Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data

Sebastian Vogl*, Robert Gawlik†, Behrad Garmany†, Thomas Kittel*,
Jonas Pfoh*, Claudia Eckert*, Thorsten Holz†

*Technische Universität München
†Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract

Generally speaking, malicious code leverages *hooks* within a system to divert the control flow. Without them, an attacker is blind to the events occurring in the system, rendering her unable to perform malicious activities (e.g., hiding of files or capturing of keystrokes). However, while hooks are an integral part of modern attacks, they are at the same time one of their biggest weaknesses: Even the most sophisticated attack can be easily identified if one of its hooks is found. In spite of this fact, hooking mechanisms have remained almost unchanged over the last years and still rely on the *persistent* modification of *code* or *control data* to divert the control flow. As a consequence, hooks represent an abnormality within the system that is *permanently evident* and can in many cases easily be detected as the hook detection mechanisms of recent years amply demonstrated.

In this paper, we propose a novel hooking concept that we refer to as *dynamic hooking*. Instead of modifying persistent control data permanently, this hooking mechanisms targets *transient* control data such as return addresses at run-time. The hook itself will thereby reside within *non-control data* and remains hidden until it is triggered. As a result, there is no *evident* connection between the hook and the actual control flow change, which enables dynamic hooks to successfully evade existing detection mechanisms. To realize this idea, dynamic hooks make use of exploitation techniques to trigger vulnerabilities at run-time. Due to this approach, dynamic hooks cannot only be used to arbitrarily modify the control flow, but can also be applied to conduct *non-control data* attacks, which makes them more powerful than their predecessors. We implemented a prototype that makes uses of static program slicing and symbolic execution to automatically extract paths for dynamic hooks that can then be used by a human expert for their realization. To demonstrate this, we used the output provided by our prototype to implement concrete examples of dynamic hooks for both modern Linux and Windows kernels.

1 Introduction

Over the last decade, the sophistication and technical level of malicious software (*malware*) has increased dramatically. In the early 2000s, we saw malware such as the *I Love You* [29] and *Blaster* worms [3] that generally operated in user space with very little in the way of defensive mechanisms. In contrast, we nowadays see complex kernel level malware such as *Stuxnet*, *Duqu*, and *Flame* [6] that show an increase in sophistication in the target of their attack, the exploitation methods used to deliver them, and their ability to evade detection. By targeting kernel space, modern malware effectively runs at the same privilege level as the operating system (OS), enabling it to attack and modify any part of the system including the kernel itself. In addition, malware can take advantage of stealth techniques that were originally only used by kernel rootkits to hide itself deep within the system. This makes the detection of malware increasingly difficult, especially as malware continues to evolve, enabling it to stay one step ahead of the defenders. Currently, one the most sophisticated methods employed is data-only malware [16, 41].

However, even very sophisticated malware such as data-only malware has an Achilles' heel: in general, malware needs to intercept events within the system to be able to fulfill its purpose [27, 44]. Without this capability, malware would be unable to react to events or provide fundamental functionality such as key logging and file hiding, which would severely limit its possibilities. Event interception, however, requires malware to divert the control flow of the infected system at run-time. To achieve this, malware must install *hooks* in the system that facilitate the required control flow transfer on behalf of the malware whenever the desired event occurs. While malware might manage to hide itself, these hooks represent an abnormality that will be *permanently* visible within the system and thus lends itself well to becoming the basis of detection mechanisms. This insight led to a wide range of research that enable the monitoring of malware hooking behavior for the purpose of signature

generation [47] or detecting malware based on control flow modifications [15, 20, 43].

Although existing systems are not yet able to detect all hooks that are placed by malware, the remaining possibilities for malware to install hooks are constantly dwindling. Hooks that are based on code modifications are usually no longer an option, since changes to code areas can be easily detected due to their static nature. This leaves attackers only with the option of data hooks [43, 47], but even here the options are increasingly restricted by modern detection mechanisms. The reason for this development is that, in contrast to malware where one can observe a constant evolution of techniques and mechanisms used, hooking techniques have not significantly changed over the course of recent years.

In this paper, we present a novel hooking concept that we refer to as *dynamic hooking*. In contrast to existing hooking mechanisms which *persistently* modify control data, our hooking approach targets *transient* control data such as return addresses at run-time. As a consequence, the resulting control flow change that is introduced by the hook only becomes visible when the hook is actually triggered. This significantly complicates the detection of dynamic hooks as security mechanisms can no longer focus on persistent control data, but must also take transient control data into account. What is even more, the hook itself will thereby reside in *non-control* data, which is much more difficult to analyze and verify [4, 15] when compared to *control* data that traditional hooks target. Despite the fact that dynamic hooks reside purely in non-control data, they are able to reliably intercept the execution flow of functions similar to traditional hooks. Furthermore, they can be used in pure data-only attacks, which are by themselves a realistic and dangerous threat [3, 5]. Thus they are not only harder to detect, but also more powerful than their predecessors.

To provide these capabilities, dynamic hooks modify data in such a way that they will trigger vulnerabilities at run-time. Through this approach, they are able to arbitrarily modify the control flow, while the hook itself only consists of the data that triggers and exploits the vulnerability. This makes them quite similar to traditional exploitation techniques with the exception that they target applications that are *already* controlled by the attacker. Due to this fact, the attack surface for dynamic hooks is much broader compared to traditional exploitation, since an attacker can not only attack external functions, but also *internal* functions.

Furthermore, dynamic hooks can be obtained automatically in a manner comparable to automated exploit generation [2]. To demonstrate this, we implemented a prototype that leverages static program slicing [40, 45] and symbolic execution [34] to automatically extract satisfiable, exploitable paths for dynamic hooks. The prototype thereby provides detailed information about each jump condition in the path and the actual vulnerability in an intuitive format, which makes the output suitable for exploit generation frameworks or a human expert. We used this prototype to automatically identify dynamic hooks for recent Linux and Windows kernels. Additionally, we implemented proof of concepts (POCs) of dynamic hooks that demonstrate how they can be used in practice to intercept events such as system calls or to implement backdoors. This proves that the suggested hooking mechanism is not only powerful, but also realistic.

In summary, we make the following contributions:

- We present a novel hooking concept called *dynamic hooking* that targets transient control data at runtime instead of persistent control-data. This approach bypasses existing hook detection techniques proposed in the last few years.
- We show how dynamic hooks for OS kernels can be automatically found by leveraging binary analysis techniques and implemented a prototype.
- We provide detailed POC implementations of dynamic hooks for both Linux and Windows kernels that demonstrate their capabilities and possibilities.

2 Technical Background

Before presenting our approach to realize dynamic hooks, we first review background information that is essential for the understanding of the remainder of the paper. We begin by defining important terms and then discuss why malware in general requires hooks within the system to function. Finally, we cover existing hooking mechanisms and their countermeasures.

2.1 Definitions

We first introduce important terms that we will use throughout the paper. In particular, we highlight the differences between control data and non-control data as well as transient and permanent control data.

Control data and non-control data. *Control data* specifies the target location of a branch instruction. By changing control data, an attacker can arbitrarily change the control flow of an application. Examples of control data are return addresses and function pointers.

In contrast, *non-control data* never contains the target address for a control transfer. In certain cases, however, it may influence the control flow of an application. For instance, a conditional branch may depend on the value of non-control data.

Transient and persistent control data. We consider control data to be *transient* when it cannot be reached through a pointer-chain originating from a global variable. This essentially implies that there is no lasting connection between the application and the control data. Instead, the control data is only visible in the current scope of the execution such as a return address which is only valid as long as a function executes.

By extension, we consider all control data that is reachable through a global variable as *persistent*, since the control data is permanently connected to the application and can thus always be accessed independent of the current scope.

2.2 Malware and Hooking

Petroni et al. [27] estimated that about 96% of all rootkits require *hooks* within the system to function. Intuitively, this makes sense: since the sole purpose of rootkits is to provide stealth, they have to hide all signs of an infection. While existing structures can be hidden using techniques such as *direct kernel object manipulation (DKOM)* [38], hooks enable rootkits to react to changes occurring at *run-time*. Consider, for instance, that a hidden process creates a new network connection or a child process. Naturally, a rootkit must also hide such newly created objects to achieve its goal. This, however, requires a rootkit to be notified of the occurrence of such events. Hooks solve this problem by enabling a rootkit to install callback functions in the system. This makes them an integral part of rootkit functionality.

In practice, rootkit functionality is often mixed with a variety of malicious payloads. According to a report by Microsoft released in 2012 [13], "some of the most prevalent malware families today consistently use rootkit functionality". The primary reason for this is that the single purpose of a rootkit is to avoid detection. Consequently, it is not a big surprise that the techniques formerly only found in rootkits are increasingly being adapted by malware. Since rootkits require hooks to function, this, however, also implies that any malware based on rootkit functionality will require the same.

2.3 Existing Hooking Mechanisms

In general, we distinguish between two different types of hooks: *code* hooks and *data* hooks [11, 43, 47]. Code hooks work by directly patching the application's code regions: wherever the attacker wants to redirect the control flow of the application, she overwrites existing instructions with a branch instruction. As a result, the control flow of the application is diverted every time the execution passes through the modified instructions.

The main problem with code hooks is that code regions are usually static. Therefore, it is generally sufficient to identify modifications to code regions to detect this type of hook. Various techniques have been proposed that leverage such an approach [22, 27, 31]. As a result, adversaries resorted to a different hooking form referred to as data hooks. Instead of modifying code directly, data hooks target *persistent* control data within the application. By modifying control data, the attacker is able to divert every control transfer that makes use of the modified data. For example, the most straightforward method for intercepting the execution of system calls is to modify function pointers within the system call table.

To counter the threat of data hooks, researchers proposed various systems that aim to protect control data within an application [9, 20, 27, 43]. However, the main focus of these systems thereby lies in the protection of function pointers that are allocated on the heap or reside within the data region of the application. This is achieved by ensuring that each function pointer points to a valid function according to its control flow graph (CFG). *Transient* control data on the other side is generally ignored by these approaches or they merely consider the protection of return addresses, which is not the only kind of transient control data. Instead transient function pointer may also exist as we will discuss in Section 5.1.

While researchers acknowledge that malware could potentially also target transient control data to modify the control flow [20, 27, 43], these attacks are usually only considered in the context of exploitation or return-oriented rootkits [16], but are not deemed to be relevant for hooking. The reasoning behind this assumption is that malware generally wants to change the control flow of the target application indefinitely in order to be continuously able to intercept events. Consequently, the malware must *permanently* redirect the control flow and thus target persistent control data as transient control data is, by definition, only used by the system for a limited amount of time. In this paper, we demonstrate that this assumption is false and can be used to circumvent existing defense mechanisms against hooking.

3 Dynamic Hooks

In the following, we introduce our novel hooking concept that we refer to as *dynamic hooking*. For this purpose, we provide an overview of the concept, discuss the vulnerabilities that can be used to implement dynamic hooks, and cover the types of dynamic hooks that exist and their properties. Before doing so, however, we first state the attacker model that we assume throughout this paper.

3.1 Attacker Model & Scope

In the following, we assume that the attacker's goal is to install persistent kernel malware such as a rootkit on the victim's system. For this purpose, we assume that the attacker has the ability to manipulate the kernel's memory arbitrarily either through a vulnerability or the ability to load a kernel module (or driver). To avoid detection, the attacker wishes to hide all the hooks of the malware. That is, we are not concerned with the stealth of the malware itself, but instead solely focus on its hooks. Consequently, we consider all malware detection mechanisms that do not detect malware based on its hooks to be out of scope for the remainder of the paper. Furthermore, we assume the target system leverages common protection mechanisms such as Address Space Layout Randomization (ASLR), stack canaries, and $W \oplus X$.

3.2 High-Level Overview

The main problem with existing hooking mechanisms is that they require the *permanent* change of code or function pointers. Consequently, the desired control flow change of the malware is *permanently* evident within the system [27]. The fundamental idea behind dynamic hooks is to solve this problem by hiding the desired control flow change within *non-control data* such that there is no clear connection between the changes that the malware conducts and the actual control flow change. This is accomplished with the help of *exploitation techniques*.

To exploit a vulnerable application, an attacker makes use of specially crafted input data that—when processed by the application—will eventually trigger a vulnerability. If the vulnerability enables the attacker to overwrite important control structures such as a return address, she will be able to modify and often control the execution flow of the application using techniques such as return-oriented programming (ROP) [35].

With dynamic hooks, we apply the same concepts that are used in traditional exploitation scenarios to hooking. That is, we manipulate the input data of the functions we want to hook in such a way that we will trigger a control flow modifying vulnerability when the data is used. This effectively allows us to overwrite control data (e.g., a return address) at run-time and enables us to control the execution flow of the application similar to a traditional hook. The main difference, however, is that such a dynamic hook will reside somewhere within the data structures of the application unnoticed until its malicious payload is eventually used by the target function.

For this approach to work, we need to identify a control flow modifying vulnerability in every function that we want to hook. At first glance this seems unlikely. However, there is a key difference between the exploitation of traditional vulnerabilities and vulnerabilities that are used to realize dynamic hooks: the attacker *already* controls the application at the time she installs a hook. In a traditional exploit, the attacker's goal is to *gain* control over an application. To achieve this, she needs to find an input to the application that will trigger a vulnerability. That is, the attacker can only control the *external* data which is provided to the application. In the case of a dynamic hook, however, this restriction does not apply. As the attacker controls the application, she is free to access and modify *any* internal data structure of the application. This results in a much stronger attacker model when compared to traditional exploitation.

Finding and exploiting vulnerabilities in such a scenario becomes much easier for several reasons. First, many existing protection mechanisms such as ASLR, stack canaries, or $W \oplus X$ only protect against an *external* attacker, but can be easily circumvented by an attacker that controls the application. Second, the attacker can *prepare* the code (or ROP chain) she wants to execute when the vulnerability is triggered beforehand and does not have to provide it during the exploitation process. This enables the attacker to exploit vulnerabilities for which traditional methods would be difficult due to space constraints of the vulnerability. Third, the attack surface for dynamic hooks is much broader. The attacker cannot only attack functions that handle user input, but can also target internal functions that cannot be influenced by the user. In fact, by manipulating internal data structures, the attacker can create new vulnerabilities that would not occur during normal operation of the application, because the targeted data structures are normally only accessed and modified by the program itself. This may even allow the attacker to circumvent checks and filters within the application as the manipulated data structures may contain values that could never occur during normal operation and may thus not have been expected by the programmer. Finally, to hook a specific event, the hook may be placed anywhere within the control flow of the handling code, it is not restricted to a single function.

Example. To illustrate the concept of dynamic hooks at a concrete example, consider the following code from the `list_del` function in the Linux kernel (version 3.8):

```
1  struct list_head {
2          struct list_head *next;
3          struct list_head *prev;
4  };
5
6  static void list_del(struct list_head *entry)
7  {
8           entry->next->prev = entry->prev;
9           entry->prev->next = entry->next;
10 }
```

This function essentially removes the given entry from its list. If the attacker controls the `next` and the `prev`

field from the entry to be deleted, she essentially can trigger an arbitrary 8-byte write on a 64-bit architecture. In particular, she can write the value of `prev` into the memory address `[next + 8]` (Line 8) and the value of `next` into the memory address `[prev]` (Line 9). To use this code fragment for a dynamic hook, the attacker could, for instance, modify a specific entry within the system and set its `prev` pointer to point to the return address of the `list_del` function and its `next` pointer to point to attacker-controlled code. When the entry is deleted, the `list_del` function will then, while processing the malicious pointers, overwrite its own return address and activate the code of the attacker on its return.

The example code above was selected as the `list_del` function is used throughout the Linux kernel and demonstrates the arguments stated above. In general, this function is not exploitable by an external attacker, as the entries that are used by the function are created by other internal functions within the kernel. While these functions initialize the values of the pointers correctly, an attacker that controls the kernel can modify them arbitrarily, thus creating a new vulnerability. The `list_del` function does not expect the manipulated values and uses them without checks. This enables an attacker to conduct an arbitrary 8-byte write, which is not enough to introduce shellcode into the system, but is sufficient to transfer the control flow to a previously prepared code region. In addition, the attacker is not hindered by any of the protection mechanisms used by the Linux kernel, since she can disable $W \oplus X$ for her code[1], does not need to overwrite the stack canary, and knows the address of her code or can calculate the address of the location of the return address[2]. Finally, since the `list_del` function is invoked by many other functions within the kernel, a dynamic hook within this function is very effective.

3.3 Suitable Vulnerabilities

In principle, any kind of vulnerability can be used to implement a dynamic hook. In this paper, however, we will, for the sake of simplicity, focus on *n-byte* writes, sometimes also referred to as *write-what-where* primitives, such as the one presented in the previous example. Such n-byte writes enable an attacker to modify *n* bytes at an arbitrary memory location. In our example, the attacker controls an *8*-byte write to an arbitrary memory address. In x86-assembly, n-byte writes are essentially a memory `mov` instruction for which the source and the destination operand can be controlled by an attacker. An example of a potential 8-byte write vulnerability in Intel assembly syntax is the following instruction:

```
mov [rax], rbx
```

If the attacker can control the contents of `rax` and `rbx` at the time the instruction is executed, she can misuse it for a dynamic hook. It goes without saying that such instructions appear frequently within software. In the Linux 3.8 kernel binary, for instance, we found more than 103,000 `mov` instructions similar to the one shown above that can potentially be abused for an 8-byte write. This corresponds to about 5% of all instructions (1,976,441) within the tested Linux kernel binary (Linux 64-bit 3.8 kernel). Note that this does not include the approximately 58,000 one, two, or four byte write instructions. Together, this equates to a total of 8% of all instructions that can potentially be used to realize a dynamic hook.

3.4 Types of Dynamic Hooks

Generally speaking, there are two different types of dynamic hooks: *dynamic control hooks* and *dynamic data hooks*. The former target the control flow of the victim application and can be used as an alternative to traditional hooks since they enable an attacker to intercept events within the application. Dynamic data hooks, on the other side, do not target control data, but rather other critical data structures within an application. As an example, consider that an attacker wants to install a backdoor. For this purpose, she places a dynamic hook into a control path that can be triggered from userland such as a specific system call. However, instead of changing control data, this dynamic hook will upon invocation directly overwrite the credentials of a predefined process and elevate its privileges to *root*. Since the task credentials are usually a data value, this can be achieved with a single memory write. Thus, instead of overwriting a return address, the attacker simply sets her hook to overwrite the memory location where the task credentials reside. As pointed out by Chen et al. [10], such non-control data attacks can be quite powerful.

While dynamic data hooks do not modify the control flow directly, they can be used to influence the control flow at a later point in time. Consider for instance data that resides in memory and is processed by a just-in-time compiler. If an attacker manages to overwrite this data with dynamic hooks before it gets compiled, she can influence the instructions that are introduced into the system, which can lead to arbitrary code execution [7].

[1] Note that this is necessary since the first write (Line 8) of the example will write the return address (`prev`) into the code (`[next + 8]`) of the attacker. However, this is not a problem in practice, as the attacker can set her code to be writable and executable. In fact, this is even the default for memory allocated in the Linux kernel via `kmalloc`.

[2] The location of the return address depends solely on the address of the kernel stack and the size of the current function's stack frame. Both values are known to the attacker as we will describe in Section 4.2.

3.5 Properties of Dynamic Hooks

Components. Dynamic hooks always consist of two integral components. On the one hand, there is the instruction that activates the hook, which we refer to as the *trigger*. In the case of an 8-byte write, the trigger is the `mov` instruction that conducts the write on behalf of the attacker. Every path that leads to the execution of the trigger is referred to as a *trigger path*. On the other hand, there is the data that was manipulated by the attacker and encodes the malicious action that the attacker wants to conduct. This is the *payload* of the hook. For n-byte writes, the payload usually consists of two manipulated pointers: the *destination pointer*, which contains the address that will be written to and the *source pointer*, which specifies the value that will be written.

Binding. While the same trigger can be shared among different dynamic hooks, each hook in general requires its own payload. The reason for this is that the payload contains the actual data that specifies the control transfer. This data, however, will only be valid in a particular *context*. To overwrite a specific return address, for example, we must first be able to predict its exact location. This requires us to know the exact path leading to the use of the payload by the trigger. In practice, this means that a payload and thus the dynamic hook is usually closely *bound* to a specific execution path. The closer the connection between an execution path and a dynamic hook, the better the control of the attacker over the hook.

In an ideal situation, a dynamic hook is *exclusively* bound to a specific execution path. In this case, the payload of the hook is *only* processed in the execution path that leads to its trigger. This enables the attacker to predict possible modifications applied to the payload before its use in addition to the state of the machine at the time of the exploitation with high probability, since she must only consider a single execution path. Similar to traditional exploits, this is essential information that is required to be able to setup a dynamic hook correctly. After all, the attacker needs to correctly predict the exact address of the control data, which should be overwritten and overwrite it with the precise address of the target code region. Without knowing the exact layout of the stack as well as the transformations that may be applied to the payload before its use, this is a hard task.

If there are multiple paths that use the payload, the dynamic hook is only *loosely* bound to the path leading to the trigger instruction. The more execution paths the payload affects, the more difficult it will become for an attacker to control the hook. On the one hand, this is due to the fact that it will become increasingly difficult to predict the necessary memory addresses and transformations as has been described above. On the other hand, the more functions access the actual payload that the attacker modified, the more likely it will be that the hook introduces side effects into the application that may lead to unexpected behavior and crash the application. Consider, for instance, that an entry that is used by the `list_del` function has been modified to act as payload for a dynamic hook. If the same entry is used by a different function to iterate through all elements within the list, this will most likely lead to a crash of the system as the `prev` and the `next` pointer do not point to the previous and next element, respectively, as would have been expected.

Coverage. Another important property of a dynamic hook is coverage: as dynamic hooks should be closely bound to the execution path containing the trigger, it is essential that this triggering path is *always* executed when the target event that should be hooked is invoked. In this case, the dynamic hook provides *full* coverage. Otherwise, the hook may only be able to intercept *some* execution paths of the target event, but not all. In that case, the hook has only *partial* coverage and must thus be combined with other dynamic hooks to be able to achieve *full* coverage of the target function. Note that while binding is a property of the payload of the hook, coverage is a property of the trigger instruction.

3.6 Automated Path Extraction

So far we have discussed the concept of dynamic hooks and provided an overview of the different types of dynamic hooks and their properties. However, the creation of a dynamic hook still remains a manual process, which can—as in the case of traditional exploitation—be very time-consuming and error-prone especially for complex binaries such as modern OS kernels. We now describe how paths for dynamic hooks can be obtained automatically for a given binary. This is essentially a two-step process: In the first step, we make use of static program slicing [40, 45] to extract potential paths that could be used for a dynamic hook. In the second step, we then employ symbolic execution [17, 34] to verify the satisfiability of the paths and to generate detailed information for their exploitation.

3.6.1 Program Slicing

To find possible locations for dynamic hooks within an application, an attacker has to find triggers that make use of a payload that she can control. Since trigger instructions can be as simple as a memory move, there usually exist many triggering instructions in many paths of the application. To identify whether a particular trigger instruction can be used for a dynamic hook, it is necessary

to analyze the data flow that leads to the particular instruction. One technique that can be used for this purpose is static program slicing [40, 45].

The basic idea behind static program slicing is to traverse back through the control flow graph (CFG) of an application starting from a *sink* node and to extract each node that directly or indirectly influences the values used at the sink. Applied to the problem of finding dynamic hooks, static program slicing thus allows us to determine where the values of the *source* and the *destination* pointer in an n-byte write originate. To achieve this, we first identify all potentially vulnerable `mov` instructions within a given binary. These are essentially all `mov` instructions which move a value contained within a register to a memory location specified by another register. In the next step, we then traverse the CFG of the binary backwards at the assembler level until we encounter the first instruction that modifies the source register of the move. We record this instruction and continue with our backward traversal. Instead of looking for instructions that modify the source register of the original move, however, we will from here on search for instructions that modify the source register of the last instruction we recorded. If we continue this process, we eventually obtain the register or memory location where the value that is later on contained within the source register originates. We then repeat the process for the destination register. All the instructions that we recorded using this method form a *slice* of the binary. Each slice contains all the instructions that affect a given vulnerable `mov` instruction.

We implemented a *slicer* which is capable of extracting potential paths that could be used for n-byte writes from a 64-bit Linux or Windows kernel binary. The implementation of the slicer is based on the disassembler *IDA Pro* [14]. In particular, we make use of the CFG that IDA provides to perform the above described *static interprocedural def-use analysis*. Starting from each trigger, we perform a breadth-first search in a backwards direction. We hereby make use of a register set to conduct the actual analysis. Initially, this register set consists of the source and destination register. Whenever we encounter an instruction that modifies a register included within the register set, we add the source register of the instruction to the set and remove the modified register. Since we walk backwards through the instruction stream, this effectively allows us to record and track the *def-use chains* for the source and destination register. In addition, we record all instructions that we visit along the way, in order to be able to reconstruct the path that we explored in case we consider it to be potentially exploitable.

The challenge that remains to be solved is to determine whether a slice can be used for a dynamic hook or not. To address this problem, we must know whether the registers in the vulnerable move can be controlled by an attacker. We consider this to be the case if the values of the source and destination register originate from a *global* variable. The reasoning behind this approach is that the data used within the move in this case stems from a persistent location. Consequently, to control the final `mov` instruction, an attacker can modify the pointer chain starting from the global variable.

To identify global variables in the kernel, we assume that each access to a fixed address or the *Global Segment register (GS)* constitutes an access to a global variable. The reason for the latter is that both the Linux and Windows kernels store important global variables that are valid for a particular CPU within a memory region pointed to by this register. For instance, both Linux and Windows store the address of the `task_struct` (`gs:0xc740`) or the `_ETHREAD` (`gs:188`) structure of the process that is currently executing in this memory region.

If both the source *and* the destination register originate from a fixed address or the memory region pointed to by GS, we consider the path to be potentially exploitable and record it such that it can later on be used as input for the symbolic execution engine.

3.6.2 Symbolic Execution

Symbolic Execution is a well-known program analysis method that has been proposed over three decades ago [8, 17]. The basic idea of symbolic execution is to treat input data of interest as symbols rather than concrete values. These symbols can represent any possible value and as we walk over the code of a program, the values become constrained. Branches, for instance, set up conditions that constrain symbolic variables. Each of these conditions can be represented as a logical formula which can then be fed into an SMT solver to obtain concrete values that satisfy the path conditions. A profound introduction is available in the literature [25, 34].

We use forward symbolic execution to verify the satisfiability of our sliced paths and to produce detailed information for the creation of the dynamic hooks. In the process, we utilize the VEX IR, which is a RISC like intermediate representation with single static assignment (SSA) properties, deeply connected to the popular Valgrind toolkit [24]. Due to space limitations, we refrain from discussing this intermediate language in detail.

To verify satisfiability, we transform each basic block of the sliced path into VEX IR code and execute the code symbolically. The translation to VEX IR is achieved by utilizing a python framework called *pyvex* [36]. We dismantle every VEX statement that we obtain from *pyvex* and link the components of the statements into our own data structures. These data structures are used to walk over the VEX code and by doing so, we semantically map the statements to Z3 expressions. Z3 is a theorem

prover developed at Microsoft Research that we use to solve our formulas [23].

As we walk over the VEX code of our sliced paths, we also keep track of three global contexts, i.e., a memory context, CPU context, and the current jump condition. Each context consists of Z3 expressions that semantically mirror the current state of the execution. Additionally, each basic block also keeps track of temporary VEX IR variables in SSA form. By constant propagation, we use these variables to resolve source and destination. Each store, load, and register set statement updates the corresponding context in form of Z3 expressions. Once we hit a jump condition, we ask the solver whether we can take the jump according to our context. If no solution exists, we can filter out the path. An unsatisfiable set of formulas stops execution of the current path, and we move on with the next slice.

At this point it is worth mentioning that we do not use symbolic execution in the traditional sense to achieve code coverage. Our main goal is to check whether we can walk down our paths and to determine what value sets lead us to the end of the slice. We use the symbolic formulas to generate detailed information about the controlled registers at the time the vulnerability is triggered as well as the jump conditions that must be fulfilled to actually reach the trigger. By processing over the VEX code, the solver also gives us possible values to set.

4 Experiments

Based on the slicer and the symbolic execution engine, we created a prototype that we used to automatically extract paths for dynamic hooks in a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. We chose this approach for three main reasons. First and foremost, since malware nowadays generally attacks the kernel, this approach allowed us to test the prototype in a realistic scenario. Second, kernel binaries are especially complex, which makes them well suited for a thorough test of our implementation. Finally, by targeting Windows and Linux, the experiments show that the proposed mechanism is applicable to two of the most popular OSs.

In the following, we first discuss the results that we obtained by providing detailed statistics about the automatically extracted paths for both kernels. To demonstrate how useful the prototype is when it comes to the actual creation of the hooks, we also describe three concrete POCs for dynamic hooks that we created based on the information that the prototype provided.

4.1 Automated Path Extraction

As stated above, we tested our prototype with a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. The goal of the experiment was to automatically extract trigger paths that could then either be used by a human expert to manually design dynamic hooks or to automatically generate exploits. Table 1 provides an overview of the obtained results.

At first, we determined the number of instructions contained within both kernel binaries for reference. In the next step, we obtained the number of potentially exploitable 8-byte `mov` instructions. In the process, we only counted those `mov` instructions that move data from one general purpose register into a memory location specified by another general purpose register with the condition that the involved registers were neither `rbp` nor `rsp`. The reason for this restriction is that our prototype implementation currently does not support a memory model, meaning that we cannot track memory store and load operations in our slicer, which is why we currently ignore any path that requires this functionality. We will cover this limitation in more detail in Section 5.3. As Table 1 shows, about 2 % of all instructions within the tested kernels are `mov` instructions that fulfill this criteria.

Next, we used the slicer to extract potentially exploitable slices for each of the identified moves. In case of Linux, the slicer considered about 4% of the `mov` instructions as potentially exploitable, while on the Windows side about 20% of the `mov` instructions were marked as possibly exploitable. We assume that the significant difference between Windows and Linux stems from the fact that Linux has substantially more `mov` instructions that store or load data from memory (61,651 vs 37,272). Since the slicer does not support a memory model, it will abort whenever such a `mov` instruction is part of a def-use chain. Due to their number, this scenario is more likely to occur on Linux than on Windows.

Finally, we symbolically executed each of the obtained slices. In total, this led to 566 exploitable paths for Linux and 379 exploitable paths for Windows. The symbolic execution engine thereby produced the required value for each conditional jump within the path and detailed information of the vulnerable `mov` instruction. In particular, the output[3] specifies exactly which memory addresses must be modified in what way to pass the conditional jumps and where the source and destination values are located, respectively. This information can directly be applied to generate exploits or to manually create a dynamic hook as we will show in the next section.

4.2 Prototypes

We now present three concrete examples of dynamic hooks to illustrate the capabilities and properties which have been discussed throughout the paper. We created

[3] An example of the output is shown in Section 4.2.3.

OS	*Size*	*Instructions*	*8-byte moves*	*Slices*	*Paths*
Linux 3.8 64-bit (`vmlinux`)	18.8 MB	1,976,441	42,130 (2.1%)	1753 (4%)	**566 (32%)**
Windows 7 SP1 64-bit (`ntoskrnl.exe`)	5.3 MB	1,330,791	26,694 (2.0%)	5450 (20%)	**379 (07%)**

Table 1: Overview of the 8-byte moves, the potentially exploitable slices, and the exploitable paths according to the symbolic execution engine for the analyzed Linux and Windows kernels.

these examples based on the output provided by our prototype. The first and the third example focus on a dynamic control hook, while the second example demonstrates a dynamic data hook. To ease the understanding of the examples, all hooks leverage a trigger instruction within the `list_del` function (as explained in Section 3.2) or its Windows equivalent. The first two hooks were implemented for Linux 3.8 and an Intel Core i7-2600 3.4 GHz CPU. To demonstrate that the proposed concept is similarly applicable to Windows, the third hook was implemented on a fully patched version of Windows 7 SP1 running on the same CPU.

4.2.1 Dynamic Control Hook: Intercepting Syscalls

A common functionality that kernel level malware requires is the possibility to intercept system calls. In this example, we show how a single dynamic hook can be used to intercept *all* system calls for a particular process. To achieve this, the hook is placed into the execution flow of the *system call handler*, which is—independent of the system call mechanism that is used (i.e., interrupt-based, sysenter-based, or syscall-based)—invoked whenever a system call on the x86 architecture is executed. The main purpose of the syscall handler is to invoke the actual system call by using the system call number as an index into the system call table.

Similar to other functions within the kernel, the system call handler can be audited for debugging reasons. Auditing can be enabled or disabled within the flags field of the `thread_info` struct associated with each process. By setting the `TIF_SYSCALL_AUDIT` flag, every system call conducted by a process will also lead to the invocation of the auditing functions. In particular, the function `__audit_syscall_entry` will be executed before the invocation of a system call and the function `__audit_syscall_exit` will be executed after the system call, but before the system call handler hands control back to user space. In our POC, the dynamic hook is set within the `__audit_syscall_exit` function.

When syscall auditing is enabled, the `__audit_syscall_entry` function records information about the system call such as the syscall number and the arguments of the syscall within the audit context of the process. The purpose of the `__audit_syscall_exit` function is to reset the audit context of the task before the system call returns. In the process of resetting the audit context, this function invokes the inline function `audit_free_names`, which resets the `names_list` within the audit context:

```
1 static inline void audit_free_names(
2         struct audit_context *context) {
3     ...
4     list_for_each_entry_safe(n, next,
5           &context->names_list, list) {
6             list_del(&n->list);
7     }
8     ...
9 }
```

The `audit_free_names` function essentially iterates over the `names_list` of the audit context (Line 4) and deletes every entry within the list (Line 6). Consequently, if we control the `names_list`, we can control the entry that is passed to the `list_del` function, which in turn allows us to exploit its vulnerability. As the `names_list` is not modified by the `__audit_syscall_entry` function or anywhere else in the kernel[4], the attacker is free to modify it in any way she wants. That is, the `names_list` structure is exclusively bound to the execution path within the syscall handler that we use for our dynamic hook.

While the `names_list` structure seems to be perfectly suited for a dynamic hook, the triggering path places additional constraints on the hook. The problem arises due to the fact that the `list_del` function is contained within a loop that iterates over all entries within the `names_list` list (Line 4). To iterate through the list, the loop will essentially follow the `next` pointer in every entry until one of them points back to the first element in the list, which is `&context->names_list`. Since we want to modify the `next` and the `prev` pointer of an entry within the list to conduct an arbitrary 8-byte write, we have to take this problem into account and assure that the list iteration will eventually terminate. To achieve this we initialize the audit context as shown in Figure 1.

The basic idea behind this setup is to make use of a special address, referred to as a "magic address", that is a valid memory address, but at the same time contains valid x86 instructions. Due to little-endian byte order, these valid instructions must be contained in re-

[4]While there are other functions in the kernel that try to access the `names_list`, these attempts can be blocked by setting the first member within the audit context (`dummy`) to one.

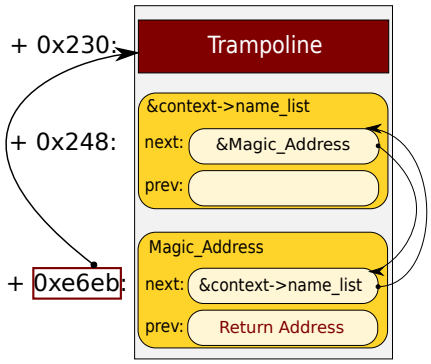

Figure 1: The audit context structure that the attacker uses to set a dynamic hook within `audit_free_names`.

verse order within the address. In Figure 1, the instruction encoded into the address is a negative relative jump (`0xe6eb` (address) $\Rightarrow$ `0xebe6` (instruction)) that will upon execution transfer control to a trampoline, that then transfers control to an arbitrary address. Initially when the loop begins iterating over the `names_list`, it follows the `next` pointer to the first entry within the list, which is located at the magic address. The `next` pointer stored at the magic address will in turn point back to the `names_list`, thus fulfilling the loop condition. However, before the loop exits, the first entry in the list (located at the magic address) is processed by the `list_del` function. Since the `prev` pointer of this entry points to the location of a return address, the `list_del` function will overwrite this return address with the value stored in the `next` pointer ($prev \rightarrow next = next$), which points to `&context->names_list`. Consequently, as soon as the return address is used, control will be transferred to the address of `&context->names_list` where the magic address is stored, leading to the execution of the magic address and the activation of the trampoline code. Note that the hook requires the audit context region created by the attacker to be writable and executable, since the `list_del` function conducts *two* write operations as has been described in Section 3.2. This is not a big problem in practice, since every memory region allocated with `kmalloc` is by default writable and executable.

The final problem that remains is which return address we are actually going to overwrite and how we can predict its location. As previously stated, the syscall handler is invoked before every system call and it will invoke the actual function handling the syscall. Thus, if we know the stack frame size of the syscall handler and the location of the kernel stack, we can predict where the return address of the function that is invoked by the syscall handler resides. The stack frame size can be obtained from the assembler code of the syscall handler, while the location of kernel stack can be obtained from a kernel variable (`get_cpu_var(kernel_stack)`). The target return address will then reside at `stack_frame_size + get_cpu_var(kernel_stack)`.

Summary. A dynamic control hook for intercepting all system calls for a particular process can be placed in the `audit_free_names` function. To ensure that execution passes through this function, we set the `TIF_SYSCALL_AUDIT` flag within the `thread_info` struct of the target process. In the next step, we modify the audit context of the target process in the way described above and use a trampoline to control the execution flow. This enables us to reliably divert the control flow at run-time. The resulting dynamic hook will have full coverage and be exclusively bound to the execution path leading to the `audit_free_names` function.

4.2.2 Dynamic Data Hook: Installing a Backdoor

In the second example, we demonstrate the possibilities of dynamic data hooks. In particular, we show how a dynamic data hook can be used to install a backdoor within a Linux system that is capable of elevating the task rights of a predefined process to *root*. For this purpose, we leverage the `ptrace` system call, which enables one process to attach to another process for debugging reasons. To install the backdoor, we simulate that a process used the `ptrace_attach` system call to attach to the target process (i.e. the process that will contain the hook). This is achieved by manually applying the changes that the `ptrace_attach` function conducts to the internal data structures of the target process. Most importantly, the `state` field of the task must be updated to include `__TASK_TRACED`, the `ptrace` field within the task must be set to 1, and the `parent` field must be set to the process which will later trigger the backdoor. We will defer the discussion of this last change for the moment and explain it in more detail later on.

Once the changes of the `ptrace_attach` function have been simulated, it is possible to invoke the `ptrace_detach` function on the so prepared process. The execution of this function eventually leads to the invocation of the `__ptrace_unlink` function, which in turn invokes the `list_del` function using the `ptrace_entry` pointer within the target process as argument:

```
1 void __ptrace_unlink(
2       struct task_struct *child) {
3       ...
4       list_del(&child->ptrace_entry);
5       ...
6 }
```

To use this code fragment for a dynamic data hook, we modify the $ptrace\_entry \rightarrow next$ pointer and the $ptrace\_entry \rightarrow prev$ pointer of the target process. This

enables us to conduct an arbitrary 8-byte write when the `list_del` function is invoked during the execution of `ptrace_detach`. In particular, we set the `prev` pointer to point to the task credentials that we want to override and the `next` pointer to an address that is writable and ends with four zero bytes. To understand this, we have to take a look at the Linux task credential structure, which defines the access rights of a process:

```
1  struct cred {
2      ...
3      kuid_t  uid;  /* real UID */
4      kgid_t  gid;  /* real GID */
5      kuid_t  suid; /* saved UID */
6      kgid_t  sgid; /* saved GID */
7      kuid_t  euid; /* effective UID */
8      kgid_t  egid; /* effective GID */
9      ...
10 };
```

Each task contains three pairs of access rights and each access right pair consists of a user id and a group id. Most important for us is the effective user id (`euid`), which specifies the effective access rights of a process. Since the root user in Linux generally has the user id zero, our goal is to overwrite the `euid` field, which has a size of 4 bytes, with zeroes. If we choose an address for the `next` pointer that has its lower 32-bits set to zero and additionally set the `prev` pointer to point to the `euid` field of the process whose privileges we want to elevate, we will—due to the little endian byte order—overwrite the `euid` ($prev \rightarrow next = next$) field with zeroes and thus set the access rights of the process to root. However, because the `list_del` function will also write the `prev` pointer into the address of `[next + 8]` ($next \rightarrow prev = prev$), we have to ensure that the address used within the `next` pointer points to a writable memory region that does not contain crucial data. A possible address that can be used for this purpose is `0xffff880000000000` since this address usually points to the first 8-bytes of the physical memory of the machine, which is not used by the Linux kernel. Finally, note that we will also override the `egid` of the process with the upper 32-bits of the address in the next pointer. This will, however, not affect the process as long as it has a valid `euid`.

We can now set up a dynamic hook as follows: First, we need to select a target process that remains running on the system as it will contain the above described dynamic hook. Good candidates are therefore background daemons such as the SSH daemon. Second, we need to specify the victim process whose privileges we want to elevate and setup the dynamic hook within the target process. Since we need to know the address of the task struct of the victim process in order to be able to set the `prev` pointer to its `euid` field, this process also needs to remain running. A good choice in this case could, for instance, be a shell process within a screen session.

To activate the backdoor, we need to call the `ptrace` syscall with the `PTRACE_DETACH` argument on the target process. However, the backdoor cannot be activated by any process because only the tracing process can detach from the traced process. Since we simulate the changes conducted by `ptrace_attach`, the process which can execute the `ptrace_detach` call, is the process that we specify as `parent` during the setup of the dynamic hook. While this ensures that the backdoor cannot be triggered by accident, this requires us to specify the process that triggers the backdoor when we setup the dynamic hook. The easiest way to solve this problem is to specify the victim process as parent of the target process. In this case the victim, whose privileges will be elevated, can trigger the backdoor itself.

Summary. A dynamic data hook can be used to implement a backdoor that can be triggered from user space with arbitrary access rights. In our example, the backdoor is closely bound to the process that was specified as the tracing process and to the execution path within `ptrace_detach`. In addition, the hook only provides partial coverage as only the detach call to a specific process will trigger it, which is desired behavior in the case of a backdoor.

4.2.3 Dynamic Control Hook: Process Termination

To show that the proposed hooking concept can be applied to other OSs as well, we will in our final example present a dynamic control hook that we implemented on a fully patched version of Windows 7. In particular, the hook is capable of intercepting the termination of an arbitrary process, which can, for instance, be useful in situations where a malicious process on the system is found and terminated by a security application or the user. Due to the hook, the malware would be notified of this event and could react to it.

When a process is exiting on Windows 7, the function `NtTerminateProcess` is invoked which in turn invokes various cleanup functions that prepare the termination of the process. One of these functions is `ExCleanTimerResolutionRequest`. To support a wide range of applications, Windows provides processes with the possibility to request a change to the system's clock interval [32]. This enables programs that have a demand for a faster response time to decrease the clock interval and thus to increase the number of clock-based interrupts. When a process emits such a request, the process is added to the `TimerResolutionLink` list, which is used by the OS to manage all timer resolution changes. As the name suggests, the purpose of the `ExCleanTimerResolutionRequest` function is to remove processes from the management list once they exit. Our automated path extraction tool discovered the following path within this func-

tion:

```
1  ----SLICE----
2  0x000000014042c396   mov     rax, gs:188h
3  0x000000014042c39f   mov     rbx, [rax+70h]
4  0x000000014042c3c6   mov     rcx, [rbx+4A8h]
5  0x000000014042c3cd   mov     rax, [rbx+4B0h]
6  0x000000014042c3d4   mov     [rax], rcx
7  0x000000014042c3d7   mov     [rcx+8], rax
8
9  ----SYMBOLIC----
10 Jump Condition in: BB_0x14042c390
11 Concat(0x0, Extract(0x1f, 0x0,
12 MEM[RBX+0x440])) >> Concat(0x0, 0xc) &1 == 0
13
14 CPU CONTEXT/CONTROLLED REGISTERS
15 RCX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4a8]
16 RAX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4b0]
```

To remove a process from the `TimerResolutionLink` list, the `ExCleanTimerResolutionRequest` function obtains the forward and the backward pointer (Line 4 and Line 5) from the `EPROCESS` structure of the process and performs the discussed list delete operation (Line 6 and Line 7). The only prerequisite for this path is that the 13th least significant bit of the memory word at location `EPROCESS+0x440` is not set (Line 11). By manipulating this memory word and the pointers, which are located within in the `EPROCESS` struct of the process at offset 0x4A8 (Line 4) and offset 0x4B0 (line 5) respectively, we can thus perform an arbitrary 8-byte write and change the control flow. In our POC we set the forward pointer (`rcx`) to point to our shellcode and the backward pointer (`rax`) to point to the return address of `ExCleanTimerResolutionRequest`. Just as in the case of our first example, the location of the latter can be obtained by subtracting the sum of the stack frames of the invoking functions from the start address of the kernel stack, which is stored within the `InitalStack` variable contained within the `KTHREAD` structure of the thread of the process. Similarly, the area where the shellcode resides must be writable *and* executable. On Windows, we can allocate such a memory region by invoking the `ExAllocatePoolWithTag` function with the argument `NonPagedPoolExecute`.

One last problem that remains, however, is that the `TimerResolutionLink` entry structure of a process is unfortunately not exclusively bound to the path of our dynamic hook, since the `TimerResolutionLink` list is also used by other functions such as `ExpUpdateTimerResolution`. The solution to this problem is quite simple, though: since the `TimerResolutionLink` list is not critical for the execution of a process and the `ExCleanTimerResolutionRequest` function does on top of that not iterate through the list, but rather accesses the forward and backward pointers directly, we can simply remove the entry from the linked list. As a result, the manipulated entry will no longer be processed by other management functions, which will bind the `TimerResolutionLink` entry structure exclusively to our trigger path. In our experiments, removing processes from the `TimerResolutionLink` list did not affect their execution in any way. The proposed dynamic hook therefore serves as an example that an exclusive binding of a hook payload must not be given by the target application, but can also be manually enforced by the creator of the hook.

Summary. By manipulating the `TimerResolutionLink` entry structure of a process in the way described above we can install a dynamic hook and intercept the termination of an arbitrary process on Windows. While the manipulated structure is by default not exclusively bound to the trigger path, the creator of the hook can enforce an exclusive binding manually by removing the manipulated entry from its linked list. In addition, the presented dynamic hook had full coverage in our experiments. It was even triggered if we forcefully terminated the process using the task manager.

5 Discussion

Up to this point, we have not discussed what kinds of transient control data exist. This is why it may seem to the reader that dynamic control hooks could be mitigated by protecting return addresses alone. In this section, we cover this topic in more detail and show that this is not the case. In addition, we cover possible countermeasures against dynamic hooks and review the limitations of the proposed hooking concept and our current prototype.

5.1 Transient Control Data

Instead of targeting *persistent* control data such as function pointers in the system call table, dynamic control hooks change *transient* control data at run-time. While return addresses are a popular example of *transient* control data, it is not the only kind of transient control data that exists. For instance, if a function allocates a local function pointer, this pointer will reside on the stack and not in the data segment or the heap. Instead of overwriting the return address, an attacker can in such a case similarly target the function pointer. While this is a rather unlikely scenario, it demonstrates a very important class of attacks where a local variable on the stack is changed to achieve the desired control flow change. This class of attacks is not restricted to function pointers alone. Consider, for example, the following code from the `read` system call in the Linux kernel[5]:

[5]For better readability we directly included the `vfs_read` function into the `read` system call. In the actual code the function call in Line 10 will occur in the `vfs_read` function.

```
1  struct fd {
2      struct file *file;
3      int need_put;
4  };
5
6  SYSCALL_DEFINE3(read, unsigned int, fd, char
7                  __user *, buf, size_t, count) {
8      struct fd f = fdget(fd);
9      ...
10     ret = f.file->f_op->read(f.file, buf,
11                              count, pos);
12     ...
13 }
```

In this case, a local structure (`struct fd f`) is allocated on the stack (Line 8). The structure contains a pointer to another structure (`struct file *file`), which in-turn contains a function pointer that is called in Line 10. With the help of a dynamic hook, an attacker could modify the pointer within the local structure (Line 2) and point it to an attacker-controlled structure instead. If she manages this before the function call in Line 10 is executed, this will effectively allow her to control the function call and thus enable her to arbitrarily change the control flow.

Instead of targeting a return address or a function pointer directly, the attacker in this scenario modifies a local pointer on the stack. This approach enables her to control any data that the local function accesses using this pointer. In the kernel, where objects in general are accessed through pointer chains, this represents a powerful attack vector, which effectively provides control over *any* object that the pointer references. Since similar code exists in many other functions within the kernel, this attack vector must be taken into account when one considers countermeasures against dynamic hooks.

5.2 Countermeasures

Dynamic hooks are installed by an attacker that already controls the application, which renders many of the existing defense mechanisms against exploits ineffective. However, while dynamic hooks are a powerful attack vector, there are, of course, countermeasures that can be used to reduce the attack surface. In the following, we first discuss possible countermeasures against dynamic control hooks, before we present defense mechanisms for dynamic data hooks.

Dynamic control hooks. What makes dynamic control hooks difficult to detect is that they do not permanently modify control data. Instead, their payload is hidden within non-control data and the actual control flow modification only occurs at run-time. This enables them to evade popular hook detection mechanisms such as HookSafe [43] or SBCFI [27], which only protect *persistent* control data, but ignore *transient* control data on the stack. However, at some point during the execution, dynamic control hooks must override control data in order to divert the control flow. Thus while a dynamic hook may be hidden at first, it will become visible when it is triggered. The resulting control flow change can potentially be detected using control flow integrity (CFI) and related approaches [1, 15, 20, 39, 42, 46, 48].

In order to detect dynamic control hooks with CFI, it is crucial that *every* control transfer of an application is verified. If a single control transfer is missed, this can potentially be abused by an attacker to install a dynamic hook. However, finding all possible control transfer instructions within complex software such as an OS kernel is a difficult problem. This is especially true if we consider attacks on transient control data such as the one present in the last section. Even worse, control transfer instructions can often have more than a single target. Consequently, one must not only identify all control transfer locations, but also all the possible targets of these transfers to avoid false positives. Additionally, if there are multiple possible targets for a given control transfer instruction, an attacker can still launch return-to-libc like attacks [37], which is a general problem of CFI mechanisms [48]. Finally, every check of a control transfer comes at a cost [39]: the more instructions we verify, the higher the overhead will be and for applications that are optimized for performance such as an OS kernel, even a small overhead can have a huge impact.

While current CFI approaches are not yet able to solve all of these problems, they certainly reduce the attack surface and make it more difficult to install dynamic control hooks. The results presented in this paper can help to further improve these mechanisms by using the discussed techniques to automatically extract possible triggers from a given application and adding additional checks to verify them. To increase the performance, one could make use of lazy control flow verification as proposed by Bletsch et al. [7]. This could result in an effective and efficient detection system, which might not be able to eliminate dynamic control hooks entirely, but will significantly raise the bar for an attacker.

Dynamic data hooks. The defense mechanisms discussed above make use of the fact that dynamic control hooks have to eventually modify the control flow. That is, the detection mechanisms do not focus on the hook itself, but rather target the *effects* of the hook's invocation. The idea behind dynamic data hooks is to complicate detection even further by modifying non-control data instead of control data. As a result, defenders can no longer concentrate on the control flow of the application alone, but rather have to detect integrity violations within the data of the application.

Verifying the integrity of data structures is a difficult

problem. Petroni et al. [26] were the first to propose a general architecture for the detection of kernel data integrity violations. Since then various systems have been proposed that try to detect or prevent malicious modification of kernel data structures [9, 15, 19, 30, 33]. What is common to all these approaches, however, is that they only *enforce* integrity checks, but leave the *creation* of the actual integrity constraints to a human expert. To the best of our knowledge, the only approach that tries to generate integrity constraints for kernel data structures automatically is Gibraltar [4]. While this approach provides a good starting point and could support a human expert in the creation of integrity constraints, the authors acknowledge that the generated invariants are "neither sound nor complete" [4]. Creating reliable and evasion-resistant integrity constraints is, however, the basis for the detection of dynamic data hooks.

To be able to effectively protect kernel data structures, additional research in the field of automatic integrity constraint generation is required. Techniques that are able to generate signatures for kernel data structures such as the ones presented by Lin et al. [21] or Dolan-Gavitt et al. [12], could thereby provide a good starting for further research, as the generated signatures could potentially be used to infer integrity invariants. In the meanwhile, initial defense mechanism could use systems such as HookMap [43] or K-Tracer [18] in combination with the techniques presented in this paper to generate integrity constraints for known dynamic hooks that can then be enforced by one of the systems mentioned above.

Summary. While the threat of dynamic control hooks can potentially be reduced with the help of (kernel-level) CFI mechanisms, dynamic data hooks pose a difficult problem that cannot be easily solved. To detect dynamic data hooks, reliable integrity constraints are required that allow the automatic verification of the kernel data regions. Until these constraints are available, one could reduce the attack surface with the help of manual integrity specifications or by automatically creating integrity constraints for known attacks.

5.3 Limitations

Dynamic Hooks. Dynamic hooks essentially face two limitations. First and foremost, not every function may contain a vulnerability that can be used to implement a dynamic hook. In contrast, it is likely that there are functions which are immune against the attack. However, this is not a big problem in practice: if a particular function cannot be hooked directly, it may still be possible to intercept calls to the function by hooking a function that immediately precedes or follows the function in the execution flow. After all, not every function contains a function pointer either. Function pointer hooks have nevertheless been proven to be very effective in practice.

Second, similar to traditional exploits, a dynamic hook may face restrictions that are caused by the vulnerability it is exploiting. For instance, specific hooks such as the one presented in our first prototype (see Section 4.2.1) may require that certain memory areas are writable and executable. Depending on its restrictions, a dynamic hook may therefore not be suitable for every scenario. This, however, heavily depends on the particular hook.

Automated Path Extraction. While our prototype already produces very valuable paths that can be used to implement powerful dynamic hooks as we have shown in Section 4.2, it also faces some limitations. First, our slicer does not yet support a detailed memory model. As a result, we are unable to find dynamic hooks on paths where registers, which are currently monitored, are loaded with values from the stack. This situation frequently occurs when subfunctions are called. In this case, the calling function often stores register values temporarily on the stack to guarantee that they are not overwritten by the subfunction. During our experiments, the slicer ignored 79,853 such paths due to this restriction.

Second, the symbolic execution engine currently only handles a subset of the available x86 instructions. Most importantly, it is unable to handle some instructions that are a ring-0 privilege. This is, however, a restriction in the VEX intermediate language. In the experiments we conducted, this led to 949 (55%, Linux) and 4,908 (90%, Windows) paths that could not be verified.

Finally, the slicer and the symbolic execution engine currently do not consider the properties of *binding* and *coverage*, while determining whether a path could be used for a dynamic hook or not. Consequently, not all of the paths extracted by our prototype will be suited for the implementation of a dynamic hook. As described in Section 3.5, especially the property of binding can be a limiting factor. If a payload is only loosely bound, it is likely that the hook will introduce side effects that can lead to a crash of the system. Determining automatically whether a path has exclusive binding or full coverage is difficult though. As the discussed POCs show, even payloads that initially seem unsuited for the implementation of a dynamic hook can through subtle manipulations of the involved data structures yield very reliable hooks. To designate the binding of a payload, we thus not only have to identify whether a payload is used in multiple locations, but we also have to establish how many of those usages can be controlled by the attacker. This requires a profound semantic understanding of the data structures and functions involved.

6 Related Work

To the best of our knowledge, Petroni et al. [27] were the first to consider the hooking of transient control data. However, their work is primarily focused on the detection of *persistent* control flow modifications. Attacks on transient control data are thereby only mentioned as a limitation of their system. Hofmann et al. [15] presented a "return to schedule" rootkit that overwrites return addresses of sleeping processes to periodically invoke itself and evade hook detection mechanisms. While related to our work, this approach does not leverage exploitation techniques to change the control of an application at runtime. As a consequence, the technique only enables the rootkit to reschedule itself, but it does not allow it to intercept events within the system, which is the actual goal of a hooking mechanism.

In addition, there has also been a lot of work concerned with the possibilities of non-control data attacks. Chen et al. [10] were the first to demonstrate that non-control data attacks are indeed a dangerous and realistic threat. Sparks and Butler [38] presented DKOM as a general mechanism to hide objects within kernel space. Baliga et al. [5] extended this work and presented another class of stealthy attacks that do not have the goal of hiding objects, but rather target crucial kernel data structures to subvert the integrity of the system. Finally, Prakash et al. [28] discussed the manipulation of semantic values in the kernel to evade virtual machine introspection (VMI).

7 Conclusion

In this paper, we presented a novel hooking concept that we coined dynamic hooks. The main insight behind this concept is that existing hooking mechanisms are based on the permanent modification of *persistent* control data. As a consequence, the resulting hooks are constantly evident within the system and can be detected by verifying persistent control data alone.

Dynamic hooks solve this problem by targeting *transient* control data at *run-time*. This is achieved by applying exploitation techniques to the problem of hooking. To install a dynamic hook, an attacker will modify the internal data structures of an application in such a way that its usage will trigger a vulnerability at run-time. The hook thereby only consists of the modified data as well as the exploitation logic. This results in a powerful attack model with a wide range of possibilities as the attacker can make use of the entire arsenal of exploitation mechanisms to achieve her goal. At the same time, the hook will remain hidden in non-control data until it is triggered, which makes dynamic hooks not only powerful, but also difficult to detect in practice.

To show the applicability of the approach, we implemented a prototype that is capable of automatically extracting paths for dynamic hooks from recent Linux and Windows kernels. The experiments that we conducted prove that dynamic hooks are not only a dangerous, but are also a realistic threat that can be applied to practical scenarios such as system call hooking and backdooring. In future work, we plan to further improve our prototype implementation and to make use of it to generate integrity constraints instead of attack vectors that can then be used for the reliable detection of dynamic hooks.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive and valuable comments. This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant 16BY1207B (iTES).

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed System Security (NDSS)* (2011).

[3] BAILEY, M., COOKE, E., JAHANIAN, F., WATSON, D., AND NAZARIO, J. The Blaster Worm: Then and Now. *IEEE Security and Privacy Magazine 3*, 4 (2005).

[4] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing 8*, 5 (2011).

[5] BALIGA, A., KAMAT, P., AND IFTODE, L. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *IEEE Symposium on Security and Privacy* (2007).

[6] BENCSÁTH, B., PÉK, G., BUTTYÁN, L., AND FÉLEGYHÁZI, M. The Cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet 4* (2012).

[7] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[8] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. SELECT–Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (1975).

[9] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[10] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data Attacks Are Realistic Threats. In *USENIX Security Symposium* (2005).

[11] CUI, A., AND STOLFO, S. J. Defending Embedded Systems with Software Symbiotes. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[12] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[13] GOUDEY, H. Microsoft Malware Protection Center, Threat Report: Rootkits. Tech. rep., Microsoft Corporation, June 2012. `http://www.microsoft.com/en-us/download/confirmation.aspx?id=34797`.

[14] HEX-RAYS. IDA Pro, February 2014. https://www.hex-rays.com/products/ida/.

[15] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[16] HUND, R., HOLZ, T., AND FREILING, F. C. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium* (2009).

[17] KING, J. C. Symbolic execution and program testing. In *Communications of the ACM (CACM)* (1976).

[18] LANZI, A., SHARIF, M. I., AND LEE, W. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[19] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security Symposium* (2013).

[20] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security 6*, 4 (2011).

[21] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Symposium on Network and Distributed System Security (NDSS)* (2011).

[22] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium* (2008).

[23] MICROSOFT-RESEARCH. Z3: Theorem Prover, February 2014. http://z3.codeplex.com/.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007). http://www.valgrind.org/.

[25] PASAREANU, C. S., AND VISSER, W. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf. 11*, 4 (2009).

[26] PETRONI, JR., N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *USENIX Security Symposium* (2006).

[27] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[28] PRAKASH, A., VENKATARAMANI, E., YIN, H., AND LIN, Z. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Conference on Dependable Systems and Networks (DSN)* (Jun 2013).

[29] RAVI, S., RAGHUNATHAN, A., KOCHER, P., AND HATTANGADY, S. Security in embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 3 (2004).

[30] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In *Availability, Reliability and Security (ARES)* (2009).

[31] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[32] RUSSINOVICH, M., SOLOMON, D. A., AND IONESCU, A. *Windows Internals Part I*, 6 ed. Microsoft Press, 2012.

[33] SCHNEIDER, C., PFOH, J., AND ECKERT, C. Bridging the Semantic Gap Through Static Code Analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security* (2012).

[34] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (2010).

[35] SHACHAM, H. The geometry of innocent flesh on the bone. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[36] SHOSHITAISHVILI, Y. Python bindings for Valgrind's VEX IR, February 2014. https://github.com/zardus/pyvex.

[37] SOLAR DESIGNER. Getting around non-executable stack (and fix), Aug. 1997.

[38] SPARKS, S., AND BUTLER, J. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Phrack 11*, 63 (2005).

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy* (2013).

[40] TIP, F. A survey of program slicing techniques. Tech. rep., Amsterdam, The Netherlands, The Netherlands, 1994.

[41] VOGL, S., PFOH, J., KITTEL, T., AND ECKERT, C. Persistent data-only malware: Function Hooks without Code. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[42] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy* (2010).

[43] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[44] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering Persistent Kernel Rootkits through Systematic Hook Discovery. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[45] WEISER, M. Program slicing. In *International Conference on Software Engineering (ICSE)* (1981).

[46] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Conference on Dependable Systems and Networks (DSN)* (2012).

[47] YIN, H., LIANG, Z., AND SONG, D. Hookfinder: Identifying and understanding malware hooking behaviors. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[48] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium* (2013).