Research Article

Exception-oriented programming: retrofitting code-reuse attacks to construct kernel malware

ISSN 1751-8709
Received on 2nd September 2015
Revised 8th November 2015
Accepted on 1st December 2015
doi: 10.1049/iet-ifs.2015.0372
www.ietdl.org

Liang Deng[1,2], *Qingkai Zeng*[1,2] ✉
[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, People's Republic of China*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing 210023, People's Republic of China*
✉ *E-mail: zqk@nju.edu.cn*

Abstract: Commodity operating system kernels are vulnerable to a wide range of attacks due to the large code base and broad attack surface. Mitigation mechanisms such as code signing, W⊕X, and code integrity protection have raised the bar for kernel security. In turn, attack mechanisms have also become increasingly advanced. They have evolved from simple injection of malicious code into more sophisticated code-reuse attacks [e.g. return-oriented programming (ROP)]. In this study, the authors describe exception-oriented programming (EOP), a novel code-reuse method to construct kernel malware. Unlike previous ROP that can only reuse a limited part of existing code (gadgets), EOP is able to reuse *any instruction* in existing code and chain the instructions in any order to generate malicious programmes. As a result, EOP can provide the attackers with more powerful capabilities and less complexity for building kernel malware.

1 Introduction

Over the last decade, kernel malware has increased dramatically. By exploiting vulnerabilities in operating system (OS) kernel, the malware effectively runs at the privilege layer with the ability to compromise any part of the system. Moreover, the malware can further hide its presence deep within the system, making the detection of such malware very difficult.

Traditional kernel malware generally relies on code injection that brings in new code or alters existing code to perform malicious activities. However, many protection mechanisms (such as code signing [1], W⊕X, and code integrity protection [2–7]) have gained adoption in OSs, and they are making it more difficult for attackers to inject malicious code. To bypass such protection mechanisms, more sophisticated kernel malware, named return-oriented malware (or return-oriented rootkit) [8, 9], is currently employed based on the return-oriented programming (ROP) technique [10–12].

Return-oriented malware carries out attacks by reusing the instruction sequence ending in a return instruction (or an indirect jump for jump-oriented programming (JOP) [12]). This kind of instruction sequences is known as ‘gadget’. If the control flow is diverted to a gadget, the gadget will execute the first part to perform the malicious computation and then execute a return. Since the attacker controls the stack, she can use the return to transfer the control flow to another gadget, and finally chain enough useful code gadgets to execute an arbitrary programme. While this procedure sounds straightforward, return-oriented malware still has the following limitations.

1.1 High complexity

As discussed above, the attacker generates malicious programme by choosing useful instructions from existing code. Generally speaking, if more instruction choices are provided to the attacker, it is simpler for the attacker to generate her desired programme. In ROP, however, the attacker can only use a very limited part (gadgets) of existing code, which may make the programme generation complex. For example, the attacker needs to choose instructions to read the carry flag (CF) in the *rflags* register for performing a conditional jump [10]. A simple way to achieve this is to choose the *lahf* instruction that transfers the CF flag to the *ah* register. Unfortunately, this instruction does not exist in any gadget that we found in Linux kernel code, though it exists at other positions of the code. In such a situation, the attacker has to use some other indirect ways [10, 12], which are complex and obscure, to read the CF flag.

Another cause of complexity is the side effects of the gadgets. For instance, Shacham [10] used the gadget *add (%edx), %eax; push %edi; ret* to perform an *add* operation. In this gadget, the first instruction *add (%edx), %eax* is exactly what we want. The next instruction *push %edi*, however, causes side effect because the value pushed onto the stack by it is then used by the *ret* instruction. In such situations, the attacker must additionally deal with possible side effects on registers, memory, or even control flow. Though a simple method to avoid side effects is to only consider the gadget that contains only one instruction preceding a return instruction (single-instruction gadget). However, this will reduce the number of instruction choices that the attacker can make. Moreover, in some kernels with small code base (e.g. library OS or hypervisor), the set of single-instruction gadgets may not even be Turing-complete.

1.2 Limited capability

Sophisticated kernel malware often requires the capability to execute privileged instructions to manipulate the underlying hardware, e.g. changing the root of page tables, flushing translation lookaside buffer (TLB) caches, generating interrupts, reading, or modifying hardware settings etc. For this requirement, the attacker must find either appropriate gadgets containing privileged instructions or special library functions dedicated to executing privileged instructions. Unfortunately, both of these may not exist in even commodity OS kernel code. First, those library functions are often defined as inline functions in the source code, and thus do not exist in the final binary code. Additionally, the probability of finding a useful gadget containing a desired privileged instruction is very low, since privileged instructions are relatively rare in kernel code. To demonstrate this, we conducted a simple experiment in which we searched Linux-3.2 kernel code binary (*vmlinux*) for all possible library functions and gadgets (both intended and unintended) containing privileged instructions. The *vmlinux* was compiled from unmodified Linux kernel source code with the default configuration except that we only disabled the para-virtualisation support for performance. We tested three privileged instructions (*mov-to-cr3*, *invlpg,* and *lidt*) and our results are shown in Table 1. We did not find any library functions or no-side-effect gadgets for each privileged instruction. Even if we set the maximum gadget length to five instructions to search for side-effect gadgets, the number of useful gadgets is still small. Note that the attacker has to additionally handle the side effects in this situation. Moreover, these gadgets with side effects, which rarely appear in kernel code, may disappear in an updated kernel version or in other kinds of OS kernels with smaller code base.

Kernel malware may also need to read the *gs* register which points to the base address of the process task structure and per-cpu variables. However, in the experiment described above, we did not find any appropriate library functions or no-side-effect gadgets in kernel code for reading the *gs* register.

Additionally, we observe that the general-purpose registers that can be used in return-oriented malware are also limited, since the instruction choices for the attacker are limited. Though one or two general-purpose registers are enough to perform Turing-complete computation, it is better if more register choices can be provided to the attacker. For example, the attacker can use additional registers to store temporary states and variables, which are useful for sophisticated kernel malware.

1.3 What the attacker wants

From an attacker's perspective, a better attack mechanism should provide powerful capabilities for the attacker to complete her malicious activities, while keeping attacker's manipulation simple at the same time. For a code-reuse attack mechanism, it is better if the mechanism can provide more instruction choices so that the attacker can find the most appropriate instructions and have the opportunity to reduce the complexity. It is better to provide the attacker with not only Turing-complete computational instructions but also privileged instructions to access the hardware, while freeing the attacker from annoying side effects. It is better if the mechanism can provide more general-purpose registers for computation, so that the attacker can better construct her programme.

Consequently, in this paper we present a novel code-reuse attack technique [named exception-oriented programming (EOP)] to construct kernel malware, which scores much better than ROP on all of the above requirements. Like traditional ROP techniques, EOP still constructs malicious programmes within existing benign code in the tradition of 'weird machines' [13]. However, rather than reusing just a limited part of existing code (gadgets), EOP is able to make full use of any instruction in existing code. That is to say, any single instruction can be served as machine code in the weird machine to chain together the final malicious programme. Consequently, EOP provides much more instruction choices for the attacker. For example, EOP can reuse any privileged instruction as long as the instruction exists in kernel code. EOP also frees the attacker from annoying side effects and thus the attacker can focus her attention only on her malicious activities.

We have implemented a proof of concepts (PoCs) for EOP, which installs kernel malware by exploiting a real-world vulnerability in a commodity OS (Linux). This proves that EOP is not only powerful, but also realistic to bypass code integrity protection mechanisms. We focus our attention on kernel malware in OSs (rather than user-space malware) because the implementation of EOP requires privilege-level access. EOP can be also used to carry out exploits in other privilege software such as hypervisors.

2 Related work

Table 1 Number of the gadgets and library functions for privileged instructions found in Linux-3.2

Privileged instructions	Library function	No-side-effect gadget	Side-effect gadget
mov-to-cr3	0	0	4
invlpg	0	0	2
lidt	0	0	1

In the related work, we focus on both protection and attack mechanisms in OS kernels.

2.1 Kernel code integrity protection

One of the most popular mitigations is W⊕X which takes advantage of the access rights in hardware paging to map single memory pages as either executable or writable. This prevents attackers from modifying existing code or bringing in new code for execution. To further prevent return-to-user attacks in which the attacker hijacks kernel control flow to execute code in user space, Pax [2] makes use of hardware segmentation (for x86-32) or temporarily maps user space into a different location with non-execute permissions (for x86-64). kGuard [3] proposes a lightweight compiler-based approach to counter such attacks by preventing kernel execution from crossing to user space. Recent Intel processors provide a hardware feature named supervisor mode execution protection (SMEP) [4]. With this feature, the processor will generate a fault whenever the kernel attempts to execute code residing in user space. We also see that advanced RISC machine (ARM) specifications describe a similar mechanism in their security extensions [5]. Some other systems make use of virtualisation to provide kernel code integrity guarantees. Specifically, SecVisor [6] is implemented as a tiny hypervisor that allows only approved code to execute in kernel mode and prevents such code from being modified. NICKLE [7] provides similar protections via virtualisation-based kernel code authentication. In addition, driver code signing [1] has been widely adopted to verify the code integrity of the loaded drivers in commodity OS.

2.2 Kernel control flow protection

Hooksafe [14] relies on virtualisation to protect dynamic function hooks in kernel code. Li *et al.* [15] proposed a compiler-based approach to replace the return address in the stack frame with a return index and disallow kernel rootkits from using their own return addresses to hijack kernel control flow. However, due to the complexity of kernel control flow (e.g. asynchronous interrupt handling, context switches, and signal dispatches), these protection mechanisms still leave attack surface for code injection and return-to-user attacks. KCoFI [16] is the first system to realise complete control flow integrity for commodity OS kernel by making use of the compiler-based secure virtual architecture framework. However, it incurs notable performance overhead and requires substantial kernel modifications.

2.3 Kernel data protection

To protect kernel data integrity, KOP [17] automatically maps all kernel objects with nearly complete coverage and high accuracy to enable systematic kernel data integrity checking. Sentry [18] provides a framework to partition kernel memory into separate regions with different access control permissions and thus it can efficiently prohibit illegal accesses to sensitive kernel data. To protect kernel data from untrusted kernel modules, Huko [19] and Silver [20] rely on virtualisation to run untrusted modules in a different domain from the core kernel. Some other systems [21–23] make use of monitoring or introspection to ensure kernel data integrity.

2.4 Address space layout randomisation

Address space layout randomisation (ASLR) randomises the virtual memory layout either when a new code execution starts or when the system is booted. In combination with the W⊕X property, ASLR can effectively reduce the attack surface for code-reuse attacks. In commodity OSes, Windows provides ASLR support since Vista in both user and kernel space, Linux implements it with the PaX patches [2], and MacOS ships with ASLR since version 10.5. Many researches [24–27] further provide fine-grained ASLR that randomises code at the granularity of functions, basic blocks, or even instructions. Giuffrida *et al.* [28] proposed the first system to realise fine-grained and comprehensive ASLR for OS kernels.

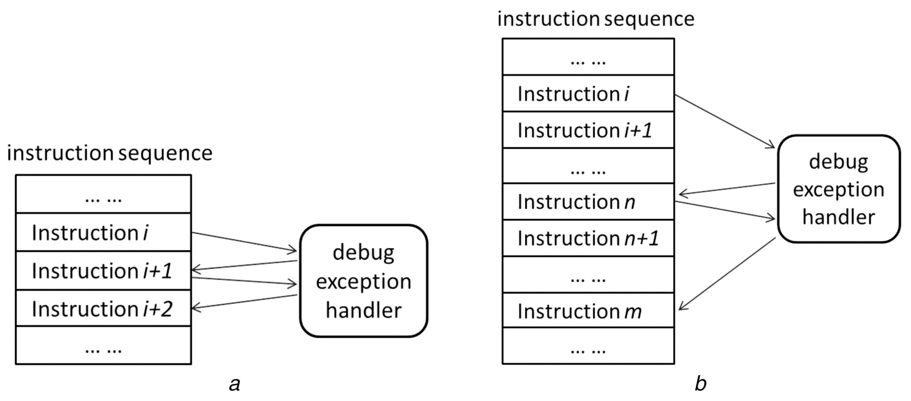

Fig. 1 *Control flows in single-step debugging and EOP*
a Control flow in single-step debugging
b Control flow in EOP

However, ASLR, even in the fine-grained implementations, is vulnerable to memory disclosure vulnerabilities [29].

2.5 Return-oriented programming

The technique of ROP was first introduced by Shacham [10] and Buchanan *et al.* [11]. Further variations on this include JOP which uses indirect jumps to chain the gadgets [12]. It has been shown that such methods are Turing-complete in many situations [8, 10, 12]. To bypass kernel code integrity protection, ROP is also used as an effective attack method in kernel space for installing return-oriented malware [8, 9]. SROP [30] leverages the *sigreturn* system call to realise ROP and needs only a single gadget for the exploit. However, SROP can only be used in user space, but cannot be used to construct kernel malware.

In the remainder of this paper, we will describe the overall design and the PoCs of EOP. For a better understanding, we start with the background: debug facilities in x86.

3 Background: debug facilities in x86

In x86, software can set the trap flag (TF) in the *rflags* register to enable single-step debugging [4]. When the processor detects that the TF flag is set, it will generate a debug exception after each instruction is executed. Then the control flow is transferred to the debug exception handler predefined by system software. The processor temporarily clears the TF flag during the execution of the exception handler, so that a nested debug exception will not occur. Software can execute the instructions such as *popf* and *iret* to set the TF flag in the *rflags* register. However, the debug exception will not immediately occur after the instruction that sets the TF flag. For instance, if an *iret* instruction is executed to set the TF flag, the processor will not generate this exception until after the instruction that follows the *iret* instruction.

4 Exception-oriented programming

4.1 High-level idea

The overall design of EOP is based on the single-step debugging facility provided by x86 processors. Fig. 1*a* shows the control flow when single-step debugging is enabled. The processor executes the instruction sequence (e.g. from instruction i to instruction $i + 2$) step by step. It generates a debug exception at each instruction boundary, pushes the interrupt stack frame (containing the saved *rip*, *cs*, *rflags*, *rsp*, and *ss*) on the stack, and transfers the control flow to the debug exception handler. Hereafter, we refer to the interrupt stack frame as ISF for simplicity. At the end of the exception handler, the processor executes the *iret* instruction to pop the ISF on the stack and transfer the control flow to the next instruction. The original use of single-step debugging is to examine the state of the programme and related data before and after the execution of each instruction.

Our key insight is that if the attacker can control the ISF (e.g. the saved *rip*) popped in the exception handler, she can divert the control flow to an arbitrary instruction (rather than the next instruction) each time the procedure is returned from the exception handler. In this way, the attacker can chain a set of instructions together to execute the programme as she desires. Fig. 1*b* gives an example to illustrate this mechanism. After Instruction i is executed, a debug exception occurs for single-step debugging. The processor pushes the ISF on the stack and transfers the control flow to the debug exception handler. The ISF's *rip* field points to the next instruction (Instruction $i + 1$), so that the control flow will be transferred to Instruction $i + 1$ when the procedure is returned from the exception handler. However, if the attacker manipulates the *rsp* register to point to an attacker-controlled ISF, she can transfer the control flow to an arbitrary instruction (e.g. Instruction n in this figure). In the same manner, the attacker can then transfer the control flow to other instructions and finally chain these instructions together to perform any computation.

Traditional ROP (or JOP) techniques make use of return instructions (or indirect jumps) to divert the control flow and chain the gadgets together. Thus, they can only reuse the instruction sequence ending in a return instruction (or an indirect jump). However, EOP leverages debug exceptions triggered at the end of the instructions to transfer the control flow, and thus can reuse any instruction in existing code. In the following, we will describe the overall design of EOP in detail.

4.2 Preconditions

For the exploitation to be successful, certain preconditions have to be fulfilled. EOP shares the same preconditions as previous kernel-level ROP attacks [8, 9]:

(i) The attacker should have control over the instruction pointer (the *rip* register).
(ii) The attacker should have control over the stack pointer (the *rsp* register).

Hund *et al.* [8] have shown how to make use of a buffer overflow vulnerability in kernel code to satisfy the above two preconditions. Specifically, the attacker exploits the vulnerability and overwrites the return address on the stack to control the *rip* register. The return address is set to point to a *pop %rsp; ret* gadget. Thus, by further overwriting 8 bytes following the return address, the attacker can use this gadget to control the *rsp* register.

4.3 Chaining instructions

As discussed in Section 4.1, the key of chaining arbitrary instructions together is to set the *rsp* register to point to an attacker-controlled ISF in the debug exception handler. This is not a problem as the attacker has control over the *rsp* register. In the following, we will use the same example present in Fig. 1*b* to illustrate how to chain Instruction i, Instruction n, and Instruction m together using EOP.

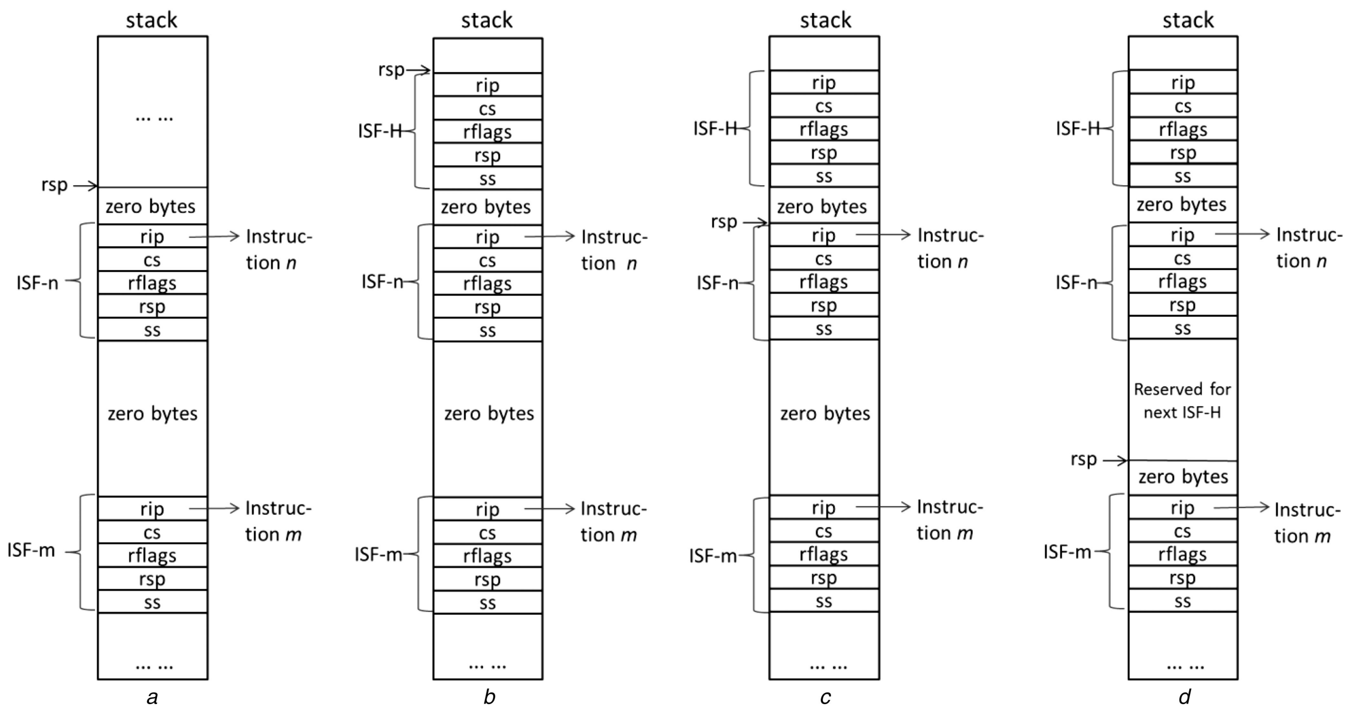

Fig. 2 *Stack layout for chaining Instructions*
a Stack layout when Instruction *i* is executed
b Stack layout after the processor pushes the ISF-H
c Stack layout after the *add 0 × 50, %rsp* instruction is executed
d Stack layout after the *iret* instruction is executed

Since the attacker controls the *rsp* register, we assume that the *rsp* register has pointed to a manipulated stack (as shown in Fig. 2*a*) when Instruction *i* is executed. There are two attacker-controlled ISFs (named ISF-*n* and ISF-*m*) on the manipulated stack. The *rip* field of ISF-*n* points to Instruction *n* and the *rip* field of ISF-*m* points to Instruction *m*. After Instruction *i* is executed, the processor generates a debug exception and pushes a hardware-generated ISF (ISF-H) on the stack before invoking the exception handler (Fig. 2*b*). The next step is to change the *rsp* register to point to the attacker-controlled ISF-*n*. To achieve this, the attacker only needs to install a new exception handler which points to the instruction sequence *add 0 × 50, %rsp; iret*. In this way, the new exception handler first executes the *add 0 × 50, %rsp* instruction to increase the *rsp* register, so that the *rsp* register can point right to ISF-*n* with some 0 bytes inserted on the stack (as shown in Fig. 2*c*). Then the exception handler executes the *iret* instruction to pop ISF-*n* and transfers the control flow to Instruction *n*. At the same time, the *iret* instruction also modifies the *rsp* register according to the *rsp* field of ISF-*n*. We carefully set the *rsp* field of ISF-*n*, so that the *iret* instruction will set the *rsp* register to reserve space on the stack for the next ISF-H (as shown in Fig. 2*d*). After Instruction *n* is executed, the processor pushes ISF-H onto the reserved space and invokes the debug exception handler again. Similarly, the debug exception handler then pops ISF-*m* and finally transfers the control flow to Instruction *m*. Note that it is necessary to reserve space for ISF-H, because otherwise ISF-*n* on the stack will be overwritten by the ISF-H and cannot be reused again.

The instruction sequence *add 0 × 50, %rsp; iret* is easy to find in Linux kernel code. In our test, this instruction sequence exists in the code of all Linux-3.x kernel versions (from Linux-3.0 to Linux-3.19). Though this sequence has been removed in the latest Linux-4.x kernel versions, we can still use other instruction sequences as substitutes. For example, in all Linux-4.x kernel versions (from Linux-4.0 to Linux-4.2), it is easy to find the return-oriented gadget such as *add 0 × 38, %rsp; ret*, which can be also used as the debug exception handler to chain instructions. Specifically, this gadget first increases the *rsp* register to bypass ISF-H and then executes the *ret* instruction to transfer the control flow to an *iret* instruction (with an attacker-controlled return address on the stack). When the *iret* instruction gets control, it will pop the attacker-controlled ISF on the stack to realise instruction chaining. In our test, these kinds of gadgets is also easy to find in the code of other privilege software (e.g. all supported versions of Xen [31]).

4.4 Chaining functions

Besides chaining instructions, we can also use the debug exception to chain functions in kernel code. For performance, when the control flow is transferred to a function being chained, we should disable single-step debugging during the execution of the function. After the function is executed, we should re-enable single-step debugging again for chaining other instructions. It is also necessary to execute the function with the *rsp* register set to a separated stack, because otherwise the function might trash the ISFs on the stack that we intend to reuse again.

In the following, we give an example that chains Instruction *i*, Function *n*, and Instruction *m* together to illustrate our mechanism. Fig. 3*a* shows the stack layout after Instruction *i* is executed. As discussed in Section 4.3, the processor generates a debug exception and the *iret* instruction in the exception handler then pops the attacker-controlled ISF (*ISF-Fn* in this figure) on the stack and transfers the control flow to the entry point of Function *n*. At the same time, the *iret* instruction disables single-step debugging, since the TF flag in the *rflags* field of ISF-Fn is cleared. The *iret* instruction also modifies the *rsp* register to point to the separated stack for the function, since the *rsp* field of ISF-Fn is set to it. Then, the processor executes the function on the separated stack as traditional. The layout of the separated stack is shown in Fig. 3*b*. When the processor finishes the execution of the function and executes a *ret* instruction to perform a function return, it will pop the manipulated return address (*return addr* in Fig. 3*b*) which points to an *iret* instruction. Then the *iret* instruction gets control and pops the manipulated ISF-*m* on the stack. Thus the control flow is then transferred to Instruction *m*. Meanwhile, the TF flag in the *rflags* register is set again to enable single-step debugging and the *rsp* register is also modified to point to the original stack.

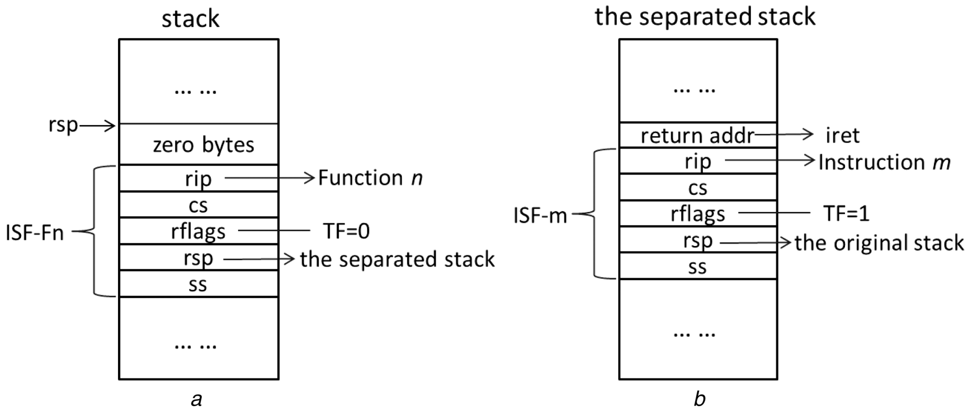

Fig. 3 *Stack layout for chaining function calls*
a Stack layout after Instruction *i* is executed
b Layout of the separate stack

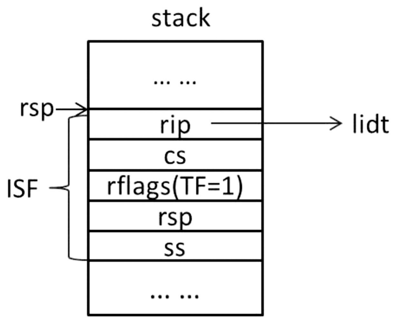

Fig. 4 *Stack layout for changing the debug exception handler*

4.5 Chaining traditional ROP gadgets

EOP can be also used to chain traditional ROP gadgets. The attacker can set the next ISF to point to a traditional ROP gadget if she desires to chain it. At the same time, the TF flag in the *rflags* field of the ISF should be cleared to disable single-step debugging. Then the attacker is free to chain ROP gadgets using the sequence of gadget addresses on the stack, as with traditional ROP attacks. To re-enable EOP, the attacker only needs to use the return to transfer the control flow to an *iret* instruction which enables single-step debugging again and starts chaining instructions.

4.6 Changing the debug exception handler

Before chaining instructions and functions, the attacker should first change the debug exception handler as discussed above. A straightforward method is to use traditional ROP (note that EOP cannot be used at this moment) to modify the corresponding entry in the interrupt descriptor table (IDT). Unfortunately, the IDT is mapped as read-only since Linux-3.10, and using ROP is also complicated. Another method is to load a new attacker-controlled IDT by executing the *lidt* instruction. However, the ROP gadgets or library functions for executing the *lidt* instruction may not be easy to find in kernel code, as discussed in Section 1.

To address this, we propose a novel approach to load an attacker-controlled IDT, which only requires a single *lidt* instruction wherever it is in kernel code. Specifically, we first execute an *iret* instruction to perform an interrupt return with a manipulated ISF on the stack, so that we can set both of the *rflags* register and the *rip* register at the same time. The manipulated ISF is shown in Fig. 4. Its *rip* field is set to point to the *lidt* instruction, and the TF flag is set in the *rflags* field. Thus, after the *iret* instruction is executed, the processor will transfer the control flow to the *lidt* instruction and enable single-step debugging at the same time. Then the processor executes the *lidt* instruction to load an attacker-controlled IDT. Note that the debug exception will not occur until after the *lidt* instruction (the instruction that follows the *iret* instruction) as discussed in Section 3. That is to say, when the debug exception occurs, the debug exception handler has been properly modified to be *add 0 × 50, %rsp; iret*. As a result, from then on we can use the new debug exception handler to realise instruction chaining as described in Section 4.3.

4.7 Turing completeness

To demonstrate the Turing completeness of EOP, we have successfully created Turing-complete instruction sets for both Linux kernel code (Linux-3.2, Linux-3.8, and Linux-4.0) and Xen hypervisor code (Xen-3.4.4, Xen-4.2.5, and Xen-4.5.1). In each Turing-complete set, load/store and arithmetic operations can be simply realised by choosing corresponding instructions in kernel code since EOP can reuse each single instruction. Unconditional jump is realised by using the instructions that can change the value of the *rsp* register, e.g. *pop %rsp* or the combination of *pop %rdi* and *mov %rdi, %rsp*.

However, conducting conditional jump needs some additional efforts. To conduct a conditional jump, we should first acquire the condition flag in the *rflags* register, and then change the *rsp* register (e.g. by adding *rsp_delta*) to complete the jump based on the condition. In EOP, the condition flag in the *rflags* register is easy to acquire, because the processor always pushes the current value of the *rflags* register onto the stack (in ISF-H) after each instruction is executed. Thus we only need to chain a load instruction to read the condition flag from the stack. After acquiring the condition flag whose value is 0 or 1, the next step is to transform it to be either 0 or *rsp_delta*, so that we can simply add this value to the *rsp* register to complete the conditional jump. Similar to the original ROP work [10], we further chain a *neg* instruction and a *bitwise and* instruction to realise this transformation.

With the conditional jump, loops can also be created by conditionally modifying the *rsp* register to point to a previous ISF, so that the control flow will be repeatedly diverted to the instruction pointed by this ISF if the loop condition is fulfilled.

5 Proof of concepts

5.1 Attack model

We assume a local attacker that has user-level access. Further we assume a vulnerability in kernel code which allows the attacker to start EOP attacks. In our current implementation, we used the real Linux kernel vulnerability CVE-2013-209410 which has been also used to trigger traditional ROP attacks [9]. In addition, we used a standard installation of a 64 bit Ubuntu 13.04, in which the Linux kernel is protected by the W⊕X property and SMEP mechanism.

5.2 Implementation

Within the attack model described above, we make use of EOP to construct the kernel malware which realises some real-world rootkit functionalities such as process hiding and kernel module hiding. Our implementation includes three stages (initialisation

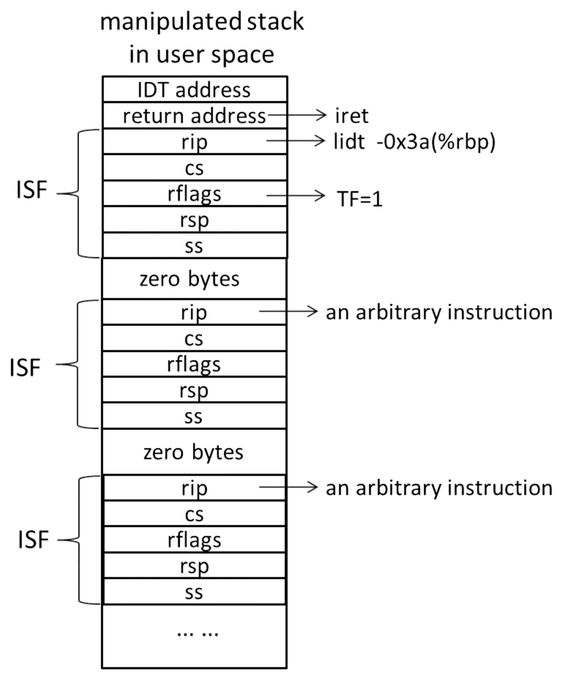

Fig. 5 *Layout of the manipulated user-space stack*

stage, exploit stage, and EOP stage) as we will detail in the following.

5.2.1 Initialisation stage: In this phase, we execute a user application to prepare a manipulated stack and a manipulated IDT in user space. In the manipulated IDT, the entry of the debug exception handler is set to point to the instruction sequence *add 0 × 50, %rsp; iret* in kernel code. Other entries are the same as those of the original IDT. Though the contents of IDT entries are kernel specific, it is not a problem for the attacker to acquire them.

5.2.2 Exploit stage: After the manipulated stack and IDT are prepared, the user application then exploits the kernel vulnerability CVE-2013-209410 which allows the attacker to control both the *rip* register and the *rsp* register in kernel space. The details of this exploit process are well discussed in the previous work [9], and will not be repeated due to the limited paper space.

In our PoC, this vulnerability is used to divert the kernel control flow to a gadget *pop %rbp; ret*, and to modify the *rsp* register to point to our manipulated user-space stack (shown in Fig. 5). The gadget *pop %rbp; ret* is used to transfer the address of our user-space IDT to the *rbp* register. Specifically, the *pop %rbp* instruction pops the next 8 bytes on the stack (*IDT address* in Fig. 5) to the *rbp* register; and then the *ret* instruction pops the return address on the stack and transfers the control flow to an *iret* instruction.

This *iret* instruction, as discussed in Section 4.6, is used to enable single-step debugging and transfers the control to a *lidt* instruction [*lidt -0 × 3a(%rbp)*] that we found in kernel code. Then, the *lidt -0 × 3a(%rbp)* instruction is executed to load our user-space IDT from the address pointed by the *rbp* register. After the *lidt -0 × 3a(%rbp)* instruction is executed, the debug exception will be continuously triggered to invoke the new debug exception handler which pops the manipulated ISFs on the stack and chains the instructions pointed by these ISFs.

5.2.3 EOP stage: Once the exploit stage is completed, we can use the sequence of ISFs on the user-space stack to chain arbitrary instructions and functions to perform any computation we desire. In this stage, we use EOP to realise two rootkit functionalities: process hiding and kernel module hiding. Process hiding is realised by setting the process identifier (PID) of the process that should be hidden to zero. In this way, the information of the process will no longer be displayed by the process listing tools such as *ps*. We additionally chain instructions to build a simple communication channel for acquiring the attacker's hiding request (e.g. the PID for the process to be hidden) from user space. Kernel module hiding is realised by removing the module object that should be hidden from the module list. As a result, the module will no longer be displayed by the *lsmod* programme. After the malicious activities are completed, we restore the original IDT by executing the single *lidt* instruction again and disable single-step debugging.

6 Discussion

6.1 Comparison with traditional ROP

In this section, we give a systematic comparison of key properties with traditional ROP to give insights on the advantages and limitations of EOP.

6.1.1 Code-reuse rate: Compared with traditional ROP, EOP can reuse every existing instruction or function in kernel code, and does not require those instruction sequences ending with returns to be useful. With this much higher code-reuse rate, EOP attackers can have a flexible choice of existing instructions and can realise some functionalities that traditional ROP may not be able to provide (e.g. executing some privileged instructions).

6.1.2 Side effects: Compared with traditional ROP, EOP makes use of a customised debug exception handler to chain each instruction together. There is no need to deal with the side effects. Therefore, EOP attackers can focus their attention only on their malicious activities.

6.1.3 Application scenarios: Since EOP requires the privilege to trigger single-step debugging and modify the debug exception handler, it can be only used in kernel space to construct code-reuse kernel malware. In addition, it also relies heavily on the debugging facilities provided by x86. However, traditional ROP is a more generic attack method, which in principle can be used in both user space and kernel space, and in different hardware platforms.

6.1.4 Payloads: The payloads that control the execution of the ‘weird machines’ are different in EOP and traditional ROP. EOP uses a sequence of crafted ISFs on the stack to control the execution order of the instructions, while ROP needs only a sequence of gadget addresses on the stack. From this point of view, it might be a bit more complicated for EOP attackers to install their desired payloads. On the other hand, using ISFs to control the execution gives EOP attackers more flexibilities. For example, the attackers can use the *rsp* field in the crafted ISF to modify the stack register at the same time.

6.2 EOP in x86-32

We have also applied EOP to x86-32. There are two main differences compared with x86-64. First, the layout of the ISF for intra-privilege interrupt return is different in x86-32. Second, in x86-32, function arguments are passed through stack, which will overlap with the next ISF. To avoid overlapping, we simply relocate the next ISF to the place that follows the function arguments on the stack. When the processor finishes function execution and performs a function return, it will pop the manipulated return address that points to a gadget *add imm32,%esp; ret*. This gadget is used to increase the *esp* register to bypass the function arguments on the stack and point right to the next ISF. Then the gadget performs a return to transfer the control flow to an *iret* instruction. Finally, the *iret* instruction gets control and pops the next ISF to chain the next instruction.

6.3 EOP in other privilege software

Though we only use EOP to implement kernel malware in our PoCs, EOP can be also used in other privilege software.

6.3.1 EOP in hypervisors: Compared with the previous ROP, the implementation of EOP requires privilege-level access. By exploiting vulnerabilities in a hypervisor such as Xen [31], the attacker can also acquire privilege-level access and thus can use EOP to install malware in the hypervisor. It is worth noting that EOP is more useful in such a situation, because the code base of a hypervisor is usually much smaller than an OS kernel's. EOP can provide more instruction choices to the attacker within such a small code base.

6.3.2 EOP in virtual machines: When the attacker exploits an OS kernel running in a virtual machine (VM), she is still able to trigger single-step debugging in the VM and execute the *lidt* instruction to modify the debug exception handler used in the VM, though some intercepts and virtualisation may be performed by the underlying hypervisor. As a result, EOP still works even if the OS kernel runs in a VM. However, in a para-virtualised VM, some modifications to the exploitation process are required. For example, we should substitute the *lidt* instruction with its corresponding para-virtualised interface.

6.4 Countermeasures

As with ROP, an EOP relies on the existing instructions located at known addresses and needs to alter the original kernel control flow. Thus previous mitigation mechanisms for ROP such as ASLR and control flow integrity are also useful to raise the bar of EOP attacks. However, EOP has its own unique characteristics. It relies on the debug exception to alter the control flow and needs to change the debug exception handler. As a result, mitigation mechanisms for EOP can take these features into account. For example, we can make use of virtualisation to prevent the attacker from modifying existing IDT or loading a new IDT after system initialisation, or intercept the debug exception to detect the violations of kernel control flow.

7 Conclusion

In this paper, we introduced a novel code-reuse approach named EOP for the attacker to construct kernel malware. We began our discussion by detailing the limitations of existing return-oriented malware and then introduced the overall design of EOP and finally presented the PoCs for a Linux kernel exploit. We showed that EOP can provide the attackers with more powerful capabilities and less complexity than previous ROP for building kernel malware.

8 Acknowledgments

This work has been partly supported by the National Science Foundation of China under grant no. 61170070, 61572248, 61431008, and 61321491; the National Key Technology R&D Program of China under grant no. 2012BAK26B01; and The programme B for Outstanding Ph.D. candidate of the Nanjing University.

9 References

[1] 'Driver Signing Requirements for Windows'. Available at https://www.msdn.microsoft.com/en-US/library/windows/hardware-/dn653563, accessed June 2015

[2] 'Homepage of the PaX team'. Available at http://www.pax.grsecurity.net, accessed June 2015

[3] Kemerlis, V.P., Portokalidis, G., Keromytis, A.D.: 'kGuard: lightweight kernel protection against return-to-user attacks'. Proc. Int. Conf. USENIX Conf. on Security Symp., 2012

[4] Intel: 'Intel® 64 and IA-32 Architectures Software Developer's Manual'. June 2015

[5] ARM: 'ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition'. July 2012

[6] Seshadri, A., Luk, M., Qu, N., *et al.*: 'SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes'. Proc. Int. Conf. ACM SIGOPS Symp. on Operating Systems Principles, 2007

[7] Riley, R., Jiang, X., Xu, D.: 'Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing'. Proc. Int. Conf. Int. Symp. on Recent Advances in Intrusion Detection, 2008

[8] Hund, R., Holz, T., Freiling, F.C.: 'Return-oriented rootkits: bypassing kernel code integrity protection mechanisms'. Proc. Int. Conf. USENIX Security Symp., 2009

[9] Vogl, S., Pfoh, J., Kittel, T., *et al.*: 'Persistent data-only malware: function hooks without code'. Proc. Int. Conf. Annual Network & Distributed System Security Symp., 2014

[10] Shacham, H.: 'The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)'. Proc. Int. Conf. ACM Conf. on Computer and Communications Security, 2007

[11] Buchanan, E., Roemer, R., Shacham, H., *et al.*: 'When good instructions go bad: generalizing return-oriented programming to RISC'. Int. Conf. ACM Conf. on Computer and Communications Security, 2008

[12] Checkoway, S., Davi, L., Dmitrienko, A., *et al.*: 'Return-oriented programming without returns'. Int. Conf. ACM Conf. on Computer and Communications Security, 2010

[13] Dullien, T.: 'Exploitation and state machines: programming the weird machine, revisited'. Proc. Int. Conf. Infiltrate Conf., 2011

[14] Wang, Z., Jiang, X., Cui, W., *et al.*: 'Countering kernel rootkits with lightweight hook protection'. Proc. Int. Conf. ACM Conf. on Computer and Communications Security, 2009

[15] Li, J., Wang, Z., Jiang, X., *et al.*: 'Defeating return-oriented rootkits with return-less kernels'. Proc. Int. Conf. European Conf. on Computer Systems, 2010

[16] Criswell, J., Dautenhahn, N., Adve, V.: 'KCoFI: complete control-flow integrity for commodity operating system kernels'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2014

[17] Carbone, M., Cui, W., Lu, L., *et al.*: 'Mapping kernel objects to enable systematic integrity checking'. Proc. Int. Conf. ACM Conf. on Computer and Communications Security, 2009

[18] Srivastava, A., Giffin, J.: 'Efficient protection of kernel data structures via object partitioning'. Proc. Int. Conf. Annual Computer Security Applications Conf., 2012

[19] Xiong, X., Tian, D., Liu, P.: 'Practical protection of kernel integrity for commodity OS from untrusted extensions'. Proc. Int. Conf. Network and Distributed System Security Symp., 2011

[20] Xiong, X., Liu, P.: 'SILVER: fine-grained and transparent protection domain primitives in commodity OS kernel'. Proc. Int. Conf. Symp. on Research in Attacks, Intrusions, and Defenses, 2013

[21] Fu, Y., Lin, Z.: 'Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2012

[22] Zhang, F., Zhang, K., Sun, K., *et al.*: 'SPECTRE: a dependable introspection framework via system management mode'. Proc. Int. Conf. Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks, 2013

[23] Lee, H., Moon, H., Jang, D., *et al.*: 'Ki-mon: a hardware-assisted event triggered monitoring platform for mutable kernel object'. Proc. Int. Conf. Usenix Security Symp., 2013

[24] Hiser, J., Nguyen-Tuong, A., Co, M., *et al.*: 'ILR: where'd my gadgets go?'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2012

[25] Pappas, V., Polychronakis, M., Keromytis, A.D.: 'Smashing the gadgets: hindering return-oriented programming using in-place code randomization'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2012

[26] Wartell, R., Mohan, V., Hamlen, K.W., *et al.*: 'Binary stirring: self-randomizing instruction addresses of legacy x86 binary code'. Proc. Int. Conf. ACM Conf. on Computer and Communications Security, 2012

[27] Backes, M., Nurnberger, S.: 'Oxymoron: making fine-grained memory randomization practical by allowing code sharing'. Proc. Int. Conf. USENIX Security Symp., 2014

[28] Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: 'Enhanced operating system security through efficient and fine-grained address space randomization'. Proc. Int. Conf. USENIX Security Symp., 2012

[29] Snow, K.Z., Monrose, F., Davi, L., *et al.*: 'Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2013

[30] Bosman, E., Bos, H.: 'Framing signals – a return to portable shellcode'. Proc. Int. Conf. IEEE Symp. on Security and Privacy, 2014

[31] 'Xen project'. Available at http://www.xenproject.org/, accessed June 2015