all code regions and known data regions, and thus have successfully defeated these hooking techniques. To evade detection, malware has moved its target to previously unknown and unexplored data regions. In particular, malware overwrites function pointers in kernel data structures, which usually reside on heaps or in dynamically allocated memory pools. These kernel data structures maintain critical system states and configurations and contain important function pointers [9]. The number of function pointers in the kernel space can be large, and without in-depth knowledge of these kernel data structures, it is very difficult to locate and validate them. Therefore, this new hooking technique is ideal for attackers to install stealthy hooks.

To tackle this severe security problem, several systems have been proposed. Systems such as HookFinder [32], K-Tracer [14], and PoKeR [20] take a *postmortem* approach. These systems analyze a new kernel rootkit to extract its attack mechanism after the rootkit has caused damages and been caught. Then the extracted hooking mechanisms can be used to update the hook detection policy for guarding against similar attacks in the future. However, postmortem analysis is not an effective defense because a large number of kernel objects and function pointers exist in memory, and the number of potential locations for placing this kind of hook is enormous. This means that even when a new attack region is discovered and blocked, attackers can simply locate and exploit another data structure to achieve the same objective.

Other systems like SBCFI [16], Gibraltar [1], HookSafe [31], and SFPD [4] take a *proactive* approach. Instead of dissecting malware to figure out what regions have been used for placing hooks, these systems examine the operating system to understand where these function pointers are and how they are used, and then generate a hook detection policy. This policy can be used to traverse kernel data structures, locate function pointers in these kernel objects, and determines if they point to legitimate targets. In order to know how to traverse kernel data structures, these systems perform static source code analysis, extract type graphs, and generate traversal templates. However, in many cases, we do not have

access to the source code of the operating system, such as Microsoft Windows. Therefore, the requirement of access to source code would impede third-party security practitioners to deploy these systems. Moreover, since we do not have source code of third-party device drivers and modules, hooks in these components will also be overlooked by static source code analysis.

In this paper, we take a proactive approach. We first systematically study the attack space and nature of this new hooking technique. We perform *whole-system dynamic binary analysis* to monitor kernel memory objects and keep track of function pointers propagating in the kernel space. By directly observing how the operating system is operating at the binary code level, we conduct a quantitative measurement study on this attack vector. To further demonstrate that this new threat is realistic, we implement two keyloggers by exploiting two different function pointers. Since these two attacks are new, they can successfully evade all the existing hook detection tools. To effectively defeat this threat, we propose a novel approach for proactive hook detection. We aim to derive the hook detection policy directly from the knowledge about kernel memory objects and function pointers. Compared to previous approaches, our approach is binary-centric. That is, it performs analysis directly on binary code, without assuming the access to source code. Therefore, this approach can be widely deployed on the system with closed-source OS kernel and third-party components.

To demonstrate the efficacy of our approach, we built a prototype, called *HookScout*. It consists of two subsystems: *analysis subsystem* and *detection subsystem*. The analysis subsystem performs binary code analysis on the operating system kernel and automatically generates a policy for hook detection. The detection subsystem residing on the user's machine enforces the generated policy and detects hooks in the kernel space.

In summary, this paper makes the following contributions:

- To assess the attack space of function pointer hooking technique, we conduct a systematic measurement study. It shows that the space of this new attack vector is enormous: there are around 18,000 function pointers in the Windows kernel space in total, the majority of these function pointers (90%) are long-lived, and very few (3%) ever change in their lifetime.
- To further demonstrate the severity of this problem, we identify two function pointers in the keyboard driver, and implement two new keyloggers by exploiting these two function pointers individually. These two new attacks can successfully evade the existing hook detection tools.
- We propose a binary-centric approach for generating a hook detection policy.
- We design and implement a prototype called HookScout, to demonstrate the effectiveness and efficiency of our approach.
- We evaluated HookScout with a popular closed-source operating system, Windows XP with Service Pack 2. The analysis subsystem can generate the hook detection policy within a few hours. The generated policy can achieve very high coverage (over 95%). The detection subsystem was able to detect all the rootkit samples in our sample set, including the two new synthetic attacks. We also showed that the performance overhead of this detection component is negligible.

```
typedef struct {
  int type;
  char name[512];
} OBJ_HEAD;

typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int (*open)(char *n, char *m);
  ...
} FILE_OBJ;

typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int state;
  int (*ioctl)(char *buf, int size);
  ...
} DEVICE_OBJ;
```

```
LIST_ENTRY ObjListHead;

CreateFile() {
  FILE_OBJ *f = malloc(sizeof(FILE_OBJ));
  ...
  InsertTailList(&f->link, &ObjListHead);
  ...
}

CreateDevice() {
  DEVICE_OBJ *d = malloc(sizeof(DEVICE_OBJ));
  ...
  InsertTailList(&d->link, &ObjListHead);
  ...
}
```

**Fig. 1.** Code snippet that illustrates a polymorphic linked list. The doubly-linked list starting with `ObjListHead` contains objects of two different types, `FILE_OBJ` and `DEVICE_OBJ`. This code snippet is a simplified example inspired by the Windows kernel hash table for organizing kernel objects.

## 2 Problem Statement

In this paper, we take a proactive approach to detecting function pointer hooking attacks. That is, we want to generate a hook detection policy that can be used to thoroughly locate and validate the function pointers in the kernel space. To facilitate deployment, we want to directly derive the hook detection policy from the binary code of OS kernel and device drivers. As a result, our technique can be widely used to protect closed-source operating systems like Windows and proprietary device drivers.

Furthermore, we have to cope with type polymorphism. That is, the actual type of a polymorphic data object is determined by the context under which this data object is created. Figure 1 illustrates such a case. A linked list `ObjListHead` stores objects of two different types, `FILE_OBJ` and `DEVICE_OBJ`. These two types share a common head structure `OBJ_HEAD`, while the remaining portions in these two types are different. The function `CreateFile` creates a `FILE_OBJ` object, and the function `CreateDevice` creates a `DEVICE_OBJ` object. If we are not aware of the different creation contexts of these two types of objects, we will not notice this type polymorphism, and thus will not locate and traverse the function pointers in these objects. Indeed, in the Windows kernel, many different types of kernel objects, such as files, devices, drivers, and processes, are managed in a centralized hash table [24]. These kernel objects keep important system states and function pointers. Thus, it becomes critical to traverse and verify the function pointers in these polymorphic data structures.

## 3 Approach Overview

At a high level, our approach consists of two subsystems: *analysis subsystem* and *detection subsystem*. The analysis subsystem performs static and dynamic binary analysis on a given distribution of an operating system, and generates a policy for hook detection. The detection subsystem is deployed on users' machines with the same distribution of the operating system installed. The detection subsystem enforces the policy generated by the analysis subsystem and actively detects hooks at runtime. Note that the system protected by the detection subsystem does not need to be the same as the one analyzed by the analysis subsystem. These two systems only need to have the same set of binary modules (including main kernel modules and common device drivers). For instance, if the analysis subsystem generates a policy for Windows XP Professional SP2, then this policy can be used for hook detection on any machines with Windows XP Professional SP2 installed. Of course, when a new kernel update is released, we need to generate a corresponding policy for it. Since our system can generate the new policy in a fully automatic manner within a few hours (as demonstrated in Sect.5.2), we believe our approach is practical for wide deployment. In this section, we give a description of the analysis subsystem and the detection subsystem.

### 3.1 Analysis Subsystem

We perform *whole-system dynamic binary analysis* on the operating system for which we want to generate the hook detection policy. In other words, we run the entire installation of an operating system along with common applications, and observe how the OS kernel behaves. In particular, we are interested in the kernel's behaviors in two aspects: (1) because function pointers become the targets for installing hooks, we want to know how function pointers are created, distributed, and used; and (2) we want to monitor memory objects that are allocated either statically or dynamically. Then we can have a complete view of the kernel memory space, in terms of where memory objects are and where function pointers are located within these memory objects. Such a complete view enables us to quantitatively and qualitatively assess the space and characteristics of kernel hooking attacks, and helps us determine appropriate detection policies.

Furthermore, we want to generate the hook detection policy by inferring invariants from this complete view (or more precisely, a series of views). In particular, we need to determine the layout of each memory object, in terms of where the function pointers are located within the memory object and what properties these function pointers have (e.g. whether they change over time). This process is essentially analogous to inferring the type of a memory object.

In order to address polymorphic data structures, we propose a context-sensitive analysis technique for inferring the policy. We take into consideration the execution context where each memory object is created. We rely on the fact that memory objects created in the same execution context are of the same or compatible types. That is, these memory objects should have the same or compatible layouts.

For the example in Fig.1, all the memory objects created in `CreateFile` are of type `FILE_OBJ`, and all the objects allocated in `CreateDevice` are of type `DEVICE_OBJ`.

By tracking function pointers and monitoring memory objects, we are able to obtain the concrete layout for each memory object at a specific moment (i.e. exactly where the function pointers exist in an object). In order to locate and validate function pointers in the future, we need to extract a generalized layout for all the memory objects that are created in the same execution context. To this end, we devise a *generalization process*, which produces a generalized layout for a given execution context by merging concrete layouts of multiple memory objects created under that context. Such a generalized layout associated with the execution context is a *context-sensitive template* in our policy. As a result, the generated policy consists of a list of context-sensitive templates.

### 3.2 Detection Subsystem

To enforce the generated policy, the detection subsystem needs to be context-sensitive as well. That is, the detection subsystem monitors the allocation and deallocation of memory objects, extracts the execution context when each memory object is created, and looks up the policy template corresponding to this execution context. Then according to the template associated with this memory object, the detection subsystem will periodically verify the validity of function pointers in this memory object. Continuing with the example given in Fig.1, we would monitor memory objects created by `CreateFile` and `CreateDevice`. The creation context will be used to look up policy. Therefore, the policy template applied to the memory objects created by `CreateFile` will be different than the one applied to those created by `CreateDevice`.

## 4 System Design and Implementation

To demonstrate the feasibility of our approach, we design and implement a system, called HookScout. We illustrate the architecture of HookScout in Fig.2. The analysis subsystem consists of two components: *monitor engine* and *inference engine*. The monitor engine watches the behaviors of the operating system

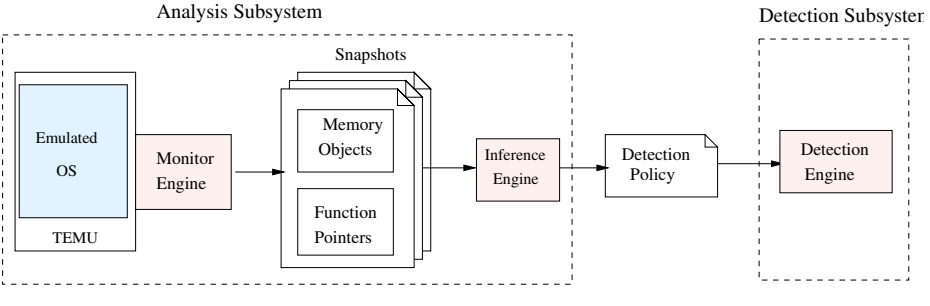


**Fig. 2.** Architecture of HookScout

of interest. More specifically, it monitors memory objects that are created either statically or dynamically, and keeps track of function pointer propagating in the kernel memory space. To perform this fine-grained dynamic binary analysis, we build the monitor engine on top of TEMU [33, 28]. TEMU is a dynamic binary analysis platform based on an open-source whole-system emulator, QEMU [2]. During the dynamic analysis, the emulated operating system is exercised with common test cases, and the monitor engine periodically records system snapshots, including the state of memory objects and function pointers. Taking the snapshots as inputs, the *inference engine* performs context-sensitive analysis and generates the policy for hook detection. In the detection subsystem, the *detection engine*, located in the system to be protected, enforces the policy generated by the analysis subsystem and detects hook in the kernel space at runtime.

### 4.1 Analysis Subsystem

**Monitor Engine.** The monitor engine is responsible for: (1) monitoring memory objects; (2) tracking function pointers; and (3) periodically generating snapshots of the OS kernel.

*Monitoring Memory Objects.* The monitor engine watches memory objects that are allocated either statically or dynamically. A static memory object is a memory region statically allocated for a kernel module for storing global variables, while a dynamic memory object is allocated dynamically from heaps and memory pools. To monitor kernel memory objects, we need to have basic knowledge about kernel memory management. For Windows, we know that `MmLoadSystemImage` is used to load a kernel module. `RtlAllocateHeap` and `RtlFreeHeap` are used for heap allocation and deallocation. Additionally, `ExAllocatePoolWithTag` and `ExFreePoolWithTag` are the root APIs for allocating and freeing memory pools. We intercept these kernel functions. When a memory object is newly allocated, we extract its base address and size and keep this information in the memory object state. We maintain the information for static and dynamic memory objects in an active memory object list. When a memory object is freed, we simply remove its information from the active memory object list. Some memory objects are special and are statically allocated and pointed by system registers. For example, `IDTR` is a register pointing to a static memory region for storing interrupt descriptor table and `FS` is a segment register pointing to a static memory region for storing the current execution context in Windows. Since these special static memory regions may contain function pointers, we also monitor these objects.

For dynamically allocated memory objects, we also need to obtain the execution contexts when they are created. The execution contexts are later used by the inference engine to perform context-sensitive analysis and generate policy. We will describe how to obtain the creation context while discussing the inference engine.

*Tracking Function Pointers.* The monitor engine identifies where each function pointer is initialized and then keeps track of the function pointer as it propagates throughout the system.

To identify the initial function pointers, we leverage the following fact: in Windows (or other relocatable OS kernels), all the modules (including the kernel itself) are generated to be relocatable. All references with absolute addresses to the statically allocated code and data sections for each kernel module have to be placed in the relocation table (e.g., `.reloc` for PE format). In this way, if the executable loader decides to load a kernel module into a different memory region than assumed, it can go through this relocation table to update these references. Due to the fact that a function pointer refers to the absolute address of a function within a relocatable module, it must appear in the relocation table. Then to determine initial assignments of function pointers, we can check for each entry in the relocation table whether it points to a function entry. Function entries can be determined through standard static binary analysis.

```
0005ed61: mov  [ebp-50h], 00015141h
```

For example, the instruction shown above moves a constant number into a memory location. This constant's location (0005ed64h[1]) appears in the relocation table and the actual value ($00015141h$) of this constant points to the entry point of a function. Then we can determine that this instruction copies a function pointer into a memory location on the stack.

Moreover, an instruction may also reference a function from another module as a function pointer. In this case, this function appears in the import address table (i.e., IAT).

```
000146ae: mov  eax, ds:[00013464h] ; READ_PORT_UCHAR
000146b3: mov [0001390ch], eax
```

For example, the two instructions above moves a function READ_PORT_UCHAR defined in IAT to a global variable located at 0001390ch. Therefore, we need to check IAT for initial function pointers as well.

We developed a plugin to IDA Pro [13] to perform this static analysis. This plugin takes a kernel module as input, automatically enumerates the entries in the relocation table and import address table, identifies the function boundaries, and determines the locations of initial function pointers. By performing this analysis on all kernel modules (including device drivers), we have identified all the initial function pointers in the kernel.

Then, to keep track of function pointers propagating over the system, we perform whole-system dynamic taint analysis, as many previous systems do [7, 34, 32, 5, 6]. That is, we mark the initial function pointers as tainted, and during the execution of each instruction, if any source operand is tainted, we mark the destination operand is tainted by checking data dependency between operands. In this way, we can track which data structures and locations these function pointers are copied into. In the implementation, we make use of the taint analysis functionality in TEMU.

[1] The instruction starts at 0005ed61h. The first three bytes are used for opcode and the first operand. So this immediate operand is located at 0005ed64h.

Therefore, relying on the relocatable property of initial function pointers and dynamic taint analysis, we can identify the vast majority of function pointers (if not all) in the kernel memory space. For the OS kernels that are not relocatable (e.g., Linux), we cannot use this technique. Alternatively, we can examine the concrete value for each memory word within a memory object to see if it points to a kernel function entry.

**Inference Engine.** The inference engine takes the system snapshots as input, performs context-sensitive analysis, and infers a policy for hook detection.

*Determining Execution Context.* In general, we want to know who creates a memory object. From the binary code point of view, this information can be obtained from the call stack when the memory allocation routine is invoked. From the call stack, we obtain the return address of the memory allocation function call. Considering that the function that invokes the memory allocation routine is called by another function, we actually obtain a chain of return addresses. Therefore, we define the execution context to be a chain of return addresses and the size to be allocated. Taking into account that kernel modules can be relocated to different locations in different executions and different systems, for each return address, instead of the absolute address, we keep the relative address — the offset to the base of the module where this return address is located.

Note that the number of return addresses to be included determines the level of context sensitivity in our analysis. The more return addresses, the more context-sensitive our analysis is. For example, if function A and function B call function C, and function C allocates memory objects for A and B, the analysis with only one return address will think memory objects created in C are of the same type, which may not be true. In comparison, the analysis with two return addresses will treat memory allocated for function A and B differently. Hence, the increase of context sensitivity results in better analysis precision. However, the increase of context sensitivity also leads to more complexity in our analysis. First, it means that we need to perform more thorough test cases to cover more execution contexts. Second, it means the number of templates in the policy would increase drastically. Therefore, we need to determine an appropriate level of context sensitivity. Fortunately, as shown in Sect.5.2, analysis with very small number (1 to 3) of return addresses can already generate high-quality policies with very high coverage.

*Inferring Policy Templates.* We merge the layouts of multiple dynamic memory objects with the same execution context into a generalized layout. Static memory objects are different because they are not associated with execution contexts so we uniquely identify them by their names (e.g., module names or register names). Thus, for static memory objects, we merge them according to their names.

Within a memory object, we classify each field (e.g., 4-byte memory in 32-bit architecture) into one of the following types: NULL, FP, CFP, and DATA. NULL is for a field that holds a concrete value 0. FP identifies a function pointer, which we determine by checking if this field is tainted. CFP indicates a constant function pointer that has never changed its value in its lifetime. To determine a

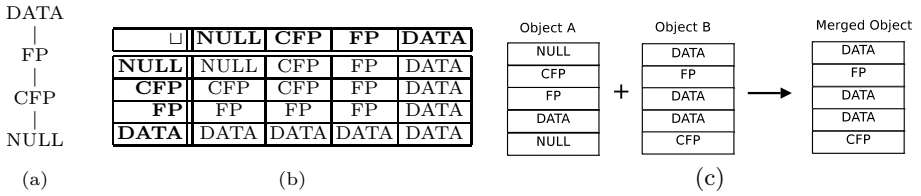


**Fig. 3.** Join operator ⊔ used in merging object layouts, shown using a lattice (a), an operation table (b), and an example (c)

CFP, we check if this field is tainted in the current snapshot, and its concrete value remains unchanged in previous snapshots since this field is initialized. Thus, CFP is a subset of FP. DATA specifies a field that holds a data value, which is not tainted and does not hold a concrete value 0.

To merge a set of observed object layouts into a single generalized layout, we conservatively infer the most general type for each field, according to the ordering shown in Fig.3 (a). In the order NULL, CFP, FP, and Data, each type covers more possibilities than the earlier ones, so we generalize to the most specific type that includes all observations. This generalization corresponds to the join operator ⊔ in a simple linearly-ordered lattice. A corresponding matrix for this join operation ⊔ is also shown in Fig.3 (b). For instance, if one type is DATA and the other is a function-pointer type FP or CFP, the field might contain either a function pointer or data. To be conservative, we mark it as DATA in the generalized layout. Similarly, if a function-pointer field was sometimes constant and sometimes not constant, it is conservatively non-constant in the merged layout: CFP ⊔ FP = FP. We illustrate a concrete example how two memory objects are merged in Fig.3 (c).

As we will show in Sect.5.1, the vast majority of function pointers are constant. In other words, they never change during their whole lifetime. Thus, the generalized layouts can be directly used as a policy to detect hooks that make modifications on these constant function pointers. In the current implementation of HookScout, we employ this simple policy. This policy does not protect non-constant function pointers. We leave it as our future work to investigate more sophisticated policies for protecting non-constant function pointers. Note that so far the generated policy is a raw policy, including all templates. For the final policy to be enforced on users' machines, we only need to include the templates that contain CFP fields, which is only a small portion of all templates, as shown in Sect.5.2.

### 4.2 Detection Subsystem

The detection engine resides on a user's machine to detect violations of the hook detection policy generated by our analysis subsystem. We are aware that the detection engine can be implemented in at least two ways. First, it can be implemented as a kernel module inside the protected operating system. Second,

it can be implemented inside a virtual machine monitor to detect attacks happening in a virtual machine. While the first approach is easy to implement and deploy, the second approach is more resilient to various attacks. In the current implementation of HookScout, we implement a proof-of-concept detection engine as a kernel module, mainly for demonstrating the effectiveness of our approach. We realize that malware is able to subvert our detection component, like any other security products sitting in the same execution environment as malware. We leave a more secure implementation as future work.

In the kernel module, we intercept the same set of kernel functions for monitoring memory objects, as those in the monitor engine. When a memory object is created, we extract its execution context and determine if there is a policy template associated with this execution context. If not, we skip this memory object. For those memory objects that are associated with policy templates, we periodically check if the constant function pointers within them hold different values than before. A different value indicates a hooking attack. When a memory object is freed, we remove it from the active object list.

As the kernel functions to be intercepted are not in the SSDT, SSDT hooking is not an option to hook these functions. Instead, we hot patch the entry of each of these functions. That is, we place a `jmp` instruction into the function entry, making the execution redirected into the detection engine. The kernel module is configured to be loaded at the earliest stage of boot time, in order to monitor the memory objects as early as possible.

Note that this periodical checking approach may still leave room for transient attacks on function pointers, depending on the frequency of checking. A more secure approach is to enforce our detection policy with HookSafe [31], where a shadow memory is maintained for protected function pointers and unauthorized changes can be detected instantly.

## 5 Evaluation

In the experiments, we aim to evaluate our system in the following aspects. In Sect.5.1, we quantitatively assess the attack space and characteristics of kernel-space hooking attacks. In Sect.5.2, we evaluate the analysis subsystem of HookScout, with respect to the coverage rate of the generated policy, the influence of context sensitivity to the quality of the generated policy, and performance overhead. In Sect.5.3, we evaluate the detection subsystem of HookScout, in terms of detecting real-world kernel rootkits, false alarms, and performance overhead.

*Experiment Setup.* Our experiments proceeded as follows. We first ran the analysis subsystem of HookScout to monitor and analyze a given operating system. To demonstrate that HookScout can work with closed-source operating systems, we chose Windows XP Professional Edition with Service Pack 2, a popular platform targeted by the majority of malware samples. During the analysis, we exercised the monitored operating system with a series of test cases that activate various

OS subsystems, including filesystem, networking, process and thread management, and so on.

It took approximately 25 minutes to boot up the Windows XP with our monitor engine and execute the test cases. Meanwhile, the monitor engine recorded system snapshots every 15 seconds. The snapshot contains the states of memory objects and function pointers. Therefore, 100 snapshots were recorded for each run. In total, we performed 3 different runs, which rendered a total of 300 snapshots. Then on these snapshots, we assessed the attack space and characteristics, and generated policy for hook detection.

We ran the analysis subsystem of HookScout on a Linux machine with a dual-core 3.0GHz CPU and 4GB RAM. We ran a Windows XP Professional SP2 disk image inside QEMU with 512MB allocated memory. We installed the detection subsystem on a machine with a 3.0GHz CPU and 4GB RAM and Windows XP Professional SP2.

### 5.1 Attack Space and Characteristics

By monitoring system execution and tracking function pointers in the kernel, we are able to assess the attack surface and characteristics of potential kernel hooking attacks.

First of all, we want to know how many function pointers exist in kernel space during the execution. This indicates the space of this attack vector. To explore this question, we picked the first run, and for each snapshot in that run, we counted the total number of function pointers in that snapshot[2]. Figure 4 shows the total number of function pointers over the 25-minute execution. We can see that the total number of function pointers climbs up in the first 5 minutes of system boot-up, and then fluctuates around 18,000 during the execution of test cases. If every function pointer could be potentially exploited, the space of kernel hooking attacks is enormous. Figure 4 also shows the number of function pointers in dynamically allocated memory objects. Because these function pointers cannot be easily located and verified by traditional rootkit detection methods, they are more attractive to attackers. We can see that the number of function pointers in dynamically allocated memory objects is fairly high, around 8,000. Therefore, there is a large attack surface for attackers to utilize in the OS kernel.

Then, we want to know how long these function pointers live in the kernel space. Since we aim to detect persistent control flow modifications, attacks would target at long-lived function pointers instead of transient ones. Therefore, we want to know how many function pointers are long-lived. We used the last snapshot in the first run as a starting point, and looked backward at each of previous snapshots. If we see a function pointer exists in one snapshot but not in the snapshot before it, we treat this snapshot as the birth time of this function pointer. Figure 5 shows the cumulative distribution function (CDF) of the function pointers' lifetime in the last snapshot of the first run. We can see that around 10% function pointers only lived less than two minutes, and approximately 90% function pointers lived longer than 17 minutes, and very few lived in between.

[2] Note that all runs had similar characteristics.

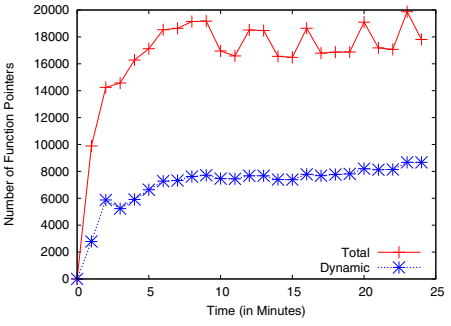


**Fig. 4.** Attack Space

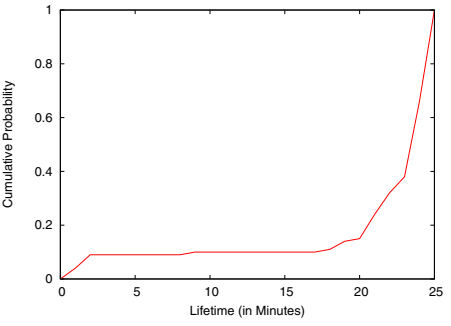


**Fig. 5.** Lifetime Distribution

Moreover, we want to know how frequently these function pointers change their targets during the execution. To answer this question, we examined all these snapshots, and for each of function pointers in these snapshots, checked if its concrete value was different in any of previous snapshots during its lifetime. We observe that up to 3.63% function pointers have ever changed during their lifetime. This observation indicates that a simply policy would suffice to validate the vast majority of function pointers.

*Two Synthetic Keyloggers.* To further assess the severity and practicality of function pointer hooking attack, we play on the attacker's side. We implemented keystroke sniffing functionality by tampering with function pointers. We performed a combination of dynamic and static binary analyses to reverse engineer a small part of kernel code related to keystroke processing. We sent some keystrokes into the emulated system and collected an execution trace for the guest kernel. Through dynamic taint analysis, we tracked how keystrokes propagate in the kernel space. In consequence, we identified several code regions that are relevant to keystroke processing. Then we statically examined these code regions using IDA Pro. It took one of the authors only a few hours to identify two function pointers (one in static memory region allocated in the keyboard driver `i8042prt.sys`, and the other in a dynamic memory region) that can be individually exploited to intercept keystrokes. To confirm that these two function pointers can be exploited indeed, we implemented two keyloggers, named `keylogger-1` and `keylogger-2`, to exploit these two function pointer respectively. We are not aware that such attacks have appeared in the literature and existing malware attacks. As shown in Sect.5.3, these two keyloggers evade the existing detection tools except HookScout. This experiment demonstrates that it is absolutely feasible for attackers to implement illicit functionalities by using this stealthy attack technique.

### 5.2 Policy Generation

Now we evaluate the analysis subsystem of HookScout. In particular, we are interested in how context sensitivity affects the coverage of the generated policy.

**Table 1.** The coverage and size of policy influenced by the level of context sensitivity

| Level | Coverage | | Templates | |
|---|---|---|---|---|
| | AVG | STDEV | Raw | Final |
| 1 | 94.67% | 2.97% | 3518 | 308 |
| 2 | 96.10% | 1.92% | 4285 | 405 |
| 3 | 96.74% | 1.64% | 5270 | 511 |

The coverage is measured as a ratio of the number of function pointers identified by the policy to the total number of function pointers. In addition, we want to see how context sensitivity affects the size of the generated policy. To measure the coverage, we used the snapshots from the first two runs to generate policy, and then applied the generated policy to the snapshots from the third run.

We listed the experimental results in Table 1. We measured the coverage for each snapshot in the third run. In Table 1, we summarized these results by calculating the average and standard deviation of the coverage. For the size of the generated policy, we listed the number of templates in the raw policy and the number of templates in the final policy respectively. We make the following observations: (1) the generated policies can achieve very high coverage, even with 1 level of context sensitivity; (2) with an increase of context sensitivity, coverage is increased accordingly; and (3) the size of policy (i.e. the number of templates) is increased considerably with the increase of context sensitivity, but the absolute number is still fairly small. Considering that 3-level context sensitivity can achieve the highest coverage and reasonably small policy size, we chose to generate a policy with 3-level context sensitivity.

It took approximately 70 seconds to process one snapshot, and around 4 hours in total to generate a policy from 200 snapshots. Due to the fact that we only need to generate one policy for each version of OS kernel and can distribute it to all machines with the same OS kernel installed, we believe that this execution time is acceptable. Moreover, the task of policy generation can be easily partitioned and parallelized, which would increase the performance significantly.

### 5.3 Hook Detection

We evaluated three aspects of the detection subsystem of HookScout. First, we compiled a set of kernel rootkit samples to evaluate the effectiveness of the detection subsystem. Second, we measured its performance overhead. Third, we evaluated the occurrence of false alarms.

*Detecting Kernel Hooks.* We obtained a set of kernel rootkits from public resources [21, 17] and collaborative researchers. We selected the rootkit samples that are known to install kernel hooks and are able to run in our test environment. We also included the two synthetic keyloggers in the experiment to evaluate how effective the existing detection tools and HookScout are in terms of detecting new attacks. As a comparison with HookScout, we chose the following hook detection tools: IceSword [12], VICE [3], and RAIDE [19]. System

**Table 2. Detection Results of Four Tools.** I stands for IceSword [12]), V for VICE [3], R for RAIDE [19], and H for HookScout.

| Sample Name | Hooking Region | I | V | R | H |
|---|---|---|---|---|---|
| HideProcessHookMDL [21] | SSDT | ✓ | ✓ | ✓ | ✓ |
| Sony Rootkit [26] | SSDT | ✓ | ✓ | ✓ | ✓ |
| Storm Worm [27] | SSDT | ✓ | ✓ | ✓ | ✓ |
| Shadow Walker [21] | IDT | ? | ✓ | ✓ | ✓ |
| basic_interrupt_3 [21] | IDT | ? | ✓ | ✓ | ✓ |
| TCPIRPHOOK [21] | Tcp driver object | × | ✓ | ✓ | ✓ |
| Rustock.C [22] | Fastfat driver object | × | × | ✓ | ✓ |
| Uay Backdoor [29] | NDIS data block | × | × | ✓ | ✓ |
| Keylogger-1 | Kbd static data region | × | × | × | ✓ |
| Keylogger-2 | Kbd dynamic data region | × | × | × | ✓ |

Virginity Verifier [23] did not function correctly in our testing environment, so we did not include this tool in the experiment.

We listed the detection results in Table 2. We can see that all detection tools, including HookScout, are able to detect SSDT hooks, and all except IceSword are able to detect IDT hooks. IceSword displays only the content of IDT and requires manual inspection to determine if there is a hook, so we leave a "?" mark for IceSword. TCPIRPHOOK [21] and Rustock.C [22] hook function pointers in `Tcp` and `Fastfat` device driver objects respectively. IceSword does not inspect kernel objects, and thus cannot detect these hooks. While RAIDE checks both `Tcp` and `Fastfat`, VICE only checks `Fastfat` object. Uay Backdoor [29] modifies function pointers in the NDIS data structure maintained for the TCP/IP network protocol. IceSword and VICE cannot detect these hooks installed by Uay Backdoor. However, RAIDE has another special policy for checking the registered network protocol list, and thus can detect these hooks successfully. By exploiting new function pointers, our two synthetic keyloggers, keylogger-1 and keylogger-2, can evade all the detection tools in our experiment, except HookScout.

As compared to the other three detection tools, HookScout is able to detect all the samples in this set. The key difference between HookScout and the other tools is that HookScout is equipped with much more thorough detection policy, which is automatically generated by the analysis subsystem, whereas the other tools have very limited policies that are manually defined. Given the high coverage of our automatically generated policy, HookScout is substantially more difficult to evade.

It is worth noting that TCPIRPHOOK, and Rustock.C tamper with function pointers in kernel objects organized in the polymorphic hash table [24]. Even with access to the source code of Windows kernel, context-insensitive analysis approaches (such as SBCFI [16] and Gibraltar [1]) would not identify these function pointers. By contrast, with context-sensitive policy inference and hook detection, HookScout can automatically generate policy and validate these function pointers successfully. Moreover, Keylogger-1 exploits a function pointer in

**Table 3.** Performance Overhead of the Detection Engine

| Workload | w/o HookScout | w/ HookScout | | Slowdown | |
|---|---|---|---|---|---|
| | | 1s | 5s | 1s | 5s |
| Boot OS | 19.43 s | 20.70s | 20.43 s | 6.5% | 5.1% |
| Copy directories | 7.57 s | 8.09s | 7.68 s | 6.9% | 1.5% |
| (De)compress files | 23.84 s | 24.44s | 23.51 s | 2.5% | -1.4% |
| Download a file | 23.59 s | 24.49s | 24.42 s | 3.8% | 3.5% |

the keyboard device driver. Without source code of this driver, source code analysis approaches [16, 1, 31, 4] will be completely unaware of function pointers defined in this driver.

*Performance Overhead.* To observe how HookScout affects performance, we performed several workloads and measured their execution times with and without the detection engine installed. We also measured the performance with two different checking intervals: 1 and 5 seconds. The workloads include booting Windows, copying a directory structure, performing compression and decompression of a directory structure with `7zip`, and downloading a file with `wget`. The total size of the directory structure is 75MB. The size of the downloaded file is 100MB. Table 3 shows the execution time for each workload. Each workload is performed 7 times and the average of 5 non-minimum/maximum runs is reported. In all, the slowdown caused by HookScout is about 4.9% and 2.1% for the checking intervals of 1 second and 5 seconds respectively.

*False Alarms.* To evaluate the occurrence of false alarms, we installed HookScout detection engine on a healthy system (without rootkits installed), and kept it running for eight hours. Meanwhile, a user operated on this machine for his regular computing tasks. We did not observe any false alarms during this period.

## 6 Discussion

In this section, we discuss the feasibility of potential evasion techniques against HookScout and our countermeasures.

*Exploit Uncovered Function Pointers.* Attackers can perform the same analysis on function pointers, and determine which function pointers would not be located and verified by HookScout. Then they can target these function pointers to evade HookScout. First of all, our policy generation technique can achieve over 95% coverage, so we have substantially reduced the space of this attack (e.g., from 18,000 to 900). In addition, it may be impractical for attackers to take advantage of many of these uncovered function pointers, because of their mutable nature: sometimes they are function pointers and sometimes not. Furthermore, for this small number of uncovered function pointers, defenders can investigate these cases first and manually define special policies for the plausible attacks.

*Exploit Uncommon Proprietary Device Drivers.* QEMU has emulated a set of common hardware devices. For these common devices, HookScout can generate policy to validate functions pointers in the corresponding device drivers. For those devices that are not supported by QEMU, HookScout cannot generate policy to protect their drivers, since these drivers are not installed and activated during the analysis phase. This limitation is not specific to HookScout. No previous solutions are able to address this issue, which requires further investigation.

*Attack Limited Test Cases.* Our hook detection policy is derived mainly using dynamic analysis. In principle, the quality of dynamic analysis is largely determined by the completeness of test cases. Attackers could potentially exploit those function pointers that were not initialized and moved around during our analysis but will be activated in other test cases. Then HookScout would not detect these attacks. By exercising the analysis subsystem with a more complete set of test cases that exercises more kernel functionalities, this problem can be alleviated. Moreover, if attackers only target the function pointers that are only initialized and moved around in uncommon situations, it means the attacks will not become effective most of time.

*Subvert or mislead HookScout.* The detection engine of HookScout is implemented as a kernel module installed in user's system, and thus is subject to complete subversion just like any other security tools running in the same privilege as malware. In addition, malware may mislead HookScout by injecting fake events or changing the existing events that HookScout monitors. In particular, HookScout relies on proper functionality of kernel memory management routines and correct call stack information to monitor memory objects. More secure implementation based on virtual machine techniques can significantly raise the defense bar against this kind of attacks. For example, Payne et al. systematically discussed the challenges of secure active monitoring, proposed a series of solutions and built a framework called Lares [18].

## 7 Related Work

*Postmortem Analysis.* Several systems have been proposed to facilitate understanding of rootkit's behaviors. HookFinder [32] can automatically identify and understand how a rootkit installs hooks. K-Tracer [14] and PoKeR [20] are profiling tools for monitoring rootkit behaviors in general, including hooking behaviors, data structure manipulation and others. The better understanding of a new kernel attack can then be used to harden the security policy against similar attacks. As our study shows, the attack space for kernel function pointer hooking is vast. After a function pointer is known to be exploited, attackers may easily switch to exploit another function pointer.

*Proactive Defense.* The first line of defense against kernel attacks is to prevent untrusted code execution in the kernel space. Several systems leveraged the virtual machine based architecture to monitor and enforce the kernel code integrity.

Livewire [8] was the first proposal to make use of virtual machine monitor to monitor system integrity, including verifying the kernel code regions and examining specific data attacks by querying the system states. SecVisor [25], is a tiny hypervisor (i.e. virtual machine monitor) that ensures code integrity for commodity OS kernels. Patagonix [15], is another system based on hypervisor to identify covertly executed binaries. This line of defense can be circumvented by return-oriented rootkits [11], which take advantages of existing kernel code to build illicit functionalities.

The second line of defense is to enforce control flow integrity. SBCFI [16], Gibraltar [1], SFPD [4], HookMap [30], HookSafe [31], and HookScout belong to this category. These system may still catch return-oriented rootkits, as long as persistent control flow modifications are made. SBCFI [16], Gibraltar [1], and SFPD [4] perform source code analysis on the OS kernel to derive the security policy. The requirement of source code would impede their deployment on closed-source operating systems and proprietary device drivers. SBCFI and Gibraltar need manual annotations for generic pointers and perform context-insensitive analysis. Therefore, these two systems cannot deal with type polymorphism. SFPD addressed these two limitations by performing more comprehensive inter-procedural context-sensitive points-to analysis. In comparison, HookScout performs context-sensitive dynamic binary analysis. In consequence, it is able to eliminate the requirement for source code and handle type polymorphism. HookMap [30] analyzes the kernel-side execution of certain security applications to help identify potential hook sites. Compared to HookMap, HookScout performs more complete analysis by monitoring the full kernel execution and thoroughly tracking function pointers. Moreover, HookScout conducts more advanced context-sensitive type inference, so it can deal with function pointers in complex data structures and achieve significantly higher coverage than HookMap.

## 8 Conclusion

In this paper we targeted a class of advanced kernel attacks: function pointer hooking. We assessed the severity of this new threat. First, we conducted a quantitative measurement study to show the attack surface is vast. Second, we implemented two new keyloggers using this attack technique, showing that this threat is realistic even on closed-source operating systems like Windows. To effectively combat this threat, we presented HookScout, a proactive, context-sensitive hook detection scheme capable of detecting this stealthiest persistent control flow modifications within the Windows kernel without the need for source code. We demonstrated HookScout's ability to generate a context-sensitive policy for detecting persistent control modifications that can be used on any machine with the same version of the OS kernel installed. We evaluated our system against real world stealthy rootkits and malware and showed that we were able to detect all of them (including our synthesized keyloggers). Additionally, we showed that our approach is easily deployable, has a low overheard and most importantly, our approach is generic and capable of detecting kernel-wide function pointer changes.

## References


1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structure invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, California, USA (December 2008)
2. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track (April 2005)
3. Butler, J., Hoglund, G.: VICE–catch 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
4. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2009) (November 2009)
5. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (Security 2004) (August 2004)
6. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2005) (November 2005)
7. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.: Dynamic Spyware Analysis. In: Proceedings of the 2007 Usenix Annual Conference (Usenix 2007) (June 2007)
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of Network and Distributed Systems Security Symposium (NDSS 2003) (February 2003)
9. Hoglund, G.: Kernel object hooking rootkits (KOH rootkits), http://www.rootkit.com/newsthread.php?newsid=501
10. Hultquist, S.: Rootkits: The next big enterprise threat, http://www.infoworld.com/article/07/04/30/18FErootkit_1.html
11. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium (July 2009)
12. IceSword, http://www.antirootkit.com/software/IceSword.htm
13. The IDA Pro Disassembler and Debugger, http://www.datarescue.com/idabase/
14. Lanzi, A., Sharif, M., Lee, W.: K-Tracer: A system for extracting kernel malware behavior. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009) (February 2009)
15. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries. In: Proc. 17th Usenix Security Symposium, San Jose, CA (July 2008)
16. Nick, J., Petroni, L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
17. Offensive computing, http://www.offensivecomputing.net/
18. Payne, B.D., Carbone, M., Sharif, M.I., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, Oakland 2008 (2008)

19. RAIDE, http://www.rootkit.com/vault/petersilberman/RAIDE_BETA_1.zip
20. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: EuroSys 2009 (April 2009)
21. rootkit.com, http://www.rootkit.com/
22. Rustock, C.: http://www.rootkit.com/newsread.php?newsid=879
23. Rutkowska, J.: System virginity verifier: Defining the roadmap for malware detection on windows systems. In: Hack In The Box Security Conference (September 2005), http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt
24. Schreiber, S.B.: Undocumented Windows 2000 Secrets. In: Windows 2000 Object Management, ch. 7 (2007)
25. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007 (2007)
26. Sony's DRM Rootkit: The Real Story, http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html
27. Storm Worm, http://news.zdnet.co.uk/security/0,1000000189,39285565,00.htm
28. TEMU: The BitBlaze dynamic analysis component, http://bitblaze.cs.berkeley.edu/temu.html
29. UAY kernel-mode backdoor, http://www.xfocus.net/tools/200602/uay_source.rar
30. Wang, Z., Jiang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
31. Wang, Z., Jiang, X., Cui, W., Ning, P.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2009) (November 2009)
32. Yin, H., Liang, Z., Song, D.: HookFinder: Identifying and understanding malware hooking behaviors. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (February 2008)
33. Yin, H., Song, D.: Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley (January 2010)
34. Yin, H., Song, D., Manuel, E., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS 2007) (October 2007)