Front. Comput. Sci., 2022, 16(4): 164818
https://doi.org/10.1007/s11704-021-0342-8

RESEARCH ARTICLE

SeBROP: blind ROP attacks without returns

Tianning ZHANG (✉)[1], Miao CAI[2], Diming ZHANG[3], Hao HUANG[1]

1 Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China
2 School of Computer and Information, Hohai University, Nanjing 211106, China
3 College of Computer Engineering, Jiangsu University of Science and Technology, Zhenjiang 212008, China

Abstract Currently, security-critical server programs are well protected by various defense techniques, such as Address Space Layout Randomization(ASLR), eXecute Only Memory(XOM), and Data Execution Prevention(DEP), against modern code-reuse attacks like Return-oriented Programming(ROP) attacks. Moreover, in these victim programs, most syscall instructions lack the following *ret* instructions, which prevents attacks to stitch multiple system calls to implement advanced behaviors like launching a remote shell. Lacking this kind of gadget greatly constrains the capability of code-reuse attacks.

This paper proposes a novel code-reuse attack method called Signal Enhanced Blind Return Oriented Programming (SeBROP) to address these challenges. Our SeBROP can initiate a successful exploit to server-side programs using only a stack overflow vulnerability. By leveraging a side-channel that exists in the victim program, we show how to find a variety of gadgets blindly without any pre-knowledges or reading/disassembling the code segment. Then, we propose a technique that exploits the current vulnerable signal checking mechanism to realize the execution flow control even when *ret* instructions are absent. Our technique can stitch a number of system calls without returns, which is more superior to conventional ROP attacks. Finally, the SeBROP attack precisely identifies many useful gadgets to constitute a Turing-complete set. SeBROP attack can defeat almost all state-of-the-art defense techniques. The SeBROP attack is compatible with both modern 64-bit and 32-bit systems.

To validate its effectiveness, We craft three exploits of the SeBROP attack for three real-world applications, i.e., 32-bit Apache 1.3.49, 32-bit ProFTPD 1.3.0, and 64-bit Nginx 1.4.0. Experimental results demonstrate that the SeBROP attack can successfully spawn a remote shell on Nginx, ProFTPD, and Apache with less than 8500/4300/2100 requests, respectively.

Keywords code-reuse attack, ROP, signal

1 Introduction

The code-reuse attack reuses the code snippets in the program to implement unintended behaviors. The Return-oriented Programming (ROP) attack [1] is a kind of code-reuse attack. In the ROP attack, the code snippet consists of basic instructions followed by a return instruction, which are called gadgets. The return instructions play a crucial role in the ROP attack because a normal ROP attack requires them to connect useful instructions distributed in the victim program. The ROP attack poses great security threats to programs. Many defense techniques, such as Address Space Layout Randomization (ASLR) [2] and eXecute Only Memory(XOM) [3], are proposed to defend against ROP attacks. These techniques prevent the ROP attack by providing strong protection for the memory location of the code segment. By deploying these protections, it is difficult for adversaries to obtain the exact memory locations of gadgets. Another prerequisite of this paper is the program source or binary code is unknown [4]. It further increases the difficulty because existing ROP attacks require the code for either binary analysis or source code scrutiny.

Moreover, according to our survey in Section 2, most victim programs lack an important kind of gadget, i.e., syscall gadget `syscall; ret`. This kind of gadget has a *syscall* instruction followed by a *ret* instruction. It enables the attacker to execute a system call and return back to the remaining ROP chain on the stack. The syscall gadget is critical in ROP attacks because attacks require it to implement advanced behaviors, such as executing malicious code to launch a remote shell.

Previous attacks usually search libraries for syscall gadgets. However, current software systems are protected by strong memory-protection methods, such as address randomization and encryption mechanisms (e.g., ASLR [2] and ASLR-Guard [5]). Therefore, it is difficult to search for these gadgets because the memory-mapping addresses of these libraries are unknown. To overcome this issue, Sigreturn Oriented Programming (SROP) [6] searches the vsyscall page for the syscall gadget. Blind Return Oriented Programming (BROP) attack [4] directly uses the functions (e.g., *dup*, *exec*) provided by the Procedure Linkage Table (PLT). However, these approaches also have limitations. The vsyscall utilized in the SROP method is now emulated in the newer kernel version and thus the vsyscall code is protected by the high-privileged OS kernel. In addition, the required attack functions in the PLT are missing in the 32-bit system.

Received July 14, 2020; accepted January 20, 2021

E-mail: zhangtianning128@126.com

To address aforementioned challenges, this paper proposes a new attack method named Signal Enhanced Blind Return Oriented Programming (SeBROP). Exploiting a stack buffer overflow vulnerability in server-side programs, the SeBROP attack can gather a set of gadgets that satisfy Turing-completeness and initiates a dangerous code-reuse attack without reading or disassembling the code segment. SeBROP attack finds all kinds of gadgets by blind execution. Our experimental result manifests that each kind of gadget has its unique runtime characteristics, which can be leveraged to differentiate them from each other. As opposed to prior work, SeBROP attack need not read or disassemble the program code segment, so it can bypass most existing protections, e.g., stack canary [7], Data Execution Prevention(DEP), ASLR [2], fine-grained ASLR [8], and XOM [3], Readactor [9] and Readactor++ [10]. In the SeBROP attack, we even do not have to possess the victim program source or binary code beforehand. All the code details are inferred by execution. SeBROP attack is also compatible with both 32-bit and 64-bit systems.

In our SeBROP attack, we exploit a side-channel in sever-side programs. This side-channel is first discovered by the BROP work [4]. Nevertheless, we show this side-channel is more powerful than they thought. In SeBROP, we carefully divert the control flow by overwriting the return address with candidate gadgets. Furthermore, we propose a variety of techniques to identify all details of the gadget. We propose an efficient algorithm to precisely differentiate every instruction operator. Besides, we also infer instruction operands by carefully analyzing the memory vestige. In summary, our SeBROP attack can blindly collect memory load/store gadgets, arithmetic operation gadgets, logical operation gadgets, and control-flow gadgets. Gathering a Turing-compete set of gadgets is necessary because the current XOM technique [3] can thwart the BROP attack by prohibiting it from dumping the binary from a remote server.

In addition, the *syscall* instruction widely exists in libraries and applications. However, most *syscall* instructions miss a subsequent *ret* instruction. As a result, it is infeasible to construct a gadget chain without necessary return instructions. SeBROP provides an alternative solution to realize execution flow control based on the vulnerable signal mechanism. In the commodity operating system kernel, when the OS kernel finishes executing a system call routine, it would check pending signals. If there exists, it will execute the signal handler and then return to the user-space. Thus, a pending signal during system call execution can divert the normal execution flow. As a consequence, signal checking is vulnerable. We design a crafted signal handler to carefully divert the control flow. This signal handler functions as a trampoline to skip the *sigreturn* function and signal frame and redirect the control flow to gadgets that are pre-set on the stack.

Note that the timing of the signal arrival is critical. If the signal arrives before or after system call execution, the control flow hijacking can not succeed. We provide two solutions to solve this challenge. First, we take advantage of a shared flag to synchronize the signal sending and receiving process. The second method does not require another process to send the signal. It registers a signal handler for *SIGSEGV* signal, which is sent by the OS kernel when illegal memory access occurs. When the program executes the instructions after the *syscall* instruction, it will encounter illegal memory access in most cases. Therefore, we can hijack the control flow as the first solution.

More generally, we can repeat this process to stitch a sequence of system calls without any return instructions. As we survey in current software libraries and applications, it is extremely difficult to find a useful syscall gadget. Therefore, our approach advances the existing code-reuse attack because it eliminates a prerequisite in ROP attacks.

To evaluate the effectiveness and efficiency of the proposed SeBROP attack. we conduct three practical attacks using three real-world applications, i.e., Apache 1.3.49, ProFTPD 1.3.0, and Nginx 1.4.0. Exploiting a stack overflow vulnerability, we identify a Turing-complete set of gadgets remotely without any pre-knowledges of victim programs. We successfully implement a chain of syscall gadgets based on our signal hijacking techniques. Putting it all together, we craft three exploits to launch the remote shell. All of these attacks can be completed within 45 minutes.

This paper makes the following contributions:

- We analyze existing ROP attack variants from both adversary and victim aspects. We conclude that missing syscall gadgets prevents current ROP attack evolution.
- We propose a novel code-reuse attack method called SeBROP, which blindly finds a Turing-complete set of gadgets without any pre-knowledge of the victim program or reading/disassembling the code segment. SeBROP greatly enhances the ROP attack by removing the constraint of return instructions.
- We conduct three practical exploits of the SeBROP attack with stack vulnerabilities for three server programs. Experimental results suggest that the SeBROP attack can defeat state-of-the-arts defense techniques.

The rest of the paper is organized as follows. Section 2 provides the necessary background and related works. Section 3 specifies the assumption of the attack environment and gives an attack outline. Section 4 describes SeBROP attack in detail. Section 5 presents the evaluation results. Section 6 mainly demonstrates the generality and limitations of our attack. Section 7 discusses possible defenses to mitigate our SeBROP attack. Section 8 summarizes our paper.

2 Background and related work

2.1 ROP and its variants

ROP attack reuses the code snippets in the victim program to implement malicious behaviors. In the following section, we describe two variants of ROP attacks. We also provide a detailed comparison of these attack methods from different technical dimensions, as shown in Table 1.

BROP The BROP attack [4] utilizes a blind execution method to find enough gadgets without reading the code segment. The BROP attack first blindly identifies a specific type of gadget (i.e., `pop; ret` gadgets) and then locates the

Table 1 Comparison of four ROP attacks. Syscall gadget means the gadget in the form of < `syscall; ret` > or < `int 0x80; ret` >. × means the attack can bypass the protection. √ means that the protection can defend the attack

Attack	Precondition		Environment			Protections			
	Dump code	Syscall gadget	32-bit	64-bit	JIT compiler	DEP	Fine-grained ASLR	XOM	Readactor
BROP [4]	Yes	No	No	Yes	No	×	×	√	√
SROP [6]	Yes	Yes	No	Yes	No	×	√	√	√
JIT-ROP [11]	Yes	Yes	No	No	Yes	×	×	√	√
SeBROP	**No**	**No**	**Yes**	**Yes**	**No**	×	×	×	×

PLT table. It exploits the `pop; ret` gadgets and functions in the PLT table to implement a *write* system call to dump the code segment from the remote server. Finally, the BROP attack disassembles the code segment to search for more useful gadgets. The BROP attack can bypass many system protection mechanisms, such as stack canary, DEP, and fine-grained ASLR.

Unfortunately, BROP is unable to defeat the XOM protection because the XOM technique restricts the read permission for the code segment. In addition, the BROP attack is merely applicable to the 64-bit system. It is because many necessary attack functions like *strcmp* and *write* are absent in the PLT table on the 32-bit system.

SROP In Sigreturn Oriented Programming (SROP) method [6], attackers set up a fake signal stack frame and invoke a *sigreturn* system call. SROP skips the signal sending and handling steps in the normal signal mechanism and directly returns from the signal. During the signal return procedure, the OS kernel will restore the process's environment with register values that are preserved on the signal stack. Since the signal stack frame is crafted by the attacker, the SROP attack can easily manipulate all register values.

SROP also has two limitations. First, SROP cannot defeat ASLR and XOM defenses because it needs to know the gadget locations in advance. It searches for the required gadgets by means of traditional methods. Nonetheless, the traditional methods usually require dumping the code online or offline to find gadgets. However, the action of dumping code segment is prohibited by the XOM defense. Second, the SROP method searches the vsyscall page for syscall gadgets. However, the vsyscall is emulated in the newer kernel version, which prevents the adversary from using the gadgets in the user-space.

Other approaches such as [11,12] require a JIT compiler. They use additional code injection methods to defeat the two recently proposed defense mechanisms: fine-grained ASLR and XOM.

2.2 ROP defense methods

The fine-grained ASLR method can thwart code-reuse attacks by randomizing executable code at function granularity [8,13], basic code block granularity [14,15], or instruction granularity [16,17], so gadgets' memory locations become unpredictable.

Meanwhile, a defense mechanism called XOM (execute-only memory) [3] is proposed, which prevents attackers from reading the code content by restricting its read permission. For instance, XnR [18] uses a page fault handler to implement the XOM mechanism. R^X [19] makes use of an extended page table to achieve XOM. Moreover, XOM can be easily realized by leveraging a new Intel hardware feature, i.e., memory protection key (MPK). MPK utilizes four reserved page-table bits in the page entry to achieve efficient intra-process memory isolation. XOM-switch [20] uses MPK to enable execute-only memory for unmodified binaries, which only introduces little runtime overhead. In the kernel, kR^X [21] is proposed as a kernel hardening scheme based on execute-only memory and code diversification.

Methods that hide or encrypt critical pointers also have been proposed. Crane et al. [9,10] introduce the concept of leakage-resilient diversification. They propose a protection mechanism called code-pointer hiding (CPH), which is used to facilitate fine-grained ASLR and XOM defense. It hides code pointers by replacing code pointers in readable memory with pointers to trampoline functions. Trampoline functions are protected by XOM and thus cannot be leaked. Even fine-grained ASLR and XOM can defeat direct JIT-ROP attacks, while the CPH technique can thwart the enhanced JIT-ROP attacks.

2.3 Missing returns behind syscalls

The syscall gadget is `syscall; ret` gadget (64-bit) or `int 0x80; ret` gadget (32-bit). Generally, attackers require return instructions in syscall gadgets to stitch a sequence of system calls. However, it is extremely hard to find these gadgets in either victim programs or libraries. According to our investigation in Section 5, we find that all of the three victim programs evaluated in our experiments, Nginx (64-bit), ProFTPD (32-bit), and Apache (32-bit), have no syscall gadgets. Here we further conduct some surveys on common programs and libraries to illustrate this issue. Detailed results are given in Fig. 1.

We survey several libraries, including *glibc*, *ld*, *libpthread*. In common cases, there is no *ret* instruction following the syscall instruction. In glibc-2.19.so, there are a total of 435 syscall instructions. Only nine of them are directly followed by a *ret* instruction. Five of them have the *ret* instruction

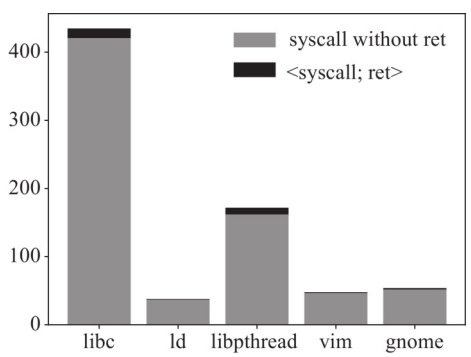

Fig. 1 The number of syscall gadgets in x86_64 libraries and binaries

behind them, but there are some other instructions between syscall and return. The remaining 421 syscalls have no *ret* instructions behind them. In ld-2.19.so, there exist 38 syscall instructions. Only one is followed by a *ret*. In libpthread-2.19.so, 172 syscall instructions are found in code. Among them, two syscall instructions are directly followed by a *ret*. We also investigate some other commonly used applications, such as *vim* and *gnome*. In vim-3.0, there are 48 syscall instructions in total. Among them, only one instruction is directly followed by a *ret*. In gnome-3.4.1, 54 syscall instructions are found in the code segment. Only two instructions are directly followed by a *ret*. In the 32-bit system, we survey some web programs, including Ghttpd, Orzhttpd, and Wuftpd. All of them have no `int 0x80; ret` gadget.

3 Overview

3.1 Attack assumption

In this section, we present some necessary assumptions of defense techniques and victim programs in SeBROP attack.

3.1.1 Assumption of defense techniques

During our SeBROP attack, we assume the following defense mechanisms are enabled in the system.

- **Stack canary** The stack canary [7] is placed on the stack to prevent attackers from overwriting the return address.
- **DEP** Data execution policy stops malicious code injection attacks by setting writable pages non-executable.
- **ASLR** Address space layout randomization technique [2] randomizes the memory-mapping address of code segments, data segments, stack segments, heap segments, and segments of libraries.
- **Fine-grained ASLR** Fine-grained address space layout randomization [8] is a variant of ASLR. It randomizes the order of basic code blocks in code segments and libraries. We assume fine-grained ASLR is deployed in all code segments, involving text and libraries.
- **XOM or heisenbyte** Both execution-only memory [3] and destructive code read [22] aim to prohibit attackers from reading the content of the code segment. We assume all code segments are non-readable.
- **Code pointers hidden** We assume that all code pointers to library functions are hidden by CPH method [10]. It prevents attackers from leaking function addresses.

3.1.2 Assumptions of victim program

In our SeBROP attack, the vulnerable program should meet the following requirements.

- The victim program must be a server-side program. The server should restart automatically after a crash without invoking *execve*. The program will not be re-randomized after a crash and the stack canary remains unchanged.
- The victim program should contain a stack buffer overflow vulnerability. If the stack canary is in use, the buffer overflow must not be a null-terminated string overflow.
- The attacker can crash the server as many times as they wish while conducting the attack.

3.2 Attack outline

Our goal is to find all kinds of gadgets blindly to construct a Turing-complete set. Generally, our attack consists of four steps: (1) Speculating the stack layout to leak the stack canary and a return address. (2) Fingerprinting all valuable gadgets blindly. These gadgets include memory load/store gadgets, arithmetic gadgets, logic gadgets, and branching gadgets. (3) Stringing a sequence of syscall instructions by leveraging the signal mechanism. (4) Launching a remote shell with found gadgets in previous steps.

4 SeBROP method

In this section, we first illustrate how to blindly fingerprint all kinds of gadgets excluding the syscall gadget. Then, we implement the syscall gadget based on the signal mechanism. Finally, we launch a remote shell and complete the attack.

4.1 Stack reading and find basic gadget

This step is analogous to the original BROP attack [4]. The attacker speculates the stack canary in three steps. First, the attacker overwrites a single byte of the canary with a value ranging from 0 to 255. If the value matches the first byte of the canary, the program will not crash. Then, the attacker continues to overwrite the remaining bytes until identifying the entire canary. Afterward, the attacker can overwrite the stack frame pointer (*rbp*) or return address byte-by-byte to obtain the exact value.

The basic gadget is a type of gadget in the form of `pop regs; ret`. The *regs* are four general 64-bit registers, i.e., *rax*, *rdx*, *rdi*, and *rsi*. In order to find the basic gadgets remotely, we scan the application code segment by overwriting the saved return address with a pointer and inspecting program behavior. Two situations may happen: the program crashes or it hangs; in turn, the connection either closes or remains open. Most of the time the program crashes. However, when it does not, a stop gadget is found. By manipulating the stack layout and inspecting program behavior, the attacker can further find the `pop regs; ret` gadgets.

4.2 Control all registers

In an ROP attack, merely controlling four general registers through the basic gadget is far from enough. We need to take control of more registers. However, there present two challenges to achieving this goal. First, after finding the `pop regs; ret` gadgets by blind execution, we need to further identify which register the gadget pops to. This procedure is quite sophisticated and inefficient. Second, other general registers are caller-saved registers. Consequently, there are no `pop regs;ret` gadgets for them in the code segment. Therefore, we cannot use the `pop; ret` gadget to manipulate these register values.

Hence we use the SROP method [6] to address these issues. We leverage the *sigreturn* system call instead of the `pop regs; ret` gadget to manipulate these general registers. This system call restores the values preserved on the stack into

the corresponding registers. If we place a crafted signal frame on the stack and execute this system call, we can control arbitrary general registers.

4.3 Blindly find turing complete gadgets

Next, we need to fingerprint more useful gadgets blindly. The Turing-complete gadget set includes six types of gadgets: memory load/store gadgets, arithmetic gadgets, logical gadgets, branching gadgets, and syscall gadgets. In the following section, we will describe how to find each kind of gadget on the x86_64 platform. The methodology of finding gadgets on the 32-bit x86 platform or other platforms is similar.

4.3.1 Load/store gadget

A memory load/store gadget is a kind of gadget that can be exploited to manipulate the register value or memory value. There are six steps to identify a load/store gadget.

Store gadget In the following, we take `mov [rax-20], rdi;ret` as an example to illustrate how to find the store gadget.

Step 1 This kind of gadget may cause a crash if the addressing register is an invalid address. In this case, if rax minus offset 20 is not in the legal memory area, the program will crash. Hence we set all register values as writable addresses (some lower addresses on the stack) and place a testing address on the stack to validate whether the program will crash. If not, we set all register values as zero and test again. Supposing this time the program crashes, we can assert that the testing address points to a memory access gadget, and the gadget is in the form of `op reg1, [reg2+off]; ret` or `op [reg1+off], reg2; ret`. Note that we have to figure out an address on the stack for use. We locate the on-stack address through stack reading, at the very beginning of our attack. As mentioned above, after we brute-forcing the stack frame pointer value, we obtain an on-stack address.

Step 2 Then we set all registers as read-only addresses. If the program crashes, we can ensure that the gadget is in the form of `op [reg1+off], reg2; ret`. There is a read-only area in the process address space called *vsyscall*. It is mapped into the fixed address in every process. We can use the vsyscall to distinguish the memory load gadget from the memory store gadget. We set all registers as a vsyscall address. If the program crashes, it is a memory store gadget. Otherwise, it is a memory load gadget.

Step 3 This step infers the addressing register *reg1*. We can validate all candidate registers one by one. However, we propose a method to improve its efficiency. Since we can control all fifteen general registers, the addressing register is among them. We set the first seven general registers as a writable address and set the remaining eight registers as zero. If the program does not crash, the addressing register is in the first set. Otherwise, the addressing register is in the second set. Then we can reduce the range of the addressing register. The test repeats until the correct register is found. Now we find that the gadget is in the form of `op [rax+off], reg2; ret`.

Step 4 This step infers the immediate offset value *off*. We set the addressing register as a lower address on the stack (unused space). Then we initiate a *write* system call to dump the memory area of the lower stack. We dump one page and the middle address is the addressing register's value. The dumped memory page contains a non-zero value, which is the same as one of the general registers' values we set beforehand. According to the location of the non-zero value in the buffer, we can identify the immediate offset. We utilize this location to subtract the middle location of the buffer to calculate the immediate offset. Here we confirm the gadget is in the form of `op [rax-20], reg2; ret`.

Step 5 To infer the source register, we set all general registers except the addressing register to different values. Then we implement a *write* system call to dump the memory that the gadget writes to from remote. The received value is the same as one of the general register values we previously set. According to the received value, we can easily identify which register is the pass value register.

Step 6 To infer the operator, supposing that the written value is always equal to the value in the pass value register, we can confirm that it is a *mov* instruction. The algorithm of differentiating various operators is given in algorithm 1. Finally, we recognize that the gadget is `mov [rax-20], rdi; ret`.

Load gadget Here we take `mov rax, [rdx+40]; ret` as an example to describe how to find the load gadget.

Step 1 and **Step 2** are similar to finding the memory store gadget. The difference is that the memory load gadget will not cause the program to crash when setting all registers to read-only addresses. As a result, this gadget is in the form of `op reg1, [reg2+off]; ret`.

Step 3 is also the same as finding store gadget. Now we can confirm that the gadget is in the form of `op reg1, [rdx+off];ret`.

Step 4 This step speculates the immediate offset value *off*. In contrast with the method used in finding the store gadget, we leverage the address range of a valid memory region to identify the immediate offset of a memory load gadget. If the value of the addressing register plus an immediate offset is in the memory-mapping area, the program will not crash. Other-

Table 2 Turing-complete gadget set. The *sigreturn* gadget consists of `pop rax; ret` and *syscall* instruction and a counterfeit signal frame. The *sigreturn* can set all registers' value, including *rsp*, *eflags*. *reg1* and *reg2* represents any general registers, *off* represents the immediate offset in the gadget.

Catagory	Gadgets
Set register	< `sigreturn` >
Load/Store	< `mov reg1, [reg2+off]; ret` >
	< `mov [reg1+off], reg2; ret` >
Arithmetic	< `add [reg1+off], reg2; ret` >
	< `adc [reg1+off], reg2; ret` >
	< `neg rax; ret` >
Logical	< `xor [reg1+off], reg2; ret` >
	< `or [reg1+off], reg2; ret` >
	< `and [reg1+off], reg2; ret` >
	< `rol [reg1+off], reg2; ret` >
	< `not rax; ret` >
Branching	< `sigreturn` > (unconditional jump)
	< *instr*$_1$...*instr*$_n$, `sigreturn` > (conditional jump)
System Call	< `syscall; ret` >

wise, the program crashes. As the offset may be a positive value or a negative value, an upper boundary and a lower boundary are both needed. The upper boundary is used to identify a positive offset, the lower boundary is for a negative offset (depicted in Fig. 2). To identify the immediate offset, we can test the addresses one by one. When the value of the addressing register does not cause the program crash, we set it to a larger/smaller one. Eventually, we can find a critical point that does not cause the program to crash, but the addition or subtraction of one will produce a crash. The mapped boundary address is a page-aligned value, so we can use the mapped boundary address minus the critical point value to get the immediate offset. Here we find the gadget is in the form of `op reg1, [rdx+40]; ret`.

Step 5 and **Step 6** show important distinction from finding memory store gadget. Identifying the operator and destination operand of load gadgets is more complicated than memory write gadgets by blind execution. The method is based on the memory store gadget we already found. We take advantage of the memory store gadgets to dump some registers to inspect the value change. If the register value is identical to the memory load gadget source operand, we find the appropriate register and confirm that the operator is a *mov* instruction.

4.3.2 Arithmetic and logical gadget

The arithmetic/logical gadget involves instructions that perform some arithmetic/logical operations, such as *add*, *adc neg*, *xor*, *or*, *and*, *rol*, *neg*.

Add/Adc We merely target two types of gadgets which are in the form of `add [reg1+off], reg2; ret` and `adc [reg1+off], reg2; ret`.

Step 1 to **Step 5** are the same as finding memory store gadgets.

Step 6 We identify the operator in this step. The insight is that the dumped value is doubled when the *add* or *adc* gadget execute twice. In contrast, the dumped value won't change for a *mov* gadget. Furthermore, as we can control the *eflags* register using SROP method, we can easily differentiate the *add* operator from the *adc* operator. If the *CF* flag is zero, the result of *adc* is equal to *add*. Once the *CF* flag is one, the result of *adc* is greater than *add*.

Xor/Or/And/Rol Here we only identify gadgets in the form of `op [reg1+off], reg2; ret`.

Step 1 to **Step 5** make no difference with finding memory store gadgets.

Step 6 The key observations are as follows. *xor* operator can be easily identified when executes twice because the written value will change to zero. The gadget with *or* instruction has similar characteristic with *mov*. But if we change the value in the pass value register between two executions, we can easily differentiate a *mov* from an *or* operator. When the instruction executes twice, only the second value is left in memory for the *mov* instruction. For the *or* instruction, if the two values are distinct, the first value still can be found in memory. The *and* and *rol* instruction do not leave any trace when one of their operands is zero. Therefore, we set an area full of 0x3f to identify both *and* and *rol*. If the received value is the same as

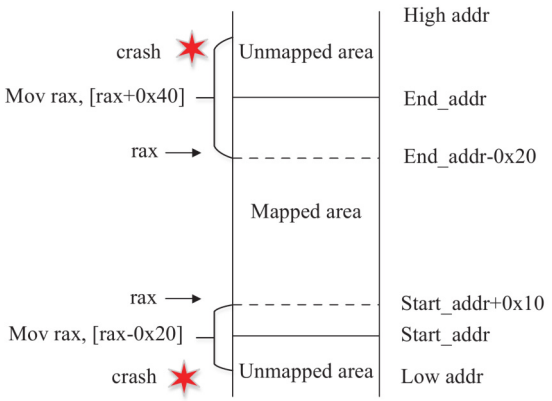

Fig. 2 Indentifying immediate offset of the load gadget. For instance, the gadget is `mov rax, [rax+40]; ret`. If we set the *rax* register value bigger than *end_addr* minus 0x40, the gadget will cause the program crash. Otherwise the program won't crash. The *end_addr* minus 0x40 is the critical point. If we find this value, it can reveal the immediate offset. The *start_addr* and *end_addr* represents the memory mapping upper and lower boundaries

Algorithm 1 Identify operators in gadgets

```
ROP_chain1 = sigframe+ write syscall. Sigframe set registers value forthe gadget.write syscall dump the memory location that the gadget writes to from remote.
In sigframe, gen_regs.value varies from 1 to 15. rip = test_addr. Addr_reg =writable_addr.
Send the exploit and receive it from the network. The received value is the valuethat the gadget writes.
if receive_value1 = 0 then
    The gadget may be a And or Rol, goto 31.
else
    receive_value1 ∈ [1, 15]. The received value reveals the pass value register,its location in the receive buffer tells the offset.
end if
ROP_chain2 = sigframe+test_addr+ write syscall. The test_addr is executed twice. Send the exploit again.
if received_value2 = receive_value1 then
The gadget may be a Mov or Or, goto 19.
else
    if received_value2 = 2 * receive_value1 then
        The gadget may be a Add or Adc, goto 25.
    else
        received_value2 = 0, The xor; ret gadget is found.
    end if
end if
ROP_chain3 = sigframe+sigframe2+write syscall. In sigframe2, src_reg = 0xf0.
Send the exploit.
if received_value3 = 0xf0 then
The mov; ret gadget is found.
else
received_value3 = 0xf0 + received_value1. The or; ret gadget is found.
end if
Send ROP_chain1. In sigframe, eflags = 1.
if received_value4 = received_value1 then
    The add; ret gadget is found.
else
    The adc; ret gadget is found.
end if
Send ROP_chain1 again. In sigframe, Addr_reg = shared_mem (full of 0x3f).
if receive_value5 ∈ (1, 15) then
    The and; ret gadget is found.
else
    receive_value5 = a cycle shift of 0x3f. The rol; ret gadget is found.
end if
```

one of the general registers, it is an *and* instruction. If the received value is the cycle shift of *0x3f*, it is a *rol* instruction. What is worth mentioning is that we use the shared memory in the process to store a series of *0x3f*. Thus we can use it all the time after the initial setting. Specifically, we use the ROP chain to invoke a *read* system call to get the input (a series of *0x3f*) from remote.

Not/Neg We observe that the operand of this gadget type are mostly *rax* or *eax*. Therefore, we identify the gadgets like this, `neg rax; ret` or `not rax; ret`. We find that the execution of this gadget will change the *rax* register value. For instance, the *rax* value after *neg* gadget invocation is the negative value of the original *rax*. We set the *rax* register value as a particular one. Then, we place a memory store gadget after the gadget to dump the *rax* value to see whether the value changes accordingly. If so, we find the right gadget.

4.3.3 Branching gadget

Branching gadget adjusts *rsp* register to transfer the control flow during gadget execution. It can be leveraged to implement the unconditional jump. To achieve a conditional jump, it needs to cooperate with several other gadgets.

Unconditional jump. we utilize *sigreturn* to realize an unconditional branch gadget. The SROP method can manipulate the *rsp* register. Thereby, we can jump to any locations in the ROP chain by modifying the *rsp* register.

Conditional jump. We leverage the *adc* gadget combining with *neg*, *and*, *sigreturn* gadgets to implement a conditional branch gadget. As depicted in Fig. 3, we place a counter in the unused lower space on the stack. Then, we subtract one from it and load the result to the register *rax*. We execute the `neg rax; ret` gadget subsequently. It influences the CF flag based on *rax* value. If *rax* is zero, CF flag is set to zero. Otherwise, the CF flag is changed to one. Next, we utilize an *adc* gadget to propagate the CF flag value into memory. If the counter becomes zero, the second memory location will be -1. Otherwise, the second memory location will be 0. We perform an *and* operation on the second memory location with Δ rsp. Δ rsp is a precalculated value. The operation result could be either the value of Δ rsp or zero. Finally, we add the value to the location of *rsp* in the fake *sigreturn* frame. The original value of *rsp* in the *sigreturn* frame is pointed to the gadget that subtract one from counter. After executing the *sigreturn* gadget, the control flow transfers either to the next gadget or back to the gadget that calculates the counter.

4.4 Consecutive sigreturn and write system call

As mentioned above, we invoke a *write* system call after a *sigreturn* system call. These two system calls can be executed consecutively without a `syscall; ret` gadget. This happens due to the unique characteristic of the *sigreturn* system call. Nonetheless, this cannot occur in the case of arbitrary two system calls. So we still need to introduce a signal mechanism to facilitate our attack.

Most specifically, the *sigreturn* system call can take control of the *rip* register's value after it returns. It utilizes the polluted on-stack value to restore the *rip* register's value. Normal system calls directly execute the instructions after system call when returning from the kernel, while the *sigreturn* system call returns to the location where attacker sets to the *rip* register through the fake signal frame on the stack. So unlike normal system calls, the program won't crash after the *sigreturn* system call. Here the attacker again seizes the control and has an opportunity to execute an additional system call.

4.5 Avoid system hanging

Our attack requires several working processes being available and not ending up in a situation where all processes are executing infinite loops. To avoid these circumstances, we exploit a working process in the victim program and execute a *kill* system call. The worker soon sends a *SIGKILL* signal to all workers in the process group. Previously hanging workers will restart. We monitor the number of living workers by testing the socket state and keep it larger than one during the attack.

4.6 Stitching a sequence of syscalls

The ROP attack relies on return instructions to stitch a

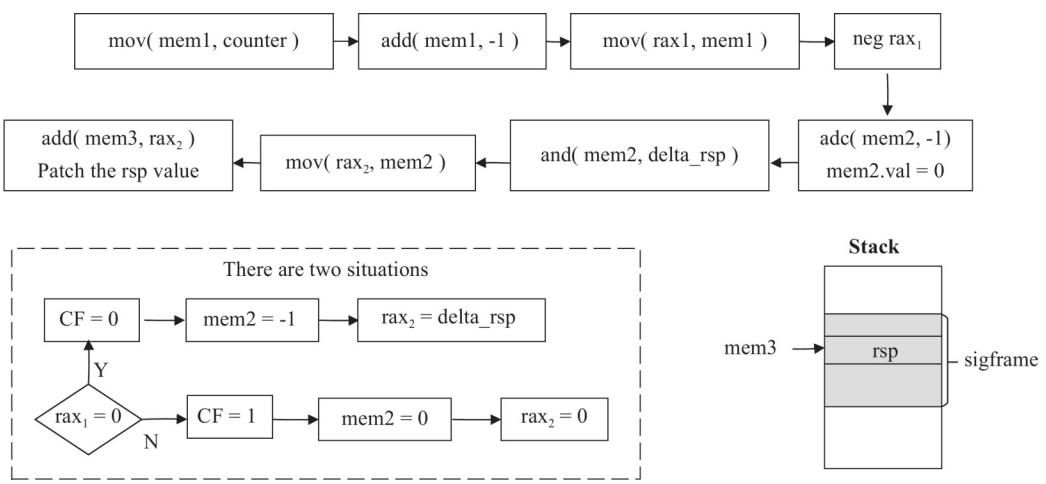

Fig. 3 Implementing conditional jump. We describe steps to realize the conditional jump with the gadgets we found. When we perform *neg rax* operation, two situations will happen according to the *rax* value. We show these two situations in dotted box. Ultimately, we use a *sigreturn* system call to divert the control flow

sequence of system calls. Nonetheless, syscall instructions are usually not followed by return instructions. Without the return instruction, attackers cannot control the execution flow after the *syscall* instruction. In this paper, we provide an alternative solution to realize this functionality. We observe that there is a potential point to divert the control flow before a system call exit. Further, we show how to control it precisely and implement a sequence of consecutive system calls.

4.6.1 Control flow hijacking

In Unix-like operating systems, when the processor handles a *syscall* instruction from a user program, the instruction causes an exception, which transfers control to an exception handler in the kernel. Notice that if a signal arrives while the processor is executing the system call routine, then just before the system call exiting, the kernel will check these pending signals. If pending signals are present, the kernel calls the signal handler. When it finishes, the kernel will return to the use-space as usual. However, this routine is vulnerable to control hijacking if we redirect the flow of the signal handler and never return control.

To exploit the potential control flow hijacking point, we first register a crafted signal handler for this process. Then, we exploit another process to send a signal to the original process when it executes a system call. The victim process checks the pending signals after the system call execution. Consequently, it diverts the control flow and executes the remaining gadgets on the stack.

Further, we can stitch a sequence of system calls. As shown in Fig. 4, attackers utilize the ROP chain to perform a *sigaction* system call to register a forged signal handler for process $proc_1$. Then, attacker use the ROP chain in process $proc_2$ to perform a *kill* system call to send a signal to $proc_1$. The signal handler redirects the control flow of $proc_1$ to the remaining ROP chain on the stack. The gadget chain is placed by attackers beforehand. In the ROP chain, the program executes the second system call. In this period, the process $proc_3$ sends the second signal to $proc_1$ through the ROP chain. $proc_1$ receives the signal and executes the third system call.

There are two critical points for this control flow hijacking mechanism: (1) a crafted signal handler that redirects the control flow to the remaining ROP chain; (2) the timing that the signal arrives.

4.6.2 Malicious signal handler design

The signal handler is leveraged to trigger the execution of the remaining gadgets. In order to circumvent the DEP defense technique, the content of the signal handler can not be injected. It must be composed of ROP gadgets. So our signal handler takes advantage of a special type of gadget, i.e., `add rsp, big_constant; ret` gadget. This kind of gadget widely exists in the program and thus can be easily found. This gadget skips the *sigreturn* system call and never return from the signal. Normally, after executing a signal handler, the OS kernel will invoke the *sigreturn* system call and then execute instructions behind the *syscall*. This will cause a crash. In our signal handler design, we adjust the *rsp* register and divert the control flow. As a result, it will not execute the *sigreturn* system call and will execute the remaining ROP gadgets on the stack instead.

Moreover, the *add rsp* instruction in the signal handler should skip the signal frame. When the process receives a signal, the kernel will insert a signal frame on the stack. The signal frame is on top of the stack. The value of the *big_constant* in the gadget must bigger than the size of the signal frame. The average size of the signal frame is 0x5a0 in x64. A value that is bigger than the size can satisfy our requirements. Note that different program provides different *add rsp* gadget, the *big_constant* value can not be determined beforehand, so the precise value of the *rsp* register is undetermined. To address this issue, we place a bunch of *ret* instructions before the ROP chain on the stack. These *ret* instructions function as trampoline. When the *ret* instruction of the signal handler is executed, it will jump to the trampoline and then slide to the remaining ROP chain.

Next, we need to determine the *big_constant* value. To this end, we put a testing address on the stack and then fill the stack with 0x5a0 zeros. Finally, we put numerous *ret* instructions ended with a stop gadget on the stack. If the program does not crash, we adjust the number of zeros to eventually confirm the *big_constant* value in the gadget.

Note that the signal handler does not limit the number of subsequent gadgets. We skip the *sigreturn* system call. When the control flow transfers to the remaining ROP chain, we can execute an arbitrary number of gadgets. Our signal handler is not sophisticated and can be used repeatedly. It only adjusts the *rsp* register value. The content of the signal handler does not consist of any system calls or special library functions.

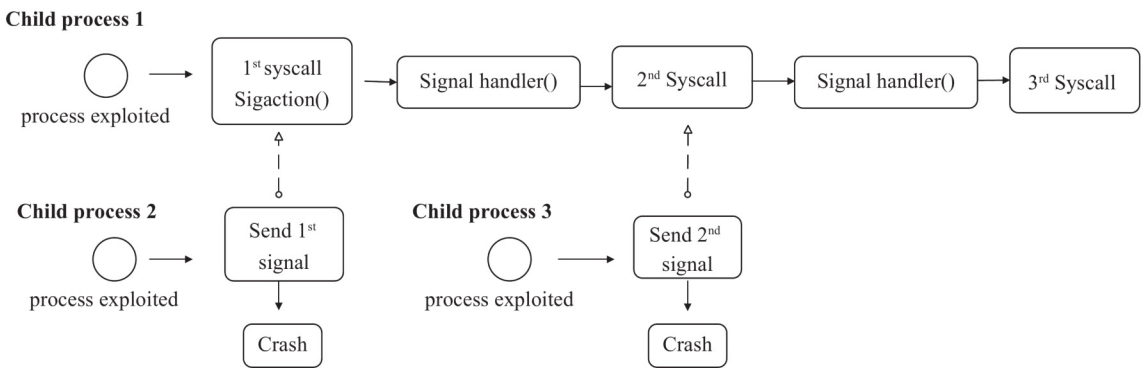

Fig. 4 Stitching a sequence of system calls

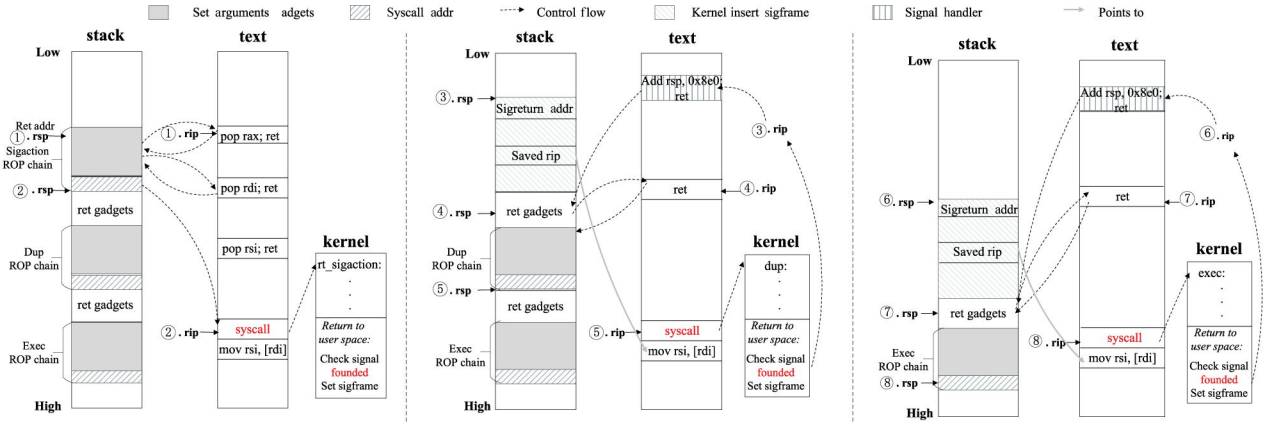

Fig. 5 Stack layout change. Here we demonstrate how our malicious signal handler works. We describe the stack layout change of the signal receiving process in eight stages

The specific functionality is implemented in the remaining ROP chain. The advantage is two-fold. First, the crafted signal handler in the ROP attack is constrained to have only one gadget. There is no extra space (gadgets) for attackers to realize more functionalities. Second, if the signal handler implements specific functionality, it will limit the generality of the signal handler. We must register different signal handlers for different system calls.

Figure 5 illustrates the stack layout change during signal execution. At stage ①, the victim program receives a crafted malicious package from the remote attacker. The attacker prepares the victim stack as the left part in Fig. 5. In stage ②, the victim program executes the *sigaction* system call, which installs our malicious signal handler. In stage ③, the victim receives the first signal. During the execution of the *syscall* instruction, the first signal arrives. The signal frame is inserted on the top of the stack by the kernel.

The stack layout changes are shown as the middle part in Fig. 5. Stage ④ shows the stack layout after the execution of the signal handler. When executing the `add rsp, 0x8e0` instruction in the signal handler, the *rsp* register is updated to point to the starting gadget of the ROP chain. When executing the next *ret* instruction in the signal handler, the program pops the address of the starting gadget of the second ROP chain into the *rip* register.

In phase ⑤, victim program executes the second system call. The second system call for execution could be any system call that is correlated with the attack. In phase ⑥, the victim program receives the second signal. During the execution of the *syscall* instruction, the second signal arrives. The signal frame is put at the top of the user stack by the kernel. The stack layout changes are shown as the right part in Fig. 5. Stage ⑦ shows the stack layout after the second execution of the signal handler. After executing the specific signal handler, the *rsp* register is updated again. The program continues to execute the third system call in phase ⑧.

4.6.3 Timing of signal arrival

A signal is sent during system call execution. The signal must arrive precisely. If it arrives too early, the system call has not been executed. If it arrives too late, the program has already crashed. The signal is sent by a sibling process that is exploited in our attack.

In our targeted program, there is usually some shared memory between processes. For example, address range from 0x7f0000000000 to 0x7ffa00000000 is shared by processes in Nginx. We put a shared flag in the shared memory to synchronize the signal sending process and the receiving process. The signal sending process monitors the shared flag and waits for the right moment to send the signal. The signal receiving process modifies the shared flag to inform the sending process to send the signal when it is ready to receive a signal.

These are all implemented through the ROP chain. The signal sending process first utilizes the gadgets we previously found to implement a conditional jump. The conditional jump depends on whether the shared flag value changes. If the value changes, the process jumps to execute a *kill* system call to send the signal. Otherwise, it continues to check the shared flag value. The signal receiving process changes the shared flag value to inform the signal sending process before it executes a system call. It is achieved by using a *mov* gadget.

Improving efficiency The whole attack mentioned above is a race condition, so it can not succeed every time. Although we carefully synchronize two processes, it is still possible that the signal can not arrive during the execution of a system call. It is because the period is too short.

We propose an optimization to improve the hit rate. We register a signal handler for the *SIGSEGV* signal. When the system call routine finishes, the program continues executing. A memory access exception will occur. As a result, it will trigger the *SIGSEGV* signal handler execution. The following gadgets will execute. This approach improves efficiency compared with the previous one.

4.7 Launch remote shell

In this subsection, we provide two solutions to launch a remote shell.

Solution I We exploit four child processes to accomplish the attack. The first three child processes, child process 1 (CP1), child process 2 (CP2), and child process 3 (CP3) are used to send signals. We have to implement four system calls (e.g., *sigaction*(), *dup*2(0, *sock*), *dup*2(1, *sock*), *exec*()), so we need three signals to connect these syscall instructions.

The first three processes only implement one system call

(*kill*) through the ROP chain. It is leveraged to send a signal to other processes. The signal sending process need not know the process ID of the signal receiving process. It can use parameter zero, which means that the signal is sent to all processes in the same group as the signal sending process.

The fourth child process (CP4) communicates with three other child processes through shared memory. The fourth child process (CP4) is used to launch a shell. The four system calls executed by CP4 are implemented in four ROP chains. These system calls are connected by a series of *ret* instructions. The first chain implements the *sigaction* system call. Note that the first system call must be *sigaction* because we must register a handler. Otherwise, the default signal handler will respond to the signal and the control flow hijacking will fail. The second and third ROP chains implement the *dup2* system call. These two system calls redirect the input and output stream to the socket. Therefore, the attacker can interact with the remote server. Finally, in the fourth ROP chain, an *exec* system call is invoked to launch a remote shell.

Solution II Our second method is an improvement on the first method. The first method needs the cooperation of sibling processes and sends the signal timely. In the second method, we register a signal handler for the *SIGSEGV* signal and only one process is in need. After the system call has been executed, the control flow executes the instructions behind it. It usually encounters a memory access instruction, which triggers a memory access exception. The kernel sends the *SIGSEGV* signal to the process. We replace the signal handler of *SIGSEGV* with ours to execute the remaining ROP chain on the stack. When the process is exploited by a remote attacker, it executes four system calls in four ROP chains (connected by a series of *ret* gadgets). The process invokes *sigaction* system call to register a signal handler for *SIGSEGV* signal in the first ROP chain. The following steps are similar to those in solution I.

5 Evaluation

5.1 Methodology

Experimental setting We conduct three experiments. Two of them (Apache and Nginx) are in a VMWare virtual machine. One (ProFTPD) is in a VirtualBox virtual machine. The host machine has an Intel Core i7-4600U with 2 cores @ 2.1GHz 2.7GHz and 8GB DRAM. In our 32 bit experiments, the operating system of the virtual machine for Apache is Fedora 6 and the operating system of the virtual machine for ProFTPD is Ubuntu 12.04. In our 64 bit experiments, the operating system of the virtual machine for Nginx is Ubuntu 14.04.3.

Program configurations We use the default configuration settings in the Apache 1.3.49. The Apache server is listening on port 80. In the default configuration, the server process number is five. The number of the child process will increase linearly with the requests that need to be handled. Equally, we use the default configuration settings in the ProFTPD 1.3.0. The FTP server is listening on port 21. We create a user *ftptest* with password *ftptest*. In Nginx 1.4.0, we change the *worker_processes* from 2 to 4. Because in the first solution of our attack, we need at least four child processes to launch a shell. In the access control part, we need to allow the remote attacker's IP to access the server. After setting the configuration file, we need to restart the server to enable this setting.

Memory layout The memory layout is as follows. Since the ASLR defense is enabled, the starting addresses of all the libraries and heap and stack are randomized in each execution. But the text and data segment are always mapped to the same memory addresses.

Deployed defense We deploy the DEP and ASLR defense in all of our experiments. The stack canary is enabled in the Nginx attack. Although our attack can bypass advanced defenses such as the fine-grained ASLR and XOM defense, we do not deploy them into the target system due to the fact that the fine-grained ASLR and XOM defense have not been widely adopted in the common systems. In addition, there is no publically available code for these defenses. Nevertheless, we do not violate any of these defenses in our attack no matter whether they are deployed or not.

5.1.1 Attack procedure on apache

Apache is an HTTP server and *mod_jk* is the Apache Tomcat Connector. The vulnerability *CVE-2007-0774*(See CVE) in *mod_jk* 1.2.20 is a missing boundary check bug that would cause a stack buffer overflow. The stack-based buffer overflow in the *map_uri_to_worker* function allows attackers to execute arbitrary code via a long URL that triggers the overflow in a URI worker map routine. As shown in Listing 1, the *url* buffer is stored on the stack with the size 4096. The function copies the *uri* string to the buffer without any bounds checking. The *uri* is a user input string. If its length is oversized, it will cause an overflow. When we construct a request with a big size which is more than 4095 bytes, we can trigger the stack buffer overflow.

Listing 1 Vulnerable code in Apache

```
const char *map_uri_to_worker(jk_uri_worker_map_t *uw_map
    ,
const char *uri, jk_logger_t *l)
{
unsigned int i;
char *url_rewrite;
const char *rv = NULL;
char  url[JK_MAX_URI_LEN+1];  //4095

if (!uw_map || !uri) {
  JK_LOG_NULL_PARAMS(l);
  JK_TRACE_EXIT(l);
  return NULL;
}

for (i = 0; i < strlen(uri); i++)
if (uri[i] == ';')
  break;
else
  url[i] = uri[i];   //no bounds check, stack overflow if
       uri.len > 4095
url[i] = '\0';
...
}
```

The way to exploit the vulnerability remotely is manifested in Listing 2. The *send_exp*() function creates a socket, packs the ROP chain in a request, and finally sends the request. The *do_try_exp*() function invokes *send_exp*() function to send an exploit and invokes *check_alive*() function to check whether the remote server is still alive. If not alive, it means the remote

process crash, the function returns zero. Otherwise, we send a new request to check whether the process can respond to the new request. If not, it executes an infinite loop, the function returns 2.

Because the target buffer filters out numerous characters, such as 0x0, 0x9, 0xa, 0xb. Many addresses cannot be put into the exploited buffer so we miss many gadgets. A complete `int 0x80; ret` type gadget is not found and even an *int 0x80* instruction does not exist. Only the *call *gs:0x10* instruction is available, so we use our signal approach to implement the execution of a sequence of system calls. We realize our signal mechanism by using the gadgets that we found blindly.

5.1.2 Attack procedure on ProFTPD

ProFTPD is an FTP server. The vulnerability *CVE-2006-5815* in ProFTPD 1.3.0 is a stack-based buffer overflow. The overflow can be exploited by an adversary to execute arbitrary code. Listing 3 shows the vulnerable code. The vulnerable function is the *sreplace* function in *support.c* file. The overflowed buffer *buf* is stored on the stack. By disassembling the code, we find that the pointer array *rarr* is stored next to it on the stack. There is an off-by-one comparison bug in this function so the string terminator is overwritten. We denote the site where the bug occurs. Since the adjacent array *rarr* is filled with some value, the string length of the *buf* is bigger than the original length *blen* when the string terminator is overwritten. Hence, in the next iteration of the while loop, the length parameter of the *sstrncpy* function becomes negative, which is perceived as a large integer. Now we can trigger the overflow to overwrite the return address.

Listing 2 Exploit code in Apache

```
def do_try_exp(rop)
  s = send_exp(rop)
  alive = check_alive(s)
  if not alive
    s.close()
    return false
  end

  req = "GET␣/␣AAA␣HTTP/1.0\r\n"
  req << "User-Agent:zzz"
  req << "\r\n"
  req << "Host:0x82-apache-mod_jk.c\r\n"
  req << "\r\n"
  s.write(req)
  alive = check_alive(s)
  s.close()

  return true if not alive
  return 2
end

def send_exp(rop)
  s = TCPSocket.new($ip, $port)

  req = ""
  req = sprintf("GET␣/%s%s␣HTTP/1.0\r\nUser-Agent:␣%s\r\
       nHost:␣%s\r\n\r\n", "A" * 4123, rop.pack("I*"), "",
        "Host:␣0x82-apache-mod_jk.c\r\n\r\n");  #  #4123
       ebp.pack("I*")

  s.puts(req)
  s.flush()
  return s
end
```

In order to trigger the execution of the vulnerable *sreplace* function, we modify the content of the *.message* file and send *CWD* to the server. The *.message* file exists in every directory are used to show some customized information. It contains some specifiers that will be replaced by other contents in *sreplace* function. The *CWD* command which changing the working directory will trigger the processing of the *.message* file and ultimately executes the vulnerable *sreplace* function.

We implement our attack in the Metasploit framework. Our attack script is written in the Ruby language. We will further illustrate the concrete steps implemented in our attack script in the following section. In ProFTPD, We find lots of useful gadgets and stitch them to launch the final attack.

5.1.3 Attack procedure on Nginx

Nginx is an HTTP and reverses proxy server, a mail proxy server, and a generic TCP/UDP proxy server. The vulnerability *CVE-2013-2028* in Nginx 1.4.0 is an integer overflow problem that causes a stack buffer overflow. We exploit this vulnerability to hijack the program's control flow and blindly find all types of gadgets that resided in its code segment.

Listing 3 Vulnerable code in ProFTPD

```
char *sreplace(char *s, ...)
{
 char *m, *r, *src = s, *cp;
 char **mptr, **rptr;
 char *marr[33], *rarr[33];
 char  buf[BUF_MAX] = {'\0'}, *pbuf = NULL;
 size_t mlen = 0, rlen = 0, blen; cp = buf;
 ...

 while(*src){
  for(mptr = marr, rptr = rarr; *mptr; mptr++, rptr++){
   mlen = strlen(*mptr);
   rlen = strlen(*rptr);
   if(strncmp(src, *mptr, mlen) == 0){
    sstrncpy(cp, *rptr, blen - strlen(pbuf));
    if(((cp + rlen) - pbuf + 1) > blen){
     cp = pbuf + blen - 1;
    }
    ...
    src += mlen;
    break;
   }
  }
  if(!*mptr){
   if((cp - pbuf + 1) > blen){ //off-by-one
    cp = pbuf + blen - 1;
   }
   *cp++ = *src++;
  }
 }
 ...
}
```

Figure 6 manifests the whole attack procedure. Our attack can be divided into the following steps.

Step 1 check vulnerability We start off by testing whether the remote server contains a stack buffer overflow. We stop the attack if the check fails. Once the stack canary is deployed, we attempt to brute-force the value by stack reading. We also obtain the stack frame pointer value and the return address in this step.

Step 2 find basic gadgets In order to initiate the *sigreturn* system call for later usage, we need to figure out the addresses of a `pop rax; ret` gadget and syscall instruction. We begin with finding all `pop reg; ret` gadgets and chain them together. We set all pop value to 33 (the *sleep* syscall number). One of them may feed into the *rax* register. Then we place a syscall candidate at the end of the chain. If the program won't

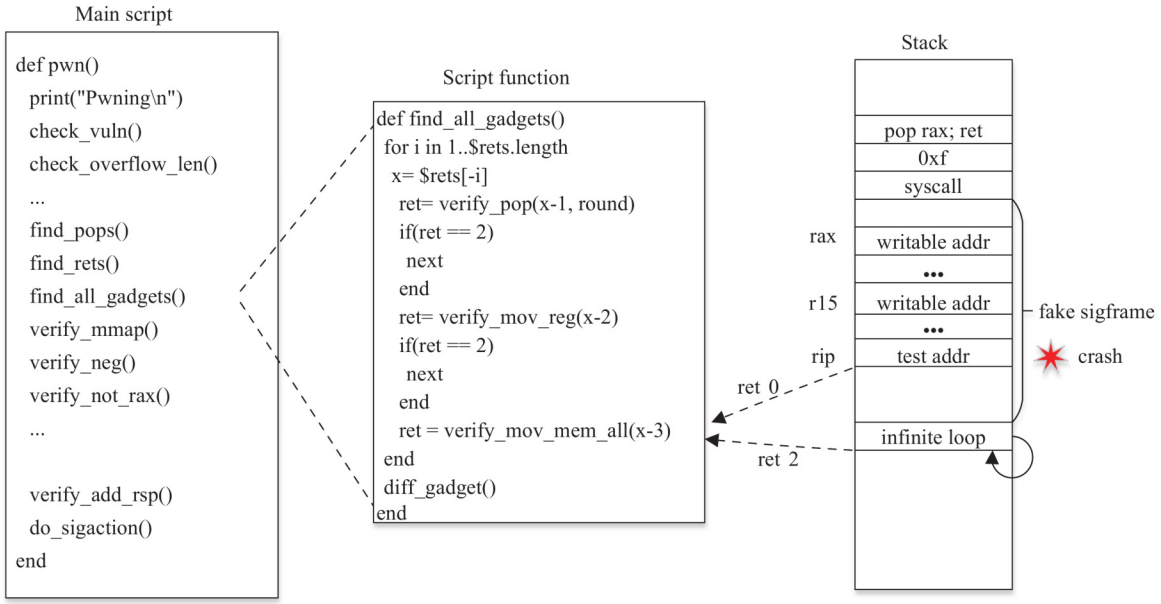

Fig. 6 The whole attack procedue in Ruby Script

crash and go to sleep, we find the syscall instruction. Next, we test the `pop reg; ret` gadgets one by one to further determine the `pop rax; ret` gadget. As a result, we can find the basic gadgets we need.

Step 3 locate all *ret* gadgets Up till now, we can further find *ret* and *ret*-like instructions. This kind of gadget does nothing but transfer control to the next gadget, so it can be easily distinguished by blind execution.

Step 4 identify all memory access gadgets This step can be implemented through the function *find_all_gadgets*. The global variable *$rets* contains all the *ret* instruction address we found in the previous step. For each *ret* instruction address, we choose the address two or three bytes before the *ret* instruction address as the testing address. These testing addresses are treated as gadget candidates. Then we use following three functions to verify them to distinguish valid gadgets. The function *verify_pop*() is invoked to test whether an address contains *pop; ret* gadget. By using the function *verify_mov_reg*(), we can confirm whether an address is pertaining to a memory read gadget. By leveraging the function *verify_mov_mem_all*(), we can recognize whether an address involves memory write gadget.

Here we elaborate on the implementation of the *verify_mov_mem_all*() function. We first place the *sigreturn* syscall and a counterfeit signal frame on the stack. In the signal frame, we set all general registers to a writable address (lower address on the stack), and manipulate the *rip* register to the test address *x-3* (an address three bytes before a return instruction) as shown in Fig. 6. Then we place a stop gadget on the stack which is an infinite loop statement. This ROP chain will lead the program to first execute the test address *x-3* then jump to execute the infinite loop. Since that we set all registers' values to a writable address, if the program crashes (the script function returns zero), it is not a gadget. Otherwise, it is a kind of memory access gadget, we can adjust the registers to further determine the addressing register of the gadget.

Step 5 differentiate memory access gadgets Step 4 merely determines whether a location contains a memory access gadget and the addressing register of the gadget. The remaining details are all unknown. Hence, the function *diff_gadget*() is leveraged to further identify all the details of a gadget, including its operator and operands. It can be used to facilitate the distinguishment of gadgets, such as differentiate *add* from *adc*. By manipulating the stack and sending different payloads, we can easily determine all the details of a gadget.

Step 6 find signal handler In this phase, we fingerprint the gadget we used as a signal handler. The gadget we used is in the form of `add rsp, big_constant; ret`. By padding enough zeros and *ret* gadget, we can finally distinguish it.

Step 7 get a remote shell Finally, we prepare a long ROP chain to initiate the *sigaction* system call. It will register a signal handler for the *SIGSEGV* signal. Then, it executes the *dup* system call and ultimately performs the *exec* system call to spawn a remote shell. All of them are implemented in the function *do_sigaction*().

During our attack, no code fragments are read in the memory and there is no prior knowledge of the binary. There is no `syscall; ret` type gadget available, so we leverage our signal approach to achieve the execution of a sequence of system calls.

Requirement For Overflow Byte The overflow bytes must meet the following requirements. First of all, the overflow bytes must not contain zero bytes, otherwise, it will stop the ROP chain. Moreover, if the victim program sanitizes user inputs, the overflow bytes must not contain the characters that will be filtered out by the program. In Nginx, the program does not perform any process on the input data, so all bytes but zero can be used in the ROP chain. In Apache and ProFTPD, the programs filter out numerous characters, such as 0x9, 0xa, 0xb. These characters can not be used as an overflow byte.

General length of ROP chain The length of the ROP chain varies during different attack phases. Table 3 exhibits the general ROP chain length in our Nginx attack.

Table 3 ROP chain length per attack phase in Nginx

Attack phase	ROP chain length
Searching for stop gadget	1 (8 bytes)
Looking for the < *pop*;*ret* > gadget	3 (24 bytes)
Fingerprinting the syscall instruction	51
Testing for a normal memory gadget	57
Differentiating a memory gadget	around 57 and 118
Launching the final attack	376

Notice that, at the time we fingerprint the syscall instruction, we need to chain all the `pop; ret` gadget we found with the test address, the ROP chain length is dependent on the pop gadgets in the victim code. In our attack, the ROP chain length is 51. When we test for a normal gadget, the ROP chain length is 57. The chain contains a forged signal frame. When we begin to differentiate a gadget, the length of the ROP chain varies depending on what kind of gadget the testing address contains. But the average length is around 57 and 118. (consists of one or two fake sigframe)

After we collect all kinds of gadgets, we need to launch the final attack. The final chain involves "sigaction; rets; dup2; rets; dup2; rets; exec". The number of returns depends on the constant value in the signal handler, which is the gadget we found in the victim(such as, `add rsp, 0x8e0; ret`). The constant value in the signal handler(such as 0x8e0) must bigger than the kernel pushed signal frame, otherwise, the remaining ROP chain will be covered by the signal frame. The number of returns must be (A-B)/8. "A" represents the constant value in the fake signal handler, here is 0x8e0. "B" stands for the length of the kernel pushed signal frame. In this case, the length of the signal frame is 0x5a0. Therefore we must set (0x8e0-0x5a0)/8 returns. Because the size of the kernel insert signal frame is not always the same, we need to pad more returns to improve the hit. Here the length of our final gadget chain is 376.

Gadgets collection: As a whole, the gadgets we found include, *pop, mov, add, adc, xor, and, or, rol, neg, not*. Now we take Nginx as an example. We show the concrete gadgets in all categories in Table 4. Here we only list the most frequently used gadgets.

Gadgets stitching After we collect all the usable gadgets, we chain them together to launch the final attack. In the following, we will elaborate on how to use the aforementioned gadgets to accomplish the attack. Towards the ultimate goal, there are two solutions, registering a signal handler for SIGSEGV and registering for other signals.

Solution 1 To register a signal handler for SIGSEGV, we first use the `mov qword ptr [rsi + 8], rax ; ret` gadget to write values into memory to construct a counterfeit *sa_sigaction* structure, before implementing the first *sigaction* system call. The pointer to the structure will be used as a parameter of the sigaction system call subsequently. To implement the *sigaction* system call, we take advantage of the *sigreturn* gadget to set the parameters for the system call. Because the *sigaction* system call contains four parameters and the fourth parameter is passed through the *r10* register. Since there isn't any `pop; ret` kind gadget for r10, we benefit from the *sigreturn*. At the same time, we manipulate the *rip* value in the fake *sigreturn* frame to the syscall instruction address, so that after the *sigreturn* sets all registers' value, the program continues to execute the *sigaction* system call.

Table 4 An excerpt of the gadgets we collect. We only list the most frequently used gadgets

Catagory	Gadgets
Set register	< `sigreturn` > < `pop rax; add rsp, 8; ret` > < `pop rsi; pop rdi; ret` > < `pop rdx; ret` > < `pop rcx; ret` >
Load/Store	< `mov dword ptr [rax], edi; ret` > < `mov qword ptr [rsi + 8], rax; ret` > < `mov rax, qword ptr [rax + 8]; ret` >
Arithmetic	< `add dword ptr [rcx], eax; ret` > < `adc byte ptr [r8 - 0x77], r9b; ret` > < `add dword ptr [rdx + 8], eax; ret` > < `neg rax; ret` >
Logical	< `and byte ptr [rcx], al; ret` > < `xor byte ptr [rax - 0x77], cl; ret` > < `not rax; add rsp, 8; ret` >
Branching	< `sigreturn` >
System Call	< `syscall` > instr

After setting the calling convention for the *sigaction* system call and padding the right number of return gadgets, we use the normal `pop; ret` gadget to manipulate the parameters for the next two system calls (*dup2*). Finally, by using the `mov qword ptr [rsi + 8], rax ; ret` gadget, we write into memory the shell file path. We make use of the `pop; ret` gadgets to set parameters for *exec* system call. To this end, the attack accomplishes.

Solution 2 To register a signal handler for a normal signal and achieve the final goal, we must take into consideration both the signal sending process's ROP chain and the signal receiving process's ROP chain. The ROP chains are more complex.

The signal sending ROP chain consists of setting variables on shared memory, monitoring the shared variables value change, and sending the signal. The semantic is when the shared variable is changed by the signal receiving ROP chain, the signal sending ROP chain detects the change and sends the signal. It is an inter-process communication.

In the first place, we utilize the `mov qword ptr [rsi + 8], rax; ret` gadget to write variables on shared memory. Additionally, We leverage lots of gadgets to realize a conditional jump to monitor the shared variable value change. Last but not least, we use `pop; ret` gadget and syscall instruction to implement a *kill* system call.

We will further illustrate how we use the gadgets we found to accomplish the second step mentioned above. We first utilize the `mov eax, dword ptr [rax + 8]; ret` gadget to read the shared variable value. Then we execute the `neg rax; ret` gadget and use the `mov qword ptr [rsi + 8], rax; ret` to move the value into memory. We take advantage of the `and byte ptr [rcx], al; ret` gadget to test whether the value change. Finally, by using `mov eax, dword ptr [rax + 8]; ret`, `mov qword ptr [rsi + 8], rax; ret`, `add dword ptr [rax-0x7d], ecx; ret` gadgets and *sigreturn* gadget, we adjust the *rsp* register value to influence

the control flow. The control flow either jumps to execute the loop or goes to execute the aforementioned step three, sending the signal.

The signal receiving ROP chain is similar to the chain in the first solution, except that it changes the value of the shared variables before executing each of the system calls. Also, it executes some loops to wait for the right moment to initiate the system call. The reason is there may be some delay from the signal receiving process that changes the shared variable to the signal sending process detects it and sends the signal. We take advantage of lots of gadgets to realize the loop. There is no sensible distinction between the gadgets we used to implement the loop and those we used to realize the shared variable monitoring. Still, there are some additional gadgets in use, such as `adc byte ptr [r8 - 0x77], r9b; ret`, `pop rdx;ret`.

5.2 Security analysis

Our approach can defeat existing defenses including ASLR/fine-grained ASLR, XOM, stack canary, and methods that hide or encrypt pointers to library functions [10]. First, since we find all the gadgets in memory on-the-fly, the ASLR and fine-grained ASLR protection can not prevent the SeBROP attack. Also, we infer the gadgets using blind execution without reading the code, so we can easily circumvent the XOM defense mechanism. We leverage the stack speculation to leak the stack canary. Finally, our attack does not require any gadgets or functions in libraries. Hence, the defense mechanisms that hide or encrypt code pointers for libraries can not thwart the SeBROP attack.

5.3 Performance analysis

The attack in Apache can complete within less than 2,100 requests or 20 minutes. The attack in ProFTPD can accomplish within less than 4,300 requests or 30 minutes. The attack in Nginx can achieve in less than 8,500 requests or 45 minutes. Most of the time and requests are spent on finding all types of gadgets. Comparing with previous research works [23] which requires an average of 32,768 requests to break ASLR, SeBROP only needs less than 2,100 requests and can defeat both fine-grained ASLR and XOM. So the SeBROP performance is acceptable. Compared to the BROP attack, SeBROP also does not introduce much overhead. The BROP attack can complete an attack in less than 4,000 requests or 20 minutes, while SeBROP can complete an attack in less than 8,500 requests or 45 minutes. However, the SeBROP attack blindly fingerprints far more gadgets than the original BROP attack. The more requests in SeBROP is spent on blindly finding all types of gadgets that construct a Turing-completed set, while the BROP attack merely finds gadgets that implement a *write* system call by blind execution. Since the SeBROP gadget searching approach is efficient, the SeBROP attack time does not grow exponentially when taking into consideration the large number of gadgets it finds. Thus, SeBROP attack performance is acceptable.

Table 5 Cumulative number of requests per attack phase

Attack Phase	Nginx	ProFTPD	Apache
Stack Reading	710	0	0
Find basic gadgets	2280	1320	990
Find shared memory	2449	1490	1027
Find all *ret* instruction	5859	3087	1844
Fingerprint all gadgets	8390	4250	2066
Launch a shell	8395	4256	2068

Accuracy Since we find all types of gadget by inferring, it is important to evaluate its accuracy. We need to validate whether the identified code location holds the correct gadget, and whether the details of a gadget are consistent with the inferred information. In our 32-bit Apache attack, we find a total of 20 gadgets, including 2 *add* gadgets, 1 *adc* gadget, 5 *mov* gadgets, 5 *xor* gadgets, 3 *or* gadgets, 1 *and* gadgets, 2 *rol* gadgets, 1 *not* gadget. The number of gadgets we find in Apache are restricted by the size of the binary and the filtering of characters. In our 32-bit ProFTPD attack, we find 59 gadgets, including 15 *add* gadgets, 2 *adc* gadget, 31 *mov* gadgets, 3 *xor* gadgets, 3 *or* gadgets, 1 *and* gadgets, 2 *rol* gadgets, 1 *not* gadget and 1 *neg* gadgets in summary. In the

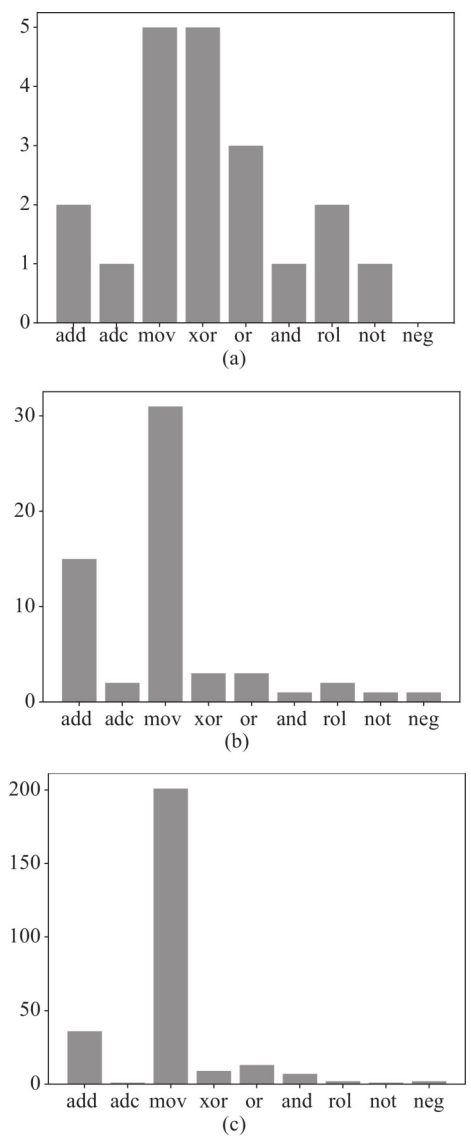

Fig. 7 Gadgets found in Apache, ProFTPD and Nginx. (a) Apache; (b) ProFTPD; (c) Nginx

64-bit Nginx experiments, we find totally 272 gadgets, including 36 *add* gadgets, 1 *adc* gadget, 201 *mov* gadgets, 9 *xor* gadgets, 13 *and* gadgets, 7 *or* gadgets, 2 *rol* gadgets, 1 *not* gadget and 2 *neg* gadgets. In both the 32-bit and the 64-bit experiments, the gadgets we find by leveraging the SeBROP approach are accurate in all details, including the operator, the operand and the immediate offset.

Stability When we use our solution I to implement the attack, we cannot achieve a one-time success, because there is a race condition. Although we synchronize the signal sending process and the signal receiving process, it is still possible that the signal does not arrive on time. Because the period of the system call is short. As a result, we have to try several times (average 30 times) to finally win the race condition and get a remote shell. But our solution II can achieve a high possibility that successfully hijack the signal control flow.

Reducing crashes Once an attack produces thousands of crashes, it is easy to be detected by the administrator. We provide a solution to alleviate this issue. The crash is usually caused by invalid memory accesses. The kernel will send the *SIGSEGV* signal to the process, so we change the *SIGSEGV* signal handler to redirect the control flow to execute an ROP chain on the stack. The chain implements a *kill* system call that will kill the program. This method can be used early in the attack when we obtain enough gadgets to implement a *sigaction* system call. This can help us significantly reduce the crashes caused by the attack.

6 Discussion

In this section, we will conduct some more discussions on the SeBROP attack. We will demonstrate which vulnerabilities give SeBROP attackers some maneuver space to accomplish the attack. We also analyze how much human efforts are involved in our attack, and discuss the limitations of SeBROP. We also show the applicability of SeBROP on other platforms and programs.

SeBROP applicability to other programs and vulnerabilities. For ease of description and implementation, in experiments, we focus on the cases that are eminently exploitable by our attack. So we select the vulnerabilities based on the two following principles. First, we select stack buffer overflow vulnerabilities in server-side programs. Since our attack is a remote attack that hijacks the control flow by overwriting the return address. Hence, a stack buffer overflow is more suitable. In SeBROP, we frequently crash the program to test a gadget's location, so a server-side program that can automatically restart meets our requirement. Second, the vulnerabilities should not limit the overflow length. They must be able to consume overflow bytes of arbitrary length, due to the fact that the final payload in our attack is relatively large. So the vulnerabilities we have chosen are all server-side programs and have no constraints on the number of overflow bytes.

Except for the three vulnerabilities, we investigate that most of the server-side program's buffer overflow vulnerability can also work as our attack targets, such as the *CVE-2002-0657* in OpenSSL, the *CVE-2001-0820* in ghttpd and *CVE-1999-1457* in thttpd. We revisit the PoC code of these vulnerabilities and find that a prototype of our attack can be implemented by performing some minor modifications to the PoC code. For instance, for the *CVE-2001-0820* in ghttpd, we can pass a long argument to the vulnerable Log function and then trigger an overflow to the stack buffer. The overflow bytes can be of arbitrary size. They are all the best cases to illustrate the wild applicability of our attack.

It is worth noting that our attack targets are not limited to server-side stack overflow vulnerabilities. SeBROP can be applied to more general scenarios.

In addition to the stack buffer overflow vulnerabilities, we can also make use of a heap buffer overflow vulnerability. We just need to find a stack pivot gadget. Then we use the gadget to jump to an attacker-controlled buffer. It then works like a stack. The remaining steps are the same as those in our stack-based attacks.

We can also take advantage of a format string vulnerability. This vulnerability can be exploited to implement arbitrary read and write primitives. We convert it into a stack buffer overflow to further hijack the control flow. We first send the overflow bytes into the program memory through user input. Then we use the arbitrary read primitive to leak the canary and user input buffer and the *memcpy* function address on the GOT table. Finally, we employ the arbitrary write primitive to set *memcpy* address and parameters to the return address on the stack (a *return-into-libc* attack). Once the program returns, it will execute *memcpy*. The *memcpy* function will copy the overflow bytes onto the stack. The program will begin to execute the ROP chain upon the *memcpy* returns. To this end, we can continue to mount a normal SeBROP attack.

Possibility of attack automation. As a variant of ROP attack, our attack procedure can be divided into gadget searching step and gadget stitching step. We will discuss whether these steps can be achieved automatically. Specifically, the gadget searching step can be automatically completed. The process of searching for gadgets presents no essential difference among victim programs. This part of attack scripts is always the same between different victims when it applies to the same architecture such as x86. We can adjust it to a new vulnerable program with very little modification.

In general though, the vulnerability triggering step is different between victim programs, the gadget gathering step is the same. After we are able to overwrite the return address, the following steps can be automatic. We begin with finding stop gadgets and pop gadgets. Then we place a counterfeit signal frame on the stack to test and identify more gadgets. These do not need any human interventions.

Nonetheless, the gadget stitching step can be semi-automatically completed. Depending on the gadgets we found, there might be some human efforts involved. The reasons are as follows.

First, different programs contain different gadgets set. We should manually choose which gadgets to use to simplify the ROP chain.

Second, the ROP chain length and the parameters set in the chain vary according to different victims and attack stages. We should manually compute the ROP chain length. Since we use *sigreturn* gadget to pass values to all registers, we should

carefully calculate the value passed to *rsp* register to make the program returning to the right position on the stack. These values are dynamically determined by the gadgets we use. Every time we send an ROP chain containing *sigreturn* gadget, we need to guarantee the *rsp* register is set to the right value.

Third, to launch our final attack in solution I (register signal handler for normal signals), the ROP chain cannot be decided beforehand. We should repeatedly test the loop counter we place onto the ROP chain to improve the hit rate. When we register a signal other than SIGSEGV, the signal receiving process must wait for the right time to launch a system call (in the period of a signal arrival). So it circulates on a loop counter. The value of the loop counter differs in victim programs. Hence we need to repeatedly test different values to find the appropriate value. Except for the loop counter, we also pad several return gadgets on the stack to control time in finer granularity. All of these will change the ROP chain length, and will in turn influence the parameters set in the ROP chain. After many trials, the ROP chain can be finally determined.

This step can be partly automatic. Because it encounters various uncertain values. It is difficult to determine these values before the attack. But we believe as our attack script becomes more and more complex, it will be able to handle various circumstances and overcome the above-mentioned issues. We will take it as our future work.

6.1 Generality and limitation of SeBROP attack

The limitation of the SeBROP attack is, it can not be applied in Windows systems. Because Windows lacks a fork-like API, the canary and text segment will be rerandomized after a crash. Also, the Windows system does not implement POSIX signals. However, SeBROP needs to leverage the signal return and signal process mechanism to accomplish the attack.

Despite the inapplicability of SeBROP in Windows, other systems, like iOS or Mac OS X, which support signal handling can be our target platform.

As a whole, the SeBROP attack is independent of the underlying architecture. No matter which instruction set the system uses, x86, arm, or PowerPC, our attack can be successfully conducted by modifying the attack script. This feature is much the same as the original ROP attack.

Meanwhile, the SeBROP attack can also support local attacks. Although our implementations are all remote attacks, browsers like chrome and ChakraCore, can be our target too. We can use JavaScript to construct vulnerable objects and send the exploit to test whether the program (JavaScript engine) crashes.

7 SeBROP prevention

In this section, we discuss how to defend our SeBROP attack. Defense methods include re-randomization, control-flow integrity, and code pointer integrity.

7.1 Rerandomization

The first protection against our attack is to re-randomize canaries [24] and code layout [25] as often as possible. If the code layout changes periodically, the gadgets that we find could become useless. The re-randomization technique is effective. However, re-randomization techniques are not deployed widely in the existing software system due to three reasons [26]: (a) re-randomization techniques have high-performance overhead; (b) re-randomization can not provide common application binary features, such as self-referencing code; (c) the whole code segment can not be re-randomized. For example, Shuffler [25], a recently proposed defense method, is unable to re-randomize the loader library.

Another method is to re-randomize the stack canary and the ASLR after a process crash. The defense makes the program randomizes every child processes independently [27], so the information leaked by attackers from one child process can not be used in another one. This method has not been widely deployed because this approach prohibits resource sharing between the parent process and its child processes [28].

Re-randomizing the code segment can effectively prevent our SeBROP attack, no matter it is re-randomized periodically or re-randomized after a crash. If the code segment is re-randomized periodically, it introduces a deadline to attackers. Because our attack spends some time finding gadgets, we cannot complete the attack during a short time period. In addition, if the code or canary is re-randomized after a crash, the gadgets that we found in one process become useless. We require these child processes to share the same memory layout. Thus our attack can be thwarted if the memory layout or canary changes.

7.2 Control-flow integrity

Another defense mechanism is control-flow integrity [29]. CFI prevents code-reuse attacks by enforcing the control flow graph (CFG) during program execution. However, a CFG construction requires massive pointer analysis. Moreover, current CFI techniques have a non-negligible performance cost. More recently hardware-assisted CFI (e.g., HCFI [30]) has been proposed to improve the performance. However, these approaches rely on specific hardware architecture modifications. Besides, researchers have proposed some coarse-grained CFI methods [31–33].

The CFI defense thwarts our attack by detecting unintended control flow transfer. When we hijack the control flow in the return address and try to execute other instructions, the CFI defense detects that the location is an invalid return location. Then it prohibits unintended execution flow.

7.3 Code-pointer integrity

A newly proposed defense mechanism called Code-pointer integrity [34] can guarantee the integrity of all code pointers (e.g., function pointers, saved return addresses) in the program. Code-pointer integrity techniques can prevent all control-flow hijacking attacks, including ROP attacks. This defense mitigates our attack by protecting the return address. When we hijack the control flow in the return address, it will detect that the return address value is modified by attackers.

8 Conclusion

In this paper, we propose the SeBROP attack, which blindly finds a Turing-complete set of gadgets without any pre-knowledge of the victim program or reading/disassembling the

code segment. We collect various types of gadgets by blind execution. Moreover, we stitch a sequence of system calls by leveraging the vulnerable signal handling mechanism. Our attack can successfully defeat the state-of-the-art defense mechanisms (e.g., fine-grained ASLR, DEP, XOM). It is compatible with both 64-bit and 32-bit systems. We implement all of our attack primitives into three programs. The complete attack is capable of spawning a remote shell on Nginx (64-bit) with less than 8,500 requests, ProFTPD (32-bit) with less than 4,300 requests, and Apache (32-bit) with less than 2,100 requests.

Acknowledgements We thank the FCS editor and all the anonymous reviewers for their constructive comments on this paper. We also thank all people that help refine this work.

References

1. Roemer R, Buchanan E, Shacham H, Savage S. Return-oriented programming: systems, languages, and applications. ACM Transactions on Information and System Security, 2012, 15(1): 2:1–2:34
2. Whitehouse, Ollie. An analysis of address space layout randomization on windows vista. Symantec Advanced Threat Research, 2007, 1–14
3. Lie D, Thekkath C A, Mitchell M, Lincoln P, Boneh D, Mitchell J C, Horowitz M. Architectural support for copy and tamper resistant software. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2000, 168–177
4. Bittau A, Belay A, Mashtizadeh A J, Mazières D, Boneh D. Hacking blind. In: Proceedings of IEEE Symposium on Security and Privacy. 2014, 227–242
5. Lu K, Song C, Lee B, Chung S P, Kim T, Lee W. Aslr-guard: Stopping address space leakage for code reuse attacks. In: Proceedings of ACM Conference on Computer and Communications Security. 2015, 280–291
6. Bosman E, Bos H. Framing signals - a return to portable shellcode. In: Proceedings of IEEE Symposium on Security and Privacy. 2014, 243–258
7. Cowan C, Pu C, Maier D, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of USENIX Security Symposium. 1998, 98: 63−78
8. Kil C, Jun J, Bookholt C, Xu J, Ning P. Address space layout permutation aslp: Towards fine-grained randomization of commodity software. In: Proceedings of Annual Computer Security Applications Conference. 2006, 339–348
9. Crane S, Liebchen C, Homescu A, Davi L, Larsen P, Sadeghi A, Brunthaler S, Franz M. Readactor: practical code randomization resilient to memory disclosure. In: Proceedings of IEEE Symposium on Security and Privacy. 2015, 763–780
10. Crane S J, Volckaert S, Schuster F, Liebchen C, Larsen P, Davi L, Sadeghi A, Holz T, Sutter B D, Franz M. It's a trap: table randomization and protection against function-reuse attacks. In: Proceedings of ACM Conference on Computer and Communications Security. 2015, 243–255
11. Snow K Z, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Proceedings of IEEE Symposium on Security and Privacy. 2013, 574–588
12. Maisuradze G, Backes M, Rossow C. What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In: Proceedings of USENIX Security Symposium. 2016, 139–156
13. Bhatkar S, DuVarney D C, Sekar R. Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of USENIX Security Symposium. 2005, 255–270
14. Davi L V, Dmitrienko A, Nürnberger S, Sadeghi A. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: Proceedings of ACM Symposium on Information, Computer and Communications Security. 2013, 299–310
15. Wartell R, Mohan V, Hamlen K W, Lin Z. Binary stirring: selfrandomizing instruction addresses of legacy x86 binary code. In: Proceedings of the ACM Conference on Computer and Communications Security. 2012, 157–168
16. Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson J W. Ilr: where'd my gadgets go? In: Proceedings of IEEE Symposium on Security and Privacy. 2012, 571–585
17. Pappas V, Polychronakis M, Keromytis A D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proceedings of IEEE Symposium on Security and Privacy. 2012, 601–615
18. Backes M, Holz T, Kollenda B, Kopp. P, Nürnberger S, Pewny J. You can run but you can't read: preventing disclosure exploits in executable code. In: Proceedings of ACM Conference on Computer and Communications Security. 2014, 1342–1353
19. Backes M, Nürnberger S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proceedings of USENIX Security Symposium. 2014, 433–447
20. Zhang M, Sahita R, Liu D. executable-only-memory switch(xomswitch): Hiding your code from advanced code reuse attacks in one shot. Black Hat Asia, 2018
21. Pomonis M, Petsios T, Keromytis A D, Polychronakis M, Kemerlis V P. krˆx: Comprehensive kernel protection against just-in-time code reuse. In: Proceedings of European Conference on Computer Systems. 2017, 420–436
22. Tang A, Sethumadhavan S, Stolfo S J. Heisenbyte: thwarting memory disclosure attacks using destructive code reads. In: Proceedings of ACM Conference on Computer and Communications Security. 2015, 256–267
23. Shacham H, Page M, Pfaff B, Goh E, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In: Proceedings of ACM Conference on Computer and Communications Security. 2004, 298–307
24. Petsios T, Kemerlis V P, Polychronakis M, Keromytis A D. Dynaguard: Armoring canary-based protections against brute-force attacks. In: Proceedings of Annual Computer Security Applications Conference. 2015, 351–360
25. Williams-King D, Gobieski G, Williams-King K, Blake J P, Yuan X, Colp P, Zheng M, Kemerlis V P, Yang J, Aiello W. Shuffler: fast and deployable continuous code re-randomization. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. 2016, 367–382
26. Wang Z, Wu C, Li J, Lai Y, Zhang X, Hsu W, Cheng Y. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In: Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2017, 143–156
27. Giuffrida C, Kuijsten A, Tanenbaum A S. Enhanced operating system security through efficient and fine-grained address space randomization. In: Proceedings of USENIX Security Symposium. 2012, 475–490
28. Lu K, Lee W, Nürnberger S, Backes M. How to make aslr win the clone wars: runtime re-randomization. In: Proceedings of Annual Network and Distributed System Security Symposium. 2016
29. Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity. In: Proceedings of ACM Conference on Computer and Communications Security. 2005, 340–353
30. Christoulakis N, Christou G, Athanasopoulos E, Ioannidis S. Hcfi:

hardware-enforced control-flow integrity. In: Proceedings of ACM Conference on Data and Application Security and Privacy. 2016, 38–49
31. Pappas V, Polychronakis M, Keromytis A D. Transparent rop exploit mitigation using indirect branch tracing. In: Proceedings of USENIX Security Symposium. 2013, 447–462
32. Cheng Y, Zhou Z, Yu M, Ding X, Deng R H. Ropecker: A generic and practical approach for defending against rop attacks. In: Proceedings of Annual Network and Distributed System Security Symposium. 2014, 1–14
33. Davi L, Sadeghi A, Lehmann D, Monrose F. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: Proceedings of USENIX Security Symposium. 2014, 401–416
34. Kuznetsov V, Szekeres L, Payer M, Candea G, Sekar R, Song D. Codepointer integrity. In: The Continuing Arms Race: Code-Reuse Attacks and Defenses, Code-Pointer Integrity. Association for Computing Machinery and Morgan Claypool, 2018

Tianning Zhang received the BS degree from Nanjing University of Chinese Medicine, China in 2013. She is currently working towards the PhD degree in the Department of Computer Science and Technology at Nanjing University, China. Her research interests include software and system security.

Miao Cai received his PhD degree in computer science and technology from Nanjing University, China in 2020. He is now an assistant researcher at Hohai University, China. His research interests include operating system and memory/storage system.

Diming Zhang received his PhD degree in computer science and technology from Nanjing University, China in 2019. In 2011, he joined College of Computer Engineering, Jiangsu University of Science and Technology, China as a lecturer. His current research interests are operating system and parallel computing.

Hao Huang received the BS degree from Xiamen University, China in 1982 and the PhD degree from Nanjing University, China in 1999. He is now a professor in the Department of Computer Science and Technology at Nanjing University, China. His research interests include operating system and system security.