Persistent Data-only Malware: Function Hooks without Code

Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert
Technische Universität München
{vogls,pfoh,kittel,eckertc}@in.tum.de

***Abstract*—As protection mechanisms become increasingly advanced, so too does the malware that seeks to circumvent them. Protection mechanisms such as secure boot, stack protection, heap protection, $W \oplus X$, and address space layout randomization have raised the bar for system security. In turn, attack mechanisms have become increasingly sophisticated. Starting with simple instruction pointer manipulation aimed at executing shellcode on the stack, we are now seeing sophisticated attacks that combine complex heap exploitation with techniques such as return-oriented programming (ROP). ROP belongs to a family of exploitation techniques called data-only exploitation. This class of exploitation and the malware that is built around it makes use solely of data to manipulate the control flow of software without introducing any code. This advanced form of exploitation circumvents many of the modern protection mechanisms presented above, however it has had, until now, one limitation. Due to the fact that it introduces no code, it is very difficult to achieve any sort of persistence. Placing a function hook is straightforward, but where should this hook point to if the malware introduces no code? There are many challenges that must first be overcome if one wishes to answer this question. In this paper, we present the first persistent data-only malware proof of concept in the form of a persistent rootkit. We also present several methods by which one can achieve persistence beyond our proof of concept.**

I. Introduction

Traditional malware generally introduces new code to a host or alters existing code to suit its needs. To counter such malware many steps can be taken. Simple measures to protect the integrity of binaries have been around for some time in the form of tripwire[1] and similar tools. The secure boot feature of the UEFI specification also tries to hinder such malware in the kernel by making use of a trusted computing mechanisms to measure the integrity of the kernel before it is loaded [36]. Finally, typical $W \oplus X$ approaches hinder the execution of new code that is introduced in portions of memory that expect data to be stored. This is by no means an exhaustive list, but illustrates the fact that protecting against the introduction or modification of code has generally been a successful method of hindering malware.

[1]http://www.tripwire.org/

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23019

To counter such protection mechanisms, data-only malware was born. Data-only malware introduces no new code into the system nor does it change any existing code, but rather changes the control flow of existing code by introducing specially crafted data. The first example of such data-only malware was possible through an exploitation technique known as return-into-libc [35], [28]. This class of attack generally starts with a buffer overflow that overwrites the return address of the current function. This newly written return address will change the control flow to point to a function (generally in the libc library). Further careful crafting of data on the stack will even allow one to chain several function calls together [23]. Due to the general nature of libraries such as libc, the attacker can achieve anything from creating a new thread of execution to launching an arbitrary process.

Such techniques have been further generalized to use not entire functions, but rather small pieces of code several instructions long ending with a `ret` instruction. These so-called *gadgets* can be chained together by manipulating the stack to achieve most tasks. In fact, it has even been shown that such methods are Turing complete [7], [32]. This form of data-only exploitation technique has been coined return-oriented programming (ROP) [32], [4]. Further variations on this include jump-oriented programming (JOP) [7], [3], but this functions in the same fundamental manner.

While such data-only malware is quite powerful and has even been used to successfully implement rootkit functionality in the kernel [18], whether it is possible to inject persistent functionality using data-only malware is still a point of contention. Until now, such attacks were always "one shot" attacks that exploited a vulnerability to achieve some goal (e. g., hiding a process) and then finish. In order to leverage some other functionality, the vulnerability must again be exploited to achieve this new goal, but no form of persistence has been shown. Without persistence malware is severely limited, since it cannot react to events within the system. In this paper, we show that it is possible to inject persistent functionality (e. g., function hooking) into a system *without* manipulating existing code or introducing new code.

In this work we make the following contributions:

- We discuss the challenges of persistent data-only malware in the context of the latest security mechanisms.
- We present several hardware and software based techniques to enable persistent data-only malware by placing function hooks.
- We propose an architecture for persistent data-only malware.

- We introduce a proof of concept implementation to demonstrate that persistent data-only malware is feasible in current commodity operating systems.

This paper is structured as follows: In Section II we motivate our work while Section III provides necessary background information. We will then discuss persistent data-only malware and the challenges associated with it in Section IV. Additionally, we will offer several mechanisms which enable one to place function hooks with data-only malware, thus achieving persistence and propose an architecture for persistent data-only malware. In Section V, we present a proof of concept implementation of a persistent rootkit for a modern Linux operating system (OS). This is followed up by an discussion of the experiments that we conducted, the security mechanisms that we had to circumvent to implement our proof of concept, and possible countermeasures in Section VI. In addition, we describe how persistent data-only malware is able to achieve the property of residence, meaning that it is capable of surviving reboots. Finally, we close with a presentation of related work and a conclusion in Sections VII and VIII.

II. Motivation

As mentioned in the previous section, data-only malware has been shown to be quite powerful even to the extent of demonstrating rootkit functionality without persistence [18]. While this is a valuable contribution, persistence is an important aspect of malware, especially in rootkits. Petroni and Hicks estimate that 96% of rootkits require some form of persistence to achieve their goals [27]. In the most straightforward example, malware will often benefit from hooking function calls. This function hook can then perform some actions either before or after calling the original function. These actions might include simply inspecting and collecting data or more intrusively manipulating arguments or return values. In any case, this function hook generally takes the form of injected code that performs the appropriate actions and then hands control back to the original control flow.

Of issue is the fact that modern protection mechanisms, especially in the kernel, make permanently introducing or manipulating code difficult. For example, with modern hardware it is possible to have the hardware measure the integrity of the bootloader and OS kernel before loading and executing them. Further, mechanisms in modern kernels allow only signed code to be loaded or give the option to disable driver loading completely. This makes any approach that aims to introduce code into the kernel very difficult and leaves only the possibility of introducing code through a vulnerability. However, further protection mechanisms such as non-executable data segments or stack and heap protection mechanisms make even loading code through a vulnerability difficult. To circumvent such protection mechanisms one must rely on data-only malware. However, until now, there has been no data-only malware that is capable of achieving persistence.

To give a concrete example, the following is a list of popular Linux-based rootkits –the latest from Nov. 2012– found on http://packetstormsecurity.com:

- adore-ng
- eNYeLKM
- mood-nt
- override
- Unix/Darbe-A

These rootkits all rely on techniques to load themselves into the kernel that are no longer possible on a modern Linux system with secure boot and driver signing/disabled driver loading. Even if they were able to load themselves through some vulnerability, they all introduce code which is impossible to execute unless the vulnerability allows one to write into a code segment of the kernel, which is unlikely. This leaves us only with the option of persistent data-only malware if we want to load a rootkit given all these modern protection mechanisms.

Upon initial consideration, it seems unlikely that persistence is possible without injecting such a code hook. Even if one is able to manipulate a function pointer in some data structure, this is not enough for a ROP approach as either the stack pointer (SP) or the stack itself must be manipulated at that point in time as well. While some researchers dismiss the possibility of persistent data-only malware [29], others have speculated that it is possible [9], [18]. In the following, we will show that persistent data-only malware is, indeed, possible.

Once we can fully understand the threat, we can also begin to discuss the possible methods for defending against such persistent threats.

III. Background

In this section, we provide some background necessary for the understanding of the rest of our paper. We begin with some definitions followed by a look at various protection mechanisms used to hinder malware in modern OSs. Next we introduce a specific form of data-only exploitation, namely ROP, as we will use this exploitation technique for many examples throughout this paper. Finally, we will provide an overview of data-only malware by discussing its prerequisites and describing the properties of its, up-to-now, only known variant *non-persistent* data-only malware.

A. Definitions

In this section we define the three types of malware we discuss throughout our paper.

1) Resident Malware: We begin by introducing our definition of *resident malware* as this definition is straightforward. Any malware that has the ability to continue to achieve its objective without any human interaction despite a reboot or power cycle is resident in the system and is therefore resident malware. This paper only focuses on resident malware as an academic exercise in the discussion. Nonetheless it is important to distinguish between resident and persistent malware.

2) Persistent Malware: On the other hand, *persistent malware* is malware that makes permanent changes in memory and permanently changes the control flow within a system such that it can continue to achieve its objective. This characteristic allows the malware to be aware of and react to changes in the system. The simplest example of such functionality is replacing a function pointer with a pointer to a malicious function that collects the data being input to the original function.

By extension, *non-persistent malware* is malware that may make permanent changes to the system in memory, but leaves the control flow completely intact beyond the initial attack (clearly the initial attack must change the control flow once). Such malware may simply remove an item from a data structure, for example. This requires making changes in memory, but does not permanently affect the control flow of the system.

3) Data-only Malware: Finally, *data-only malware* is a malware-type that introduces specially crafted data into the system with the intent of manipulating the control flow without changing or introducing new code. Very important for this class of malware is that the instruction pointer (IP) never points to anything introduced by the malware itself. That is, such malware makes use of code that already existed before its presence.

B. Protection Mechanisms

Current OSs, including Linux and Windows, employ several protection mechanisms to mitigate attacks. The following paragraphs introduce the most contemporary of them.

1) Compiler-enforced Protection: One of the first approaches to counter buffer overflow attacks was StackGuard [12]. StackGuard places a new field (canary) between a function's local variables and saved return address on the stack. When initializing a function's local stack, the program automatically initializes the canary with a random value whose integrity is verified before returning. The general idea behind this approach is that any buffer overflow attack that overwrites the return value must also overwrite the canary value, and since this canary is initialized with a random number, it is very difficult for an attacker to guess the correct value. Based on this, the program can determine that if the canary values changed a buffer overflow likely occurred and can take appropriate actions (e. g., terminate).

Further variations of this approach "xor" the return value with a random value or use a fixed value that is difficult for an attacker to write. In case of Ubuntu Linux, for example, the canary consists of characters that terminate string operations such as a '\0' byte. An additional approach keeps a separate shadow stack for verifying return values in the form of mechanisms such as StackShield[2] and PointGuard [11].

2) Kernel-enforced Integrity Protection: Similar to using string termination values in stack canaries, Ascii Armor[3] is a Linux kernel-enforced method that controls where libraries are loaded to hinder overflow attacks. The Linux kernel loads external libraries to addresses beginning with a '\0' byte. Since '\0' is handled as a string terminator, this makes it more difficult for an attacker to write library addresses.

Beginning with the 64-bit version of Windows XP, Microsoft introduced a mechanism for monitoring the integrity of key kernel code segments and data structures called PatchGuard [1]. This mechanism is fairly straightforward and runs at regular intervals to verify the integrity of those portions of the kernel that are often patched by a rootkit or other malware such as the system service tables, descriptor tables, etc. The goal of such a mechanism is to identify and react to the manipulation of key data structures and code sections.

[2]http://www.angelfire.com/sk/stackshield

[3]http://lwn.net/Articles/31032/

3) $W \oplus X$: One of the oldest protection mechanisms is $W \oplus X$. This approach makes use of the paging (or segmentation) features of particular hardware (e. g., x86). Such features often allow a level of read/write/execute access control at page or segment granularity. The $W \oplus X$ mechanism works by marking single pages as either writable or executable, but never as both simultaneously. This prohibits an attacker from introducing new code as "data", then manipulating the system to execute that code. It also prohibits an attacker from directly introducing new code in those areas of memory reserved for code as they are not writable. That is, memory is split up to contain either code or data and the code sections can not be written to while the data sections cannot be executed.

4) Address Space Layout Randomization: Another contemporary protection mechanism is Address Space Layout Randomization (ASLR) and its kernel equivalent Kernel ASLR (KASLR) [17], in which specific code sections are not loaded at fixed or predictable locations inside the address space. Instead, these code sections are loaded at randomized offsets. This makes it difficult to employ exploits that make use of existing code. The goal of such an approach is to act as the hard counter to data-only malware such as ROP and return-into-libc. Due to the fact that data-only malware makes use of existing code snippets, these attacks are hindered when those code snippets are loaded at random offsets. This leaves the attacker in a situation in which she is forced to guess the location of gadgets or functions.

5) Supervisor Mode Execution Protection: A fairly new protection mechanism introduced by Intel is supervisior mode execution protection (SMEP) [20]. When this feature is enabled, the processor will fault when the current protection level (CPL) of the processor is less than three and an attempt to execute code for which the page's supervisior bit is not set is made. This means that the processor will not allow the execution of code in user space while operating in kernel mode. This is useful against attacks in which code is loaded into user space –which requires no special privileges– and the kernel control flow is manipulated into jumping to this code segment.

While SMEP is an Intel-specific feature, other architectures support similar features. The latest ARM Cortex-A series specifications describe an analogous feature within their Security Extensions [2]. When enabled, the processor will generate a fault if it is operating in secure mode and attempts to execute code from a page marked as non-secure.

6) Code Signing: In contrast to the other mechanisms described so far which generally aim to prevent dynamic exploits, code signing approaches work by validating the integrity of the code while it is loaded. This is accomplished by leveraging digitally signed binaries that are checked before loading and are only loaded if the binary is unchanged and signed with the key of a trusted party. As such an approach is quite restrictive, it is generally used in kernel protection rather than in userland protection.

With the assistance of hardware, code signing can be used to implement a trusted boot sequence. This works such that the boot ROM (the root of trust embedded in the hardware) only loads an untainted and signed bootloader, this bootloader only loads an untainted and signed kernel, and finally the kernel only loads untainted and signed drivers or modules.

By building such a chain of trust and rooting it in hardware one can be very certain that all code that is loaded into the kernel is untainted and trusted *at the time it is loaded*. The UEFI specification describes such a mechanism and is used in modern PCs [36]. However it is important to note that this will not prevent code from being introduced at run-time through a vulnerability, for example.

C. Return-oriented Programming (ROP)

In this section, we give a short introduction to return-oriented programming (ROP) [32] as it is a data-only exploitation technique that we will use often throughout the rest of this paper to illustrate several details. In addition, our proof of concept (POC) introduced in Section V makes use of ROP.

The fundamental idea behind ROP is to create a new program by combining existing instruction sequences. The execution order of the these instruction sequences is controlled by a data structure. As a result, ROP can get around the restrictions introduced by $W \oplus X$ (see previous section) by using the system's own instructions against it. ROP, however, requires the existence of a *control structure* that defines the execution order of the sequences.

Naturally, not every instruction sequence is suited for use with ROP. To be able to use an instruction sequence as a building block it must end with a `ret` instruction. Such instruction sequences are referred to as *gadgets*. The property of this instruction that makes it so useful is that it pops the top value off the stack into the instruction pointer (IP) redirecting the control flow to the address that was on the top of the stack. By carefully constructing the stack, it is therefore possible to execute sequences of gadgets one after another. This is achieved by placing the memory addresses of the gadgets in the order on the stack in which they should be executed. Since every gadget ends with a `ret` instruction, the final instruction of each gadget starts the execution of the next gadget by getting its address from the stack and placing it into the IP.

To make this scheme work, the stack pointer (SP) must initially point to a control structure containing the addresses of the individual gadgets, often called a *ROP chain*. In the simplest case, this can be achieved by directly copying the ROP chain in the stack such that the saved IP is overwritten by the address of the first gadget. However, this may not always be possible as the available space in the stack may be limited. In such a case, one must copy the ROP chain somewhere else in memory and point the SP to this location. This is achieved through a *stack pivot* sequence. A stack pivot sequence generally only consists of a few gadgets or, in the worst case, only one. Setting the SP to the beginning of the payload chain in such a scenario is a very challenging task. Achieving this is often only possible if the attacker additionally has control of a register or can place an additional ROP chain near the SP such that adding or subtracting an offset from the SP is enough to activate this chain. Luckily, in the case of an exploit the machine state is often predictable for an attacker such that these conditions are met.

A popular gadget [13] that is often used as a part of a stack pivot sequence is the following[4]:

[4]`Eax` is just an example here. Similarly, the technique could be used with any other register, given that a gadget `xchg <reg>, esp; ret;` exists.

Listing 1. Commonly used stack pivoting gadget in ROP-based exploitation.
```
; Load address of control structure into EAX
pop eax;
;  Stack Pivot
xchg eax, esp;
; Start the execution of the control structure
ret;
```

A prerequisite for the use of this gadget is that the address of the control structure is either on top of the stack, placed into `eax` by a previous gadget, or already contained within `eax`. The latter is, for example, the case when a vulnerable function returns a pointer to a data structure that can be controlled by an attacker. Although it seems unlikely that such a data structure address winds up in a register, it is actually a quite common exploitation scenario which is used by techniques such as "ret2reg" (Return to Register) [34].

Other examples of commonly used stack pivoting gadgets are `sub esp, <offset>; ret;` and `add esp, <offset>; ret;`, since the ROP chain often resides somewhere within the stack. Unfortunately, there is no universal gadget that can be used to pivot the stack. The instruction sequence that is used to accomplish this task always depends on the machine state at the time the exploit is triggered.

Although ROP seems to be quite limited at first glance, it has been shown that the technique is actually very powerful [7]. Given a code base as large as libc, for example, an attacker can find many different gadgets that enable her to build arbitrary functions by combining the individual blocks in a clever way. This is especially true for the x86 architecture. Due to the variable instruction format of the architecture an attacker cannot only find intended, but also unintended gadgets [32].

D. Data-only Malware

While we defined data-only malware in Section III-A, we will take a closer look in this section. We will present the prerequisites for data-only malware and discuss the properties of non-persistent data-only malware, which is the only form of data-only malware known so far.

1) Prerequisites: In order for malware to deliver a payload, a victim must first be infected. Generally, this infection can take many forms. In the simplest case, a user might be deceived into executing a malicious binary through social engineering (e. g., malware that spreads as email attachments). On the other hand, more advanced malware may be able to infect a victim without any human involvement through the remote exploitation of some vulnerability (e. g., worms).

However, when one considers data-only malware, the possibility of conducting the infection process is quite constrained. Due to the fact that an attacker may not introduce any additional code into the victim's system, simple attack vectors such as sending an email attachment are no longer an option[5]. Instead, the attacker must find a way to perform the infection by only using data. This essentially requires a *vulnerability* on the victim's system that allows the attacker to manipulate the control flow of the vulnerable software component. The initial

[5]Of course, it is possible to combine data-only malware with other malware types to open further attack vectors, but for the purpose of this paper we consider data-only malware in a purist sense.

execution of the malware will then be triggered by exploiting the vulnerability using the malware as payload. Notice that this implies that data-only malware will always be executed in the context of some pre-existing software, in contrast to traditional malware, which may also run in the context of a user.

Naturally, not every vulnerability is suited for the purpose of bootstrapping data-only malware. Instead, there are specific constraints that the required vulnerability must fulfill. First and foremost, the vulnerability must allow the attacker to control the IP. This is important as the attacker cannot introduce code, yet needs to influence the control flow. In general, this is done by careful manipulation of the IP. For example, in ROP this is achieved by constructing a volatile stack with pointers to gadgets. If the stack is properly constructed, the functionality of the `ret` instruction can be leveraged to control the execution flow. However, before this volatile stack can be used, an attacker must find a way to set the SP to its location. For the general case of data-only malware, this therefore implies that the vulnerability must not only provide control of the IP, but must also enable the attacker to activate her control structure.

In addition, the vulnerable program must provide the attacker with the ability to transfer the required control structure into memory. This requires some functionality that moves external data into memory. Hereby, it is essential that the memory that is used by the routine to store the data is large enough to contain the necessary control structure. In ROP this control structure (i. e., ROP chain) usually requires significantly more space than traditional shellcode.

Lastly, the vulnerable program must provide the instruction sequences (i. e., gadgets) that are necessary to trigger the execution of the data-only malware and to implement its functionality. While this seems as though it might be the most difficult challenge at the first glance, researchers have shown that only a relatively small codebase is required to obtain the necessary instruction sequences [32], [7].

In the following, we summarize the above mentioned prerequisites for data-only malware:

Instructions	The victim's system must contain the instruction sequences that are required to implement the malware's functionality.
Vulnerability	Data-only malware requires a vulnerability within the victim's system.
Memory	The software that contains the vulnerability, must provide a mechanism to load the required control structure into memory.
Control	The vulnerability must provide the attacker with control of the instruction pointer (IP) and enable her to activate the necessary control structure.

2) Non-Persistent Data-only Malware: Non-persistent data-only malware was, until now, the only form of data-only malware to our knowledge. The key difference between persistent and non-persistent data-only malware is that the non-persistent variety does not permanently change the control flow of a system. Instead, actions are performed by exploiting a vulnerability over and over again. At its heart, non-persistent data-only malware thus consists of an exploit that can handle different data-only payloads. In the case of ROP, a payload is essentially a ROP chain that implements the desired functionality. Each action that the malware supports is implemented as an individual payload. To execute a particular action, the exploit is triggered using the corresponding payload.

The main disadvantage of any type of non-persistent malware is that it cannot place function hooks. While the effects of non-persistent malware might last (e. g., the modification of a data structure), there is no way for the malware to respond to further actions within the system. That is, non-persistent malware is incapable of actively intercepting events that occur within the infected host and could, for instance, not be used to implement a key logger. Instead, the malware relies on an external entity (e. g., an attacker that executes it) to run.

The reason that data-only malware was, until now, always non-persistent is that it is very difficult to achieve persistence without introducing any code. Therefore, in the next section we will take a closer look at the challenges involved and how one might overcome them.

IV. Persistent Data-only Malware

Having discussed non-persistent data-only malware in Section III-D2, we will, in this section, take a closer look at *persistent* data-only malware. We will first provide an overview of persistent data-only malware, then we will discuss what challenges must be overcome to create such a malware form. For the sake of discussion, we will consider ROP-based malware on the x86 architecture in our examples. However, much of what we present is also relevant for other data-only exploitation techniques (i. e., JOP) and other architectures.

A. Overview

In contrast to non-persistent data-only malware, persistent data-only malware is capable of permanently altering the normal control flow of a software system. Loading the malware is achieved by exploiting the assumed vulnerability once. This infection process has essentially two stages. The first stage is the *initialization stage*. During this stage the vulnerability is exploited, which leads to the execution of the *initialization control structure* that performs the bootstrapping of the malware. In particular, the initialization control structure is responsible for overwriting the targeted function pointers, which will later trigger the execution of the malware, and the loading of the second stage. Consequently, this first stage is no different than in the case of non-persistent data-only malware, except that its sole purpose is to set up the second stage. This second stage or *persistent stage* then implements the persistent functionality of the malware. While this sounds very straightforward, there are several major challenges that one must overcome in order to be successful. These are described in the following section.

B. Challenges

Persistent data-only malware faces four fundamental challenges. However, before we describe these challenges in more detail, it is important to separate the challenges faced in the initialization stage and the challenges faced in the persistent stage. As it turns out, the initialization phase must face the same challenges that a traditional exploit faces. Since these challenges have already been discussed in detail in previous work ([5], [21], [23], [25], [28], [32], [33], [35]), we will not cover these challenges within this paper. Instead, we will focus on the challenges that the persistent stage faces. That is, the challenges that have to be overcome before, during, and after the execution of the persistent stage. Consequently in the following, we will assume that the initialization control structure of the malware has already taken control of the system and now prepares the execution of the persistent stage.

1) Finding a suitable memory location: First, a memory location must be located that can contain the persistent control structure of the malware. For example, in ROP we need a place to store the ROP chain that encodes our persistent behavior. It is essential that this memory location is exclusively owned by the malware itself in order to avoid the control structure being destroyed during the normal execution of the vulnerable program. As a result the stack is usually not suited for such a task. Instead a memory area must either be reserved within the system or an existing unused memory area can be occupied. The latter is, for instance, possible if the vulnerable application does not make full use of a data region that has been allocated to it. Finally, care must also be taken that this memory location is never deallocated after the initial stage has taken place.

2) Protecting against overwrites: Second, the persistent control structure has to be protected against overwrites. If the control structure is modified in an uncontrolled way, it is very likely that the malware will malfunction on the next execution. Notice that finding a memory location that is exclusively owned by the malware as described in the first challenge, is not enough to guarantee that the persistent control structure is not overwritten. In the case of ROP, for instance, we have to set the SP to point to the persistent control structure to execute it. If another thread of execution interrupts our control flow and tries to make use of the stack before we finish, it could overwrite gadgets of the persistent chain that have been executed before we were interrupted.

In general, there exist two possible types of overwrites: self-induced and interrupt-induced. The former refers to overwrites that are triggered by the malware itself. As an example, consider a `call` instruction that is part of a gadget used within a ROP chain. This instruction will essentially push the return address on the stack and then transfer control to the location specified by its operand. Since the SP points to the control structure, the call instruction will overwrite parts of it by pushing the address. In fact, the push will, in many cases, overwrite the address of the gadget that contains the `call` instruction as shown in Figure 1. This is due to the fact that the address of the gadget that is currently executing (A in Figure 1) usually resides directly before the current SP.

While self-induced overwrites have to be kept in mind when designing persistent data-only malware, they can be avoided by carefully selecting the gadgets that are used to implement its functionality. Interrupt-induced overwrites on the other hand, are overwrites that are triggered by an external event and can therefore not simply be avoided. Instead, the malware must be designed to protect itself against these overwrites. Due to the fact that interrupts are very frequent events, it is very likely that persistent kernel malware is interrupted during its execution and an interrupt handler is invoked. This interrupt handler may, amongst other things, make use of the current stack during its execution. In the case of ROP this

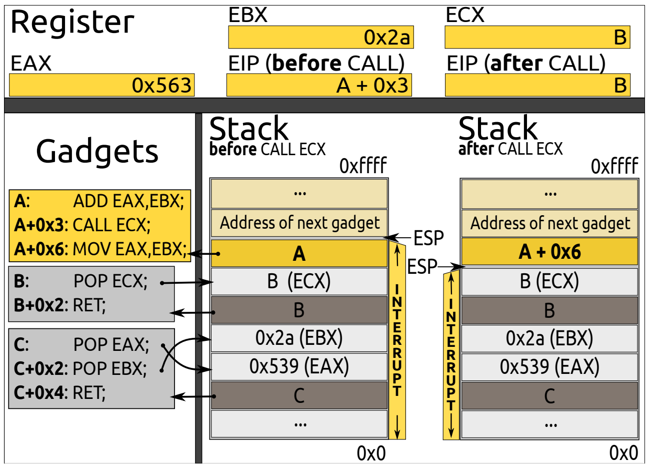

Fig. 1. Self-induced and interrupt-induced overwrites in the case of ROP. To visualize self-induced overwrites, the picture shows the state of the machine before and after the execution of the `CALL ECX` instruction at address A + 0x3. Interrupt-induced overwrites are displayed using the bars next to the stack, since they could overwrite any value before the current SP. The ROP chain that is shown uses three different gadgets to load two immediates into EAX and EBX, add their values, and store the result in EAX.

means that the part of the control structure that resides before the current SP may be overwritten.

The types of overwrites that can occur heavily depend on the technique that is used to to implement the malware (e. g., ROP) and its functionality. Interrupt-induced overwrites, for instance, usually only occur within kernel space and thus are not a big issue in user space. We will discuss the overwrites that we encountered during our implementation in more detail in Section V.

3) Resuming the original control flow: Third, since the persistent stage of the malware is invoked by a function hook, we have to make sure execution continues normally after the malware has run. This is due to the fact that the execution path, which led to the invocation of the hook, will most likely expect a result from the original function that was replaced by the malware. This result has to be provided by the malware. Otherwise, if the malware would simply try to gracefully terminate the execution path (e. g., by returning to the main function), certain code paths would never be executed, which will lead to a reduction in functionality and in the worst case a system crash. This is especially a problem for kernel space persistent data-only malware, where a failure to restore a valid execution path, will in most cases crash the entire system.

To be able to continue the original execution path, persistent data-only malware must be careful to not overwrite register or memory values during its execution that it might need later on to resume the execution path. This essentially requires that the persistent stage backs up important registers/memory locations *before* it makes use of them, unless the malware can predict/infer their values. Additionally, the malware must restore the original values before it hands back the execution.

4) Activating the control structure: Fourth and more importantly, a mechanism or specific instruction sequence must be found that activates the persistent control structure when a hook is invoked. If we once again consider ROP as an example, it is not enough that the IP is manipulated through an overwritten function pointer, but we must also manipulate the system in a way such that the SP points to the beginning of our persistent ROP chain. Since this chain is stored somewhere in memory (not on the stack) the first instruction sequence that is executed on behalf of the malware when the function hook is invoked must modify the SP to point to the ROP chain. That is, it must *switch the stack*. This *switching sequence* is a requirement for a persistent data-only malware. Notice that the switching sequence must in general be a sequence of continuous instructions. As we do not yet have control over the SP at this stage it is very difficult to build a chain of multiple gadgets. Setting the SP to a specific value under these conditions is quite challenging.

At first glance, it seems as if the previously described challenge is the same challenge that we face in a data-only exploit. In this case we also need to find a way to activate our control structure, which for ROP means that we have to point the SP to our volatile stack as described in Section III-C. However, while the problems are related, the scenarios in which we try to solve them are very different. To see this, lets us briefly compare the machine state in both situations.

In the case of a traditional ROP exploit, we usually have a very solid understanding of the state that the machine will be in when our exploit is triggered. For instance, consider an exploit that overwrote the saved IP on the stack. When the vulnerable program tries to return to this address, attacker controlled code will be executed. Since we overwrote the return address of a particular function, we know which functions will be executed before the overwritten return address is used. Due to the deterministic nature of these functions, we are able to gather a lot of information about the machine state. In particular, we will know the layout of the stack and the data types of the general purpose registers and the local and global variables, which will allow us to use techniques such as “ret2reg”. In the case of a traditional stack-based overflow, we might even be able to simply place our ROP chain on the stack by writing past the saved return address.

In contrast to this, consider persistent data-only malware which is invoked by a hook placed somewhere within the system. Depending on the location of the hook, there might be dozens of execution paths that lead to the invocation of the hook. In general, we will therefore not be able to make any assumptions about the stack or the general purpose registers. In short, the only register whose value we can predict when the hook is invoked is the IP. Since we only control the IP, but have not yet activated any control structure, we will only be able to use a single gadget to switch the stack. This is the worst case scenario that we described in Section III-C. However, in this case we neither control another register nor a buffer on the stack. Consequently, we cannot simply use common stack pivot gadgets such as the ones presented in Section III-C. Activating our persistent control structure in this situation is a difficult problem. What is even more, we will need to find a way to conduct the stack switch without corrupting register or memory values that are needed later on. After all we must hand back the execution to the previous function after we have handled the hook in order to avoid any side effects on the system as has been described in the previous section. This is also not the case for traditional exploits, where a graceful exit is in general enough to avoid a crash of the system.

An ideal initialization sequence for a ROP-based piece of malware on the x86 architecture might look as follows:

Listing 2. An ideal stack switching gadget.

```
;  store the current ESP in EAX
mov eax, esp;
; move control structure address into ESP
mov esp, &control_structure;
; trigger the control structure
ret;
```

However, it is obviously very unlikely that such an instruction sequence exists. In the following section, we discuss hardware and software-based solutions that can be used to manipulate the SP when a hooked function is called for a ROP-based approach.

C. Hardware Mechanisms

All of the following hardware-based mechanisms require the highest privilege-level to use them. Therefore these mechanisms are mainly of interest for attacks on the kernel.

1) The `sysenter` instruction: The `sysenter` instruction was introduced by Intel with the Pentium-II processor as a replacement for the interrupt-based system call mechanism. Since `sysenter` fulfills all the tasks that are required for a switch from a lower privilege level to the highest privilege level without intermediate table look-ups, it is much faster then the previously used interrupt-based system call invocation [20]. As a result, all modern OSs support the use of the `sysenter` instruction as a alternative to interrupt-based system calls.

Internally, `sysenter` relies on three model-specific registers (MSRs) to perform a context switch from a lower privilege level to ring 0. Namely these MSRs are:

IA32_SYSENTER_CS	Defines the target code segment that will be used after the context switch.
IA32_SYSENTER_EIP	Holds the IP that will be used after the context switch occurred.
IA32_SYSENTER_ESP	Holds the SP that will be used after the context switch

By carefully manipulating the `IA32_SYSENTER_EIP` and the `IA32_SYSENTER_ESP` MSRs, an attacker can control both the SP as well as the IP. In order to leverage this approach to place hooks within the system, the malware would first need to set the appropriate MSRs to point to the malware’s persistent control structure (SP) and the first gadget (IP). The hook itself then needs to point to a `sysenter` instruction within memory. As a result, every invocation of the hooked function would transfer the execution control to the malware.

A problem with such an approach is that the current SP is not saved anywhere and is simply overwritten. Therefore extra steps must be taken to restore the original SP after the malware executes. How this is achieved heavily depends on the particular hook and OS used. For a specific example on how this problem can be solved, see our POC in Section V-B.

Finally, it may seem that such an approach would break the original system call mechanism, however this need not be the case. First, it is often the case that 64-bit OSs prefer to use yet a third mechanism for implementing system calls, namely the `syscall` instruction. In this case, we need not worry as the `sysenter`-based mechanism is not in use anyway. On the other hand, if the host does make use of the `sysenter` instruction, the malware must simply handle this case and determine whether the call to our hook is the result of a "real" system call or the result of a function hook and react appropriately. In the case of a "real" system call, the malware simply needs to hand control to the system call dispatcher.

2) Task State Segment (TSS): Although the feature is not used by most modern OSs, the x86 architecture provides a hardware mechanism for performing context switches between processes. For this purpose there exist so-called TSS descriptors, which are part of the Global Descriptor Table (GDT). By invoking a TSS descriptor, an attacker can load a completely new execution context. Consequently, persistent data-only malware can make use of this feature to control the IP as well as the SP during the invocation of a hook. To achieve this, the malware must first set up a TSS descriptor and then point the function hook to an instruction sequence that invokes this descriptor. A *far jump* to the descriptor is often used for this purpose[6] (`jmp <tss_desc>:0x0000`).

When the hook is executed the machine will then use the TSS descriptor to perform a "context switch" to the malware. During this process the hardware will first save the value of all general purpose registers in the TSS descriptor of the current task, before setting them to the values stored in the just activated TSS descriptor. This allows the malware not only to load a completely different execution context, but also enables it to easily access and restore the old execution context.

While this approach is very powerful, it is restricted to 32-bit systems. On 64-bit systems the x86 architecture no longer supports the above described context switching feature. However, while context switching has been disabled, a new mechanism has been introduced to the TSS that similarly allows an attacker to control the SP, the Interrupt Stack Table (IST). The IST is essentially a table of pointers, where each pointer contains the address of a memory region that can be used as stack region by an interrupt handler. This mechanism allows the kernel to individually assign a stack from the IST to each interrupt handler.

In the x86 architecture, interrupt handling is based on the Interrupt Descriptor Table (IDT). The IDT contains an interrupt-gate descriptor for each individual interrupt. Amongst other things this descriptor specifies the address of the interrupt handler that should be invoked. Whenever an interrupt occurs, the number of the interrupt is used as an index into the IDT. Based on the number of the interrupt, it is therefore possible to obtain its corresponding interrupt-gate descriptor, which in-turn specifies the address of the interrupt handler.

Besides the address of the interrupt handler, the interrupt-gate descriptor also contains an index into the IST. Should this index be greater than zero, the hardware will load the address contained within the specified IST entry into the SP, before invoking the interrupt handler. In addition, the machine will push the old value of the SP and the IP as part of the interrupt-stack-frame onto the new stack such that the interrupt handler is able to restore their original values after its execution.

[6]The interested reader can find an overview of other possible sequences in Section 7.3 of the Intel Software Developer's Manual 3A [20].

To use this mechanism for a stack switch, the initialization stage of the malware has to point one of the entries within the IST to the location of the persistent stage. Additionally, one of the interrupt gate-descriptors in the IDT has to be setup to use this modified IST entry and to point to the first gadget in our persistent ROP chain. This first gadget must increase the SP by the size of the interrupt-stack-frame as the hardware automatically pushes this frame to the new stack when the interrupt is invoked. Finally, the hook has to be set to a gadget that invokes the interrupt whose descriptor was prepared in the way just described. The invocation of the hook will then lead to an interrupt, which will in-turn lead to stack switch that in combination with the stack increasing gadget will lead to the execution of the persistent chain. Once the persistent chain finished its execution, it can restore the original SP and IP from the interrupt-stack-frame [20].

D. Software Mechanisms

Having described some hardware mechanisms in the previous section, we will now introduce several software mechanisms for switching the stack. These software-based mechanisms are specific to the hook that is placed within the system. Therefore, these approaches are not universal, but serve as examples of what is possible.

1) Adapting the location of the control structure: The main problem that hinders us from using one of the common stack pivot gadgets described in Section III-C to switch the stack is that we neither control a register value nor a buffer on the stack when a hook is invoked. However, similar to the case of heap-based ROP exploits we might be able to control the location in which our persistent ROP chain will reside. If this is possible, we can circumvent this problem by placing the persistent control structure of the malware above the stack of the process. By this we simply mean that the control structure must be loaded at an address that is smaller than the original stack base minus the maximum stack size of the process. A switch to the persistent control structure can then be performed using the common stack pivot `sub esp, <offset>; ret;`.

Since the malware control structure resides above the process stack, it will not be destroyed during the normal program execution given that the maximum stack size of the process is known. Whether it is possible to place the persistent chain at such a location often depends on the structure of the vulnerable program as well as the vulnerability itself. In general, an attacker can apply exploiting techniques such as heap spraying [10] to influence the memory allocation.

Finally, notice that the constant stack offset must not necessarily point directly to the malware's control structure. Instead the attacker can introduce a NOP sled at the beginning of her control structure (in ROP terms a NOP is simply the address of a `ret` instruction). In this case the constant offset must simply point somewhere into the NOP sled. The success of the approach will then depend on the variation of the stack –which generally occupies only one or two pages– and the size of the sled.

2) Adapting the location of the stack: Instead of placing the persistent control structure at a suitable location above the stack, it is also possible to change the location of the stack itself. This is due to the fact that OSs usually store the address of the stack for each process and the kernel in specific registers or memory locations. In the case of Linux, for example, the kernel SP is stored within a `per_cpu` variable. When the kernel switches from user space to kernel space, it will load the address stored at this location into the SP. Thus by overwriting the stored address, an attacker is able to set the kernel stack to an arbitrary memory location. Similarly, an attacker can overwrite the saved SP of a single or multiple processes.

If the attacker controls the SP through this technique, she can, for instance, place the SP in front of the persistent control structure. The stack switch can then be performed using a gadget such as `add esp, <offset>; ret;`.

3) Using function pointer chains: One of the biggest problems that persistent data-only malware faces when attempting to set hooks is that until the stack is switched (in the case of ROP) it can only rely on a single sequence of instructions to perform the necessary task of activating the control structure. If the malware could chain multiple instruction sequences this task would be much easier since it could combine multiple gadgets to reach this goal.

One possibility that would allow one to create a small chain of instruction sequences is to overwrite multiple function pointers that are called in sequence. To demonstrate this, consider the following example: The vulnerable program contains a global buffer that is located in the data section. Imagine that the initialization stage loaded the persistent control structure in this buffer. It is very likely that the vulnerable program contains various instruction sequences that operate on the buffer. For example, there may be a 'strncpy' operation that copies data into the buffer. On the x86 architecture this could result in the following assembler code:

Listing 3. Call to 'strncpy' in assembler.
```
1 mov [esp + 8], size;
2 mov [esp + 4], &source;
3 ; move absolute address of the global buffer
4 ; to the top of the stack
5 mov [esp + 0], &dest;
6 call <strncpy@plt>
```

This provides the malware with an instruction sequence that loads the absolute address of the buffer onto the stack (Line 6). In addition, the malware can control the function call in Line 7. This is due to the fact that library functions (e. g., strncpy) are called as function pointers that are offsets within a global table. In Linux this table is called the Global Offset Table (GOT) while in Windows this table is referred to as the Import Address Table (IAT). If we continue with our example, overwriting the function pointer of the strncpy function within the GOT[7] allows the malware to execute a second instruction sequence that loads the absolute address into the SP.

Note that while this is a contrived example, it is quite generic and the constructs used are very common practice. The prerequisites for such an approach are (1) the existence of a global buffer, (2) a library function that operates on that buffer, and (3) a writable table that facilitates the linking of library functions. In both Windows and Linux environments these are commonly found in processes.

[7]Notice that this overwrite has to occur during the initialization stage. When the persistent stage of the malware is invoked by a hook, the function pointers must already be overwritten.

E. Architecture

Up to this point, we have presented a rather abstract view of the architecture of persistent data-only malware. To discuss the challenges associated with the creation of data-only malware and the mechanisms that can be used to activate it, we considered two stages: the initialization stage and the persistent stage. In this section we want to refine this view and present a concrete architecture for persistent data-only malware. This architecture is shown in Figure 2. As one can see, the architecture makes use of four different control structures: the initialization chain, the copy chain, the dispatcher chain, and the payload chain. While the initialization stage of the malware only consists of a single control structure (initialization chain), the persistent stage has been divided into the copy chain, the dispatcher chain, and the payload chain. In the following we will describe each of the chains in more detail. In the process we will also state which of the previously described challenges each individual chain faces.

1) Initialization Chain: As has been described in section III-D, data-only malware is loaded using a vulnerability. The initialization chain (1) of the persistent data-only malware is the component that is executed during this initial exploitation phase. Since this component is only executed *once*, it acts very much like more traditional ROP exploits, which means that it does not require an exclusive memory area and is not affected by overwrites as outlined in Section IV-B1 and Section IV-B2, respectively. In addition, the initialization chain usually does not have to restore the original execution path as outlined in Section IV-B3. Instead, any execution path that leads to a graceful exit is in general sufficient.

The initialization chain is responsible for conducting all steps that are necessary for bootstrapping the execution of the persistent stage. In particular, it must place a hook (2) within the victim's system, setup a switching mechanism (3) as outlined in Section IV-B4, and copy the copy chain (4a) into memory. The latter requires that the initialization chain solves the first challenge we discussed in Section IV-B1. That is, the initialization chain must copy the copy chain to a memory location that is exclusively owned by the malware.

In addition, the initialization chain may have to create global state (4b), if the malware requires this. State is essential if the malware requires data to be stored across multiple invocations. Such a data area can either be integrated into the copy chain or be placed at a separate memory location as shown in Figure 2. In any case, the memory region used to contain the state must - similar to the copy chain - be exclusively owned by the malware.

2) Copy Chain: The copy chain is invoked every time the hook that the initialization chain placed is triggered. In particular, the hook will transfer control to the switching mechanism, which in-turn will invoke the copy chain.

The copy chain is the only truly persistent chain of the malware. Due to this fact it faces the most restrictions and

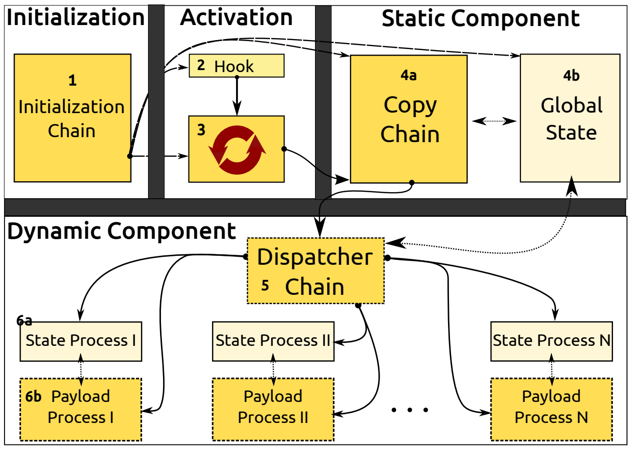

Fig. 2. Overview of the proposed architecture for persistent data-only malware.

must be carefully created to avoid overwrites as outlined in Section IV-B2. It fulfills two main tasks. First and foremost, it must save the values of all general purpose registers when it begins execution in order to be able to restore the original register values after the malware has been executed as outlined in Section IV-B3. To achieve this, the malware may only leverage gadgets that use registers which have already been saved, must not be saved according to the calling conventions, or whose values can be predicted. Consequently, this chain is severely limited when it starts execution, but will have access to an increasing number of gadgets with every register it saves. The values of the registers can be stored in the global state.

However, even when all registers have been saved, the copy chain is still tightly restricted as it must be executed with interrupts disabled and cannot invoke external functions to avoid overwrites as discussed in Section IV-B2. Such restrictions could severely limit the functionality of the malware. To solve this problem, the copy chain creates a separate dynamic component upon each invocation of the hook. To do this, it simply copies the next control structure (the dispatcher chain in case of Figure 2) to a predefined memory area that is created by the initialization chain. Once activated, this dynamic control structure can then execute without having to consider self-induced overwrites as it is dynamically created on the fly for every invocation of the hook.

While this approach is sufficient for malware that only infects a single process, kernel-level malware may set a hook that can be triggered by multiple executions paths that are running concurrently. Consider a hook on a system call, for instance. When a system call is invoked, it is not guaranteed that the system call execution will return before a context switch and a different process makes the same system call. As a result, the same hook may be triggered multiple times by different processes simultaneously. In such a case, it is possible that the dynamic component of the previous system call hook is overwritten by the currently executing hook. This situation would likely lead to a kernel crash once the context and execution of the previous process is restored.

The most straightforward method to avoid this may simply be to disable interrupts. While we make use of this method in the copy chain, it is not an ideal solution for the entire dynamic component. This is due to the fact that this would lead to a further constraint for the malware. This constraint being that the malware may not use any external functions, since they may reenable interrupts during their execution. As we want to keep the malware as constraint-free as possible, we look to a more elegant solution. To this end, we make use of a dispatcher chain (5).

3) Dispatcher Chain: A dispatcher chain (5) is required whenever multiple concurrent threads of execution can invoke one of the hooks used by the malware. The general idea behind a dispatcher chain is to create an individual payload for each process. To achieve this the chain allocates an individual memory area for each process and copies the payload chain (6b) into this memory area on each invocation occurring in the context of the process. Similarly, the dispatcher will create individual state (6a) for each process, where information such as the register values for this process, which are copied from the global state, are stored. The dispatcher must therefore also guarantee that a specific payload chain will always have access to the same state area. To do this the dispatcher must patch the address of the state area into the payload chain at run-time.

This approach provides each process with individual persistent state and a unique payload and thus effectively avoids the problem described above. Finally, notice that the dispatcher chain must be directly invoked by the copy chain and can only make use of external functions that do *not* enable interrupts. In other words, interrupts must remain disabled while the dispatcher chain is executing.

4) Payload Chain: The payload chain (6b) contains the actual functionality of the malware. Since it is recreated on each invocation by the dispatcher chain and is additionally unique for each process, it is neither affected by self-induced overwrites nor by interrupt-induced overwrites. That is, the malware can, at this point, invoke any external function and can make use of any register that has been saved by the copy chain. Thus the payload chain is essentially a traditional ROP-chain with the benefit that it may make use of persistent state to store data between invocations. As a result, the payload chain is very flexible and is only limited by the gadgets that the victim's system provides.

At the end of its execution, the payload chain must restore the original register values and hand control back to the execution path that was executed before the hook was invoked as outlined in Section IV-B3. While the former can be easily achieved, since the register values are saved by the copy chain and placed into the process state by the dispatcher chain, the latter requires the restoration of the original SP. Since this information may be lost (depending on the switching mechanism that is used), this process is usually application dependent and must be solved on a case by case basis. In general, the frame pointer (FP) can be used to restore the original SP given that the frame size of the function executing before the hook is invoked is known. We will describe the solution that we used to solve this problem in more detail in the next section.

V. Proof of Concept

To demonstrate the feasibility of the above discussed concepts, we implemented a persistent data-only ROP rootkit.

Our POC rootkit was implemented for a 64-bit Ubuntu 13.04 server standard installation. We chose to implement our POC in the kernel as this demonstrates an especially dangerous form of malware. However, we argue that the concepts outlined can easily be leveraged to infect a userland process as well.

A. Attack Model

For our POC we assume a local attacker that has user-level access. Further we presume a vulnerability in kernel space which enables us to load our rootkit. To provide a realistic attack scenario, we used a real Linux kernel vulnerability for this purpose in our POC. It is interesting to note that generally one assumes that an attacker has root privileges at the time she is ready to install a rootkit. However, since loading our rootkit requires a vulnerability, it can generally be loaded without requiring root privileges. We will defer the description of the vulnerability that we used to the next section of the paper.

Additionally, we are using a standard installation of a 64-bit Ubuntu 13.04 server with an UEFI BIOS which includes a variety of the security mechanisms presented in Section III-B[8]. In particular, our attack model assumes the following system/kernel level protection mechanisms:

- UEFI secure boot
- disabled module loading and disabled `/dev/kmem`
- stack canaries
- stack reordering
- $W \oplus X$
- module ASLR

B. Implementation

In this section we describe the implementation of our persistent ROP rootkit. In the interest of space, we cover only the interesting points from a research perspective. For further technical detail, we encourage the reader to read our technical report and inspect the exploit source code itself.[9]

Before we go into the details of our implementation, we begin with a quick overview. Our POC is designed according to the architecture presented in Section IV-E. To refresh the reader, the architecture consists of two stages, the initialization stage and the persistent stage. These stages are further divided into four ROP-chains. The initialization stage is only composed of a single ROP-chain, the initialization chain. The initialization chain is only executed once during initial exploitation and its single purpose is to setup the execution of the persistent stage. The persistent stage on the other side is composed of three different ROP chains, the copy chain, the dispatcher chain, and the payload chain. The copy chain is thereby the only truly persistent chain. It is invoked whenever the hooks the malware placed into the system are triggered. On every invocation the copy chain builds a dispatcher chain in a predefined memory area. The dispatcher chain will then in-turn create an unique state and a unique payload chain for each process. The payload chain provides the actual functionality of the rookit. In the case of our POC, the rootkit hooks the read and getdents system call to provide key logging, process hiding, and file hiding. We chose to implement these mechanisms, to demonstrates that persistent data-only rootkits can indeed provide functionality similar to traditional rootkits.

[8]https://wiki.ubuntu.com/Security/Features

[9]http://www.sec.in.tum.de/persistent-data-only-malware/.

1) Initialization Stage: The initialization phase in a kernel rootkit requires a vulnerability in the kernel code. We used the real Linux kernel vulnerability CVE-2013-2094[10] for this purpose. This vulnerability essentially allows a user space application to take control of a pointer variable within the kernel. With the help of this pointer variable the application can increase the value of memory words within kernel space. By increasing the address of an interrupt handler, the handler can be made to point to a `leave; ret;` gadget in the kernel. The `leave` instruction moves the current FP into the SP and then pops the current value on top of the stack into the FP (`mov rsp, rbp; pop rbp;`). By placing the address of the initialization chain into the FP, we can use this gadget to start the execution of our initial ROP-chain. For this scheme to work, we have to trigger the interrupt handler that we modified by provoking an exception or executing an `int X;` instruction, where X is the number of the interrupt whose handler was changed. In addition, we have to setup the FP to point to our initial ROP-chain, which will then be loaded into the SP by the `leave` instruction. Since the attacker triggers the exploit, this is not a problem in this scenario. Notice, however, that this stack pivoting mechanism is only possible as we control other registers besides the IP (the FP in this case). The triggering of a hook on the other side is *not* controlled by an attacker, which implies that we cannot simply use such a technique as a switching mechanism. This demonstrates the difference of stack switching in the case of exploits and hook invocations at a practical example. For a more detailed description of CVE-2013-2094 and our exploit, we refer the reader to our technical report.

Once the initialization chain gains control, it will allocate three memory areas within kernel space. First, memory for the global state is allocated. The address of the state is then patched at run-time into every location where it is used within the copy chain and the dispatcher chain. In the next step, memory is allocated for the dispatcher chain. The address of this memory area is then once more patched into a predefined location within the copy chain such that the copy chain can directly use it during its execution. This step is necessary, since the copy chain needs to copy the dispatcher chain into this memory region on every invocation. Notice that the copy chain cannot simply allocate memory as this would involve an external function call that would overwrite parts of the persistent chain. Finally, memory for the copy chain is allocated and the copy chain is copied into this memory area.

At this point the initialization chain sets up our stack switching mechanism. In this case, we use the `sysenter` mechanism described in Section IV-C1. This means that our initialization chain must write the correct values in the `sysenter` MSRs. Specifically, the address of a `ret` instruction is written into the `IA32_SYSENTER_EIP` MSR and the address of the copy chain is written to the `IA32_SYSENTER_ESP` MSR. Finally, the hooks are set by overwriting the read and the getdents system call in the system call table with the address of a `sysenter` instruction.

2) Persistent Stage: Once the initialization phase is completed, the hooks are set and the persistent stage is waiting to be triggered. The triggering of the hooks is illustrated in

[10]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2094

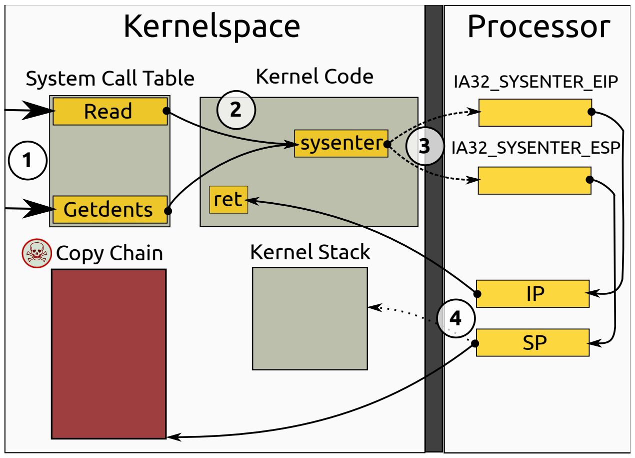

Fig. 3. Overview of the sysenter hook on the read and getdents system call that is used by our POC to permanently alter the control flow of the system.

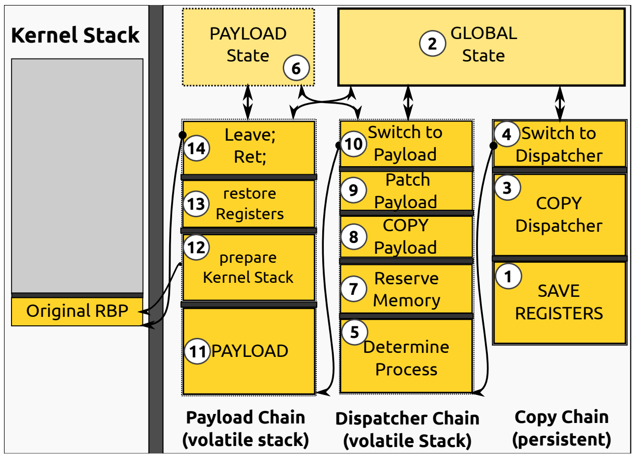

Fig. 4. Overview of the persistent stage of the rootkit and individual chains used in this stage.

Figure 3. As the read and the getdents system call function pointer were overwritten in the initialization stage, any call to the read or getdents system call (1) will result in the `sysenter` instruction being called (2). As described in Section IV-C1, the `sysenter` instruction will load the new IP from the `IA32_SYSENTER_EIP` MSR and the new SP from the `IA32_SYSENTER_ESP` MSR (3). Since we overwrote these MSRs, our copy chain is executed (4). It is important to note that the `sysenter` instruction is our mechanism for switching the stack and is completely *independent* of the fact that we are hooking system calls.

When a hook is triggered, it invokes the switching mechanism, which in-turn invokes the persistent stage of the rootkit. As outlined in Section IV-E, interrupts must be disabled when the persistent stage of the malware is invoked to avoid interrupt-induced overwrites. In our POC, this is achieved with the help of the `sysenter` instruction that automatically disables interrupts when it is executed. Notice that both of the proposed hardware-based switching mechanisms provide this feature. Should a software-based switching mechanism be used where interrupt-induced overwrites are an issue (e.g. in kernel space), the gadget that performs the stack switch must also disable interrupts.

An overview of the persistent stage of the POC is shown in Figure 4. The first chain that is activated in the persistent stage is the copy chain. The first important task that this chain performs is to save the current state of the CPU (1) such that control can be gracefully restored after the payload has executed. The copy chain stores the values of critical registers in the global state (2). To use this approach, the initialization chain patches the addresses of the global state into the copy chain before its first execution.

After the registers have been saved, the copy chain copies the dispatcher chain (3) into the memory area that has been preallocated for it by the initialization chain. As soon as this copy operation is completed, execution is transferred to the newly created dispatcher chain (4). The dispatcher chain can now use all of the registers saved by the copy chain.

The dispatcher chain starts by obtaining the current process data structure (5). To do this we make use of a Linux data structure that is associated with each CPU, the `per_cpu` data structure. This data structure contains, among other things, a pointer to the `task_struct` structure of the process that is currently executing. The rootkit obtains this pointer and uses it to locate the state (6) for this specific process. For this purpose, the global state contains an array that stores a pointer to the state of each process together with the address of it's `task_struct` pointer. By searching this array for the `task_struct` pointer of the process, the rootkit can thus obtain the address of the process's state area. If a state does not yet exist for the current process, the dispatcher chain will allocate a new memory region for the state and store it together with the `task_struct` pointer in the array.

Once the address of the state of the process is known, the dispatcher chain will copy the values of the registers that have been saved by the copy chain (and currently reside within the global state) into the process state. In the next step, the dispatcher will allocate a new memory area for the payload chain of the process (7) and copy the payload chain into the newly allocated region (8). Finally, the dispatcher must patch (9) all the locations in the newly created payload chain that refer to the process state (6) or the global state (2) at run-time.

After the payload has been patched, it is ready for execution and the dispatcher will transfer the control to the newly created chain (10). At this point the payload chain is free to enable interrupts as the payload is now unique for each process. The payload (11) will then execute the desired functionality. In the case of our POC this functionality consists of key logging, process hiding, and file hiding. To achieve the former the rookit will copy every character that is typed by the user into a buffer within the process state. Data typed by the user is thereby identified based on the file descriptor that is specified in a read system call. As soon as the user types a return character, which marks the end of a command, the data in the process state is interpreted. When the data within the state corresponds to a specific rootkit command such as hiding a specific process, the rootkit will execute the command and delete the data within the buffer. Otherwise the rootkit will write the data entered by the user to the kernel log. Thus the attacker is able to control the rootkits behavior from the command line.

Process hiding is realized by setting the PID of the process that should be hidden to zero. As a result, the process will no longer be displayed by programs such as `ps`. The process that is to be hidden (or unhidden) can thereby be specified using the above described communication mechanism. File hiding on the other hand is realized by intercepting the getdents system call and removing any entries from the returned structures that should be hidden. The filename of hidden entries must thereby begin with a predefined string. Since these are well-known techniques, we will not describe them in more detail at this point. Notice that the payload can distinguish between read and getdents system calls based on the system call number.

At the end of the payload chain, the original execution path must be restored and control must be handed back to the kernel. In our POC implementation we make use once more of a `leave; ret;` gadget (14) for this purpose. By placing the value of the kernel SP into the FP, we can switch the SP back to the original kernel stack using this instruction. However, deducing the original value of the kernel SP remains an issue as the SP was overwritten at the moment our copy chain assumed control through the execution of the `sysenter` instruction. To solve this problem, we once more rely on the `per_cpu` data structure, which also contains the value of the SP at the time when the kernel assumed control from user space. By reading this value and subtracting the stack frame size of the system call handler (the function that is executed immediately before our hook is invoked), we can calculate the value the SP had before the `sysenter` instruction was invoked. We were forced to use this approach as the FP was not set in our case (i.e., we find ourselves at the bottom of the stack). Usually, however, it is possible to use the FP in the same manner we use the saved SP in the `per_cpu` data structure. That is, one can subtract the size of the current frame from the FP to obtain the original value of the SP.

To make use of a `leave; ret;` gadget, we first have to prepare the kernel stack for the switch (12). Since the `leave` instruction will move the FP to the SP and then pop the current value on top of the stack into the FP, we have to copy the original FP onto the kernel stack. This will ensure that the original FP is loaded, when we hand control back to the kernel.

Finally, there is one last step we must take to ensure a smooth transition back to the original execution flow. We have to restore the original register values (13) that were saved by the copy chain. These registers values are currently stored within the payload state. Before the `leave; ret;` instruction is executed, the stored values are transferred back into the corresponding registers. The only exception is the FP. Since the FP is used by the `leave` instruction, it will be loaded with the address of the kernel stack pointer. The original value of the FP will than be restored in the process of the stack switch as has been described above.

VI. Discussion

In this section we will first summarize the experiments that we conducted to test our POC implementation. We will then revisit the protection mechanisms introduced in Section III-B and describe how our POC implementation circumvents them. Finally, we will discuss possible countermeasures against data-only malware and explain how persistent data-only malware can achieve the property of residence.

A. *Verification*

We conducted three separate experiments with our POC. The goal of the first test was to verify whether data-only malware could indeed be used to infect a standard server system with the default Ubuntu security features[11]. For this purpose we installed the default version of 64-bit Ubuntu 13.04 server on a Virtual Machine (VM). We then loaded our rootkit by exploiting the vulnerability described in the previous section. To test the functionality of the rootkit, we activated key logging and hid several processes and files within the guest system. As expected the hidden processes and files where no longer visible to standard programs such as `ps` and `ls`. In addition, the keystrokes were successfully logged by the rootkit and written to the kernel log. This entire process was accomplished without introducing a single instruction.

In the second experiment we wanted to validate that our persistent data-only rootkit is also able to execute despite all the security mechanisms outlined in our attack model, including a trusted boot setup (see Section V-A). To conduct this experiment we made use of OVMF[12], which provides UEFI support for VMs. We then configured a trusted boot process using a self-signed boot loader and kernel image. This secure boot process combined with the fact that loading additional kernel modules is disabled results in a very strong kernel for which it is very difficult to introduce any new code. Such a setup would hinder most modern rootkits. As in the first experiment, our POC successfully loaded itself into the kernel. In addition, the functionality of our rootkit was not affected and the rootkit was still able to log keystrokes and to hide files and processes. This demonstrates that a persistent data-only rootkit is a very dangerous form of malware that is able to load itself where many modern rootkits would fail.

Finally, we wanted to verify whether the sysenter-based switching mechanism that we use in our POC is hindered by current OS protections such as PatchGuard. For this purpose, we setup a fully patched Windows 7 Professional 64-bit guest system. We then modified the sysenter MSRs in the guest and monitored the execution of the guest system. The guest remained operational without incident. In addition, the values within the sysenter MSRs remained unchanged. This shows that protection mechanism in popular OSs such as Windows and Linux do not verify the state of the sysenter MSRs.

B. *Protection Mechanisms Revisited*

In Section III-B we summarized several protection mechanisms that are used by current OSs. Within this section we will discuss these mechanisms in the context of persistent data-only malware and our POC implementation of a persistent rootkit.

Due to the described protection mechanisms, writing malware and reliable exploits has become quite challenging. This is especially true for kernel-level malware, since the kernel only provides a well-defined interface for loading code into kernel space. Due to code signing, this interface becomes even more restrictive. As a result, most contemporary Linux rootkits cannot be loaded as stated in the motivation for this work.

[11]https://wiki.ubuntu.com/Security/Features
[12]http://www.linux-kvm.org/page/OVMF

Data-only malware on the other hand is not affected by these restrictions, since it makes use of a vulnerability to load itself. While "normal" malware could use a similar approach to load itself, it would still be detected by code signing and code integrity mechanisms that verify kernel code not only at load time, but also at run-time. In contrast, data-only malware is, by design, neither affected by code signing or code integrity approaches nor by mechanism such as $W \oplus X$.

The only protection mechanisms that must be circumvented in the case of data-only malware are exploit prevention mechanism such as stack canaries and ASLR. Whether these mechanism can be bypassed heavily depends on the vulnerability. In our POC implementation we used a real-world vulnerability to provide a realistic attack scenario. In the case of this vulnerability we can defeat stack canaries by controlling a pointer variable, which allows us to directly change data in memory without having to modify the canary. ASLR on the other side is not an issue as we override a pointer to an interrupt handler in our exploit. The location of this pointer can be obtained by getting the base address of the IDT from the Interrupt Descriptor Table Register (IDTR) and adding the offset of the handler to this base address. The initialization chain itself is than run from an attacker controlled location.

What is most interesting though, is the fact that all of these security mechanisms also have to be circumvented in the case of non-persistent data-only malware. That is, non-persistent data-only malware must also defeat exploit prevention mechanisms to be executed. While persistent data-only malware faces additional technical challenges, which we discussed in detail in Section IV-B, the malware type does not require the circumvention of additional security mechanisms for its implementation. Consequently, if non-persistent malware is considered to be a threat, persistent malware should be considered an equally likely threat, since, from a initial exploitation point-of-view, they are extremely similar.

C. Countermeasures

In the following section we discuss the security impact of our data-only persistent rootkit. While we have shown that our POC is able to circumvent a high level of security there are, of course, methods by which one could hinder such malware.

Data-only malware raises the bar for detection because it does not introduce any code and data tends to change frequently during the normal operation of a system so it is difficult to discern between benign and malicious changes to data. However, while persistence is a crucial feature for many malware forms, there are side-effects introduced by persistence that make detection easier compared to data-only malware that does not permanently reside within the system. In order to achieve persistence we stated that the control flow of a software system must permanently change. This generally requires that some function pointer is overwritten. These hooks (i. e., overwritten function pointers) are an apparent place to start when considering how one might detect persistent rootkits. However, manually auditing the integrity of all function pointers in a modern kernel would be a herculean task. Thankfully researchers have suggested methods for detecting hooks or function pointers without having to manually inspect all function pointers. Wang et al. propose two such methods. The first method works by tracing the execution of common processes that query the kernel for information (e. g., ps, netstat, etc.) and identifying the function pointers in the execution trace [39]. The second method makes use of virtualization to redirect function pointers to a single physical memory frame that the hypervisor can protect [38]. Additionally, Carbone et al. [6] as well as Schneider et al. [30] propose dynamic tools for finding function pointers among other things within the Windows and Linux kernel, respectively. All of these methods, while proposed for more general means would also be effective against data-only rootkits as there is no way to get around having to set hooks if persistence is the goal.

In addition to detecting hooks, there are more generic approaches to hindering ROP (or other forms of data-only attacks) in general. For example, ASLR attempts to randomize the layout of code such that consistently finding gadgets becomes difficult. While such an approach would be detrimental to any data-only attack in theory, ASLR has constraints of its own. For example, the Linux kernel randomizes the layout of the kernel modules, but leaves other parts of the kernel code in static locations for technical reasons. On the other hand, modern Windows versions randomize the location of kernel components but do not perform any layout randomization within those components. That is, if one can find a single address within such a component, the address of any gadget within that component can be calculated. As it turns out, finding such addresses is straightforward especially with API functions such as NtQuerySystemInformation. Additionally, researchers have proposed more advanced side channel attacks for finding such addresses [19]. So, while the concept of ASLR can be a strong hindrance to data-only malware, its implementation is still lacking.

Another general approach to detecting ROP is achieved by looking for inconsistencies on the stack when a `ret` instruction is executed. Generally such an approach works by taking a snapshot of the stack when a `call` instruction is executed, then checking whether the changes on the stack or to the stack itself are consistent with normal operation when the analogue `ret` instruction is called. The snapshot of the stack is often referred to as a shadow stack and several implementations have been proposed on various hardware architectures [16], [37], [15].

A further approach that targets ROP directly is an approach that attempts to remove usable `ret` instructions completely from code. Li et al. [22] present an approach that essentially replaces both the `call` and `ret` instructions with some bookkeeping code followed by a `jmp` instruction. Another compiler based approach to mitigate ROP was introduced by Onarlioglu et al. [24]. They introduce alignment sleds between different instructions to prevent unaligned `ret` or `jmp` instructions. In addition, they also encrypt addresses used for returns or jumps by XOR-ing random data with the addresses. This is why the remaining gadgets can also not be used by an attacker. Finally, there are also approaches like that of Pappas et al. [26] that use static in-place code transformations (e.g. instruction reordering) and instruction substitution.

D. Residence

Until now, our discussion was focused on the persistence of data-only malware. However in Section III-A, we make a clear distinction between persistent and *resident* malware. In this section, we will cover the property of residence in more detail. To refresh the reader, resident malware is simply malware that is able to survive a reboot without any human interaction.

In order to survive a reboot with data-only malware, one must automatically execute the initialization stage of the malware. This effectively requires the re-exploit of the vulnerability, after each boot. Consequently, resident data-only malware places additional constraints on the vulnerability that is used for infection. First, the vulnerability must be contained in a program that is executed during boot. This ensures that the malware is loaded with the system. Second, the vulnerability must be *self-triggering*. That is, the vulnerable program must read some external data source such as a configuration file during its initialization, which will trigger the exploit and start the initialization stage of the data-only malware.

When considering persistent data-only malware that is also resident, surviving a reboot means that the initialization stage described in Section IV must be invoked again after boot. However, if the vulnerability used to achieve persistence does not adhere to the constraints mentioned above, then a second vulnerability must be found. But, this vulnerability must not necessarily be in the same piece of software. As an example, consider the persistent ROP rootkit we presented. It uses a kernel-level vulnerability to load itself that is not self-triggering. To solve this problem we can make use of a self-triggering user space vulnerability to bootstrap the execution of the kernel-level malware. Thereby the user space vulnerability will provide the malware with control over a process that is loaded at system boot. The malware will use this control to actually trigger the kernel-level vulnerability, which loads the kernel component of the malware. So instead of targeting the kernel vulnerability directly, the malware will use a two staged loading process to place itself into the kernel.

Although a multi-staged loading process requires additional vulnerabilities to function, it can enable an attacker to overcome various obstacles. By initially leveraging vulnerabilities that are simpler to exploit, the attacker gains a platform for further exploitation. This might be useful if the second stage of the loading process requires additional information about the running system. This could be the case if the second stage must overcome ASLR, for example.

To show that data-only malware can also be resident, we created a vulnerable user space application that reads data from a configuration file. By modifying the configuration data, an attacker can trigger a vulnerability within the initialization function of the application. Since the application is loaded at system start, it can be used by our persistent rootkit for a multi-staged loading process. For this purpose, the attacker modifies the configuration file of the user space application to contain the required exploit as well as the initialization ROP chain of the rootkit. Once the system is started, the application will read the configuration data and thereby trigger the exploit. The executed ROP chain will then exploit the vulnerability within the kernel providing the initialization chain as payload. This will in-turn lead to the execution of the persistent stage within kernel space as has been described in Section IV.

While it seems unlikely that the same two required vulnerabilities exist on a multitude of heterogenous systems, we would like to stress that we present this as an academic exercise. It is possible to create a resident and persistent piece of malware that introduces no code into the system. In a practical setting, it is entirely possible that residence is achieved by introducing code into the system and the persistence is achieved through data-only means. One could imagine a rootkit that requires the introduction of a very small portion of code to load itself into the kernel and remains persistent in the kernel in a data-only manner. Such a rootkit would be very dangerous as it still able to bypass many integrity mechanisms in the kernel.

VII. Related Work

There has been much work done in the field of ROP and ROP detection (as already discussed in Section VI-C) in addition to other forms of data-only malware, such as JOP. For example, it has been shown that ROP is Turing complete [4] and multiple compilers producing ROP gadget chains have been developed [18], [31]. Further, proof of concepts showing that ROP is applicable to other architectures such as ARM and SPARC have also been presented [4], [14], [8]. In addition, other forms of data-only malware have been presented and also shown to be Turing complete [3].

Hund et al. were the first to consider using ROP to implement rootkit functionality, though they only speculated that persistence was possible [18]. This was the first work that tackled this subject. This work was followed up on by Chen et al. who present a data-only rootkit that does not make use of return gadgets but rather uses JOP [9]. They also speculated that persistence was possible, but were unable to provide a POC, as they are not able to load the gadget chain by using only a single function pointer. As the final result they suggest that for a ROP rootkit to be persistent "the gadgets must not use stack operations so that the ROP callback routine can be loaded in a fixed memory location" [9].

VIII. Conclusion

In this paper we introduced the concept of persistent data-only malware. This class of malware permanently changes the control flow of the host software without introducing a single instruction. We began our discussion of persistent data-only malware by outlining the requirements for such malware. We then proposed various methods for leveraging OS and hardware features to place arbitrary hooks in software without introducing any additional code and proposed an architecture for persistent data-only malware.

In culmination, we presented a ROP-based POC which demonstrates that persistent data-only malware is not only a theoretical threat, but can indeed be implemented and executes despite a variety of modern protection mechanisms. By testing and verifying our POC on one of the most recent Linux releases we have proven that despite all the restrictions that come with persistent data-only malware it is possible to circumvent state-of-the-art security mechanisms.

We concluded with a discussion of possible countermeasures for dealing with such a dangerous threat. Additionally, we presented the requirements for extending this work even further and constructing malware that is, not only persistent, but also resident. That is, we discussed the requirements for assuring that such malware can survive a reboot.

While there is some work in the academic field to combat data-only malware, as future work we plan to consider new methods for detecting or hindering persistent data-only malware based on the knowledge and insight that we obtained through this work.

References

[1] "Kernel patch protection: Frequently asked questions," January 2007. [Online]. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg487353.aspx

[2] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, Section B3.7.2*, ARM, July 2012.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th Symposium on Information, Computer and Communications Security*. ACM, 2011.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to risc," in *Proceedings of the 15th conference on Computer and communications security*. ACM, 2008.

[5] Bulba and Kil3r, "Bypassing stackguard and stackshield," *Phrack*, vol. 56, no. 5, January 2000. [Online]. Available: http://www.phrack.org/issues.html?issue=56&id=5

[6] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th conference on Computer and communications security*. ACM, 2009.

[7] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th conference on Computer and communications security*. ACM, 2010.

[8] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can DREs provide long-lasting security? The case of return-oriented programming and the avc advantage," in *Proceedings of the conference on Electronic voting technology/workshop on trustworthy elections*. USENIX Association, 2009.

[9] P. Chen, X. Xing, B. Mao, and L. Xie, "Return-oriented rootkit without returns (on the x86)," in *Information and Communications Security*, ser. LNCS. Springer, 2010, vol. 6476, pp. 340–354.

[10] corelanc0d3r, "Exploit writing tutorial part 11 : Heap spraying demystified," Corelan Team, Tech. Rep., December 2011. [Online]. Available: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[12] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, 1998.

[13] D. A. Dai Zovi, "Practical return-oriented programming," April 2010. [Online]. Available: http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

[14] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," Ruhr-University Bochum, Tech. Rep. HGI-TR-2010-002, 2010.

[15] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: a detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th Symposium on Information, Computer and Communications Security*. ACM, 2011.

[16] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the 1st Workshop on Secure execution of untrusted code*. ACM, 2009.

[17] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security symposium*. USENIX Association, 2012.

[18] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 2009.

[19] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2013.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 3: System Programming Guide*, Intel Corporation, March 2013.

[21] V. Katoch, "Whitepaper on bypassing aslr/dep," Secfence, Tech. Rep., September 2011. [Online]. Available: http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf

[22] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010.

[23] Nergal, "The advanced return-into-lib(c) exploits," *Phrack*, vol. 58, no. 4, December 2001. [Online]. Available: http://www.phrack.org/issues.html?issue=58&id=4

[24] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 12 2010.

[25] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 49, no. 14, November 1996. [Online]. Available: http://www.phrack.org/issues.html?issue=49&id=14

[26] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2012.

[27] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th conference on Computer and communications security*. ACM, 2007.

[28] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[29] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer, 2008.

[30] C. Schneider, J. Pfoh, and C. Eckert, "Bridging the semantic gap through static code analysis," in *Proceedings of the 5th European Workshop on System Security*. ACM, 2012.

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2011.

[32] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th conference on Computer and communications security*. ACM, 2007.

[33] A. Shishkin and I. Smit. (2012, September) Bypassing intel smep on windows 8 x64 using return-oriented programming. [Online]. Available: http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html

[34] sickness, "Linux exploit writing tutorial part 2: Atack overflow aslr bypass using ret2reg," Tech. Rep., March 2011. [Online]. Available: http://dl.packetstormsecurity.net/papers/attack/lewt2-aslrbypass.pdf

[35] Solar Designer, "Getting around non-executable stack (and fix)," Bugtraq Mailing List, Aug. 1997.

[36] *UEFI Specification*, 2nd ed., UEFI Inc., July 2013. [Online]. Available: http://www.uefi.org/specs/download/UEFI_2_4.pdf

[37] S. Vogl and C. Eckert, "Using hardware performance events for instruction-level monitoring on the x86 architecture," in *Proceedings of the 5th European Workshop on System Security*. ACM, 2012.

[38] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th conference on Computer and communications security*. ACM, 2009.

[39] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer, 2008.