eSROP Attack: Leveraging Signal Handler to Implement Turing-Complete Attack Under CFI Defense

Tianning Zhang[1,2(✉)], Miao Cai[2,3,4], Diming Zhang[5], and Hao Huang[1,2]

[1] Department of Computer Science and Technology, Nanjing University, Nanjing, China
[2] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
zhangtianning128@126.com
[3] Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University, Nanjing, China
[4] School of Computer and Information, Hohai University, Nanjing, China
[5] College of Computer Engineering, Jiangsu University of Science and Technology, Nanjing, China

Abstract. Signal Return Oriented Programming (SROP) is a dangerous code reuse attack method. Recently, defense techniques have been proposed to defeat SROP attacks. In this paper, we leverage the signal nesting mechanism provided by current operating systems and propose a new variant of SROP attack called enhanced SROP (eSROP) attack. eSROP provides the ability of invoking arbitrary system calls, simulating Turing-complete computation, and even bypassing the fine-grained label-based CFI defense, without modifying the return address and instruction register in the signal frame. Because the signal returns to the interrupted instruction, the shadow stack defense can hardly detect our attack. Signal has strong flexibility which can interrupt the normal control flow. We leverage such flexibility to design a new code reuse attack. To evaluate eSROP, we perform two exploits on two real-world programs, namely Proftpd and Wu-ftpd. In our attacks, adversaries can invoke arbitrary system calls and obtain a root shell. Both attacks succeed within 10 min under strict system defense such as data execution prevention, address space layout randomization, and coarse-grained control flow integrity.

Keywords: Code reuse attack · SROP · Signal security

1 Introduction

Code reuse attacks [25] are still the major attack means nowadays. Among them, sigreturn-oriented programming [5] is a powerful attack technique. It overwrites the return address on the stack to invoke the *sigreturn* system call and prepares a counterfeit signal frame on the stack to manipulate the rip register's value.

F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 752–769, 2023.
https://doi.org/10.1007/978-3-031-25538-0_39

After the *sigreturn* function's invocation, the control flow will transfer to an arbitrary location with an arbitrary register context.

SROP attack has been greatly restricted by several proposed defenses. For example, the signal cookies [3] method uses the kernel to insert a secret value into the signal frame to detect whether the signal frame has been tampered with. This is analogous to stack canary [2].

Another method that PiCFI [24] and MCFI [23] use to mitigate SROP attacks is to inline *sigreturn* system calls into each signal handler, and make the *sigreturn* system call unreachable from other application code. As a result, attackers need to trigger real signals to execute the *sigreturn* system call. From the authors' perspective, the attackers may have to either exploit a buggy signal handler to corrupt the saved thread context or use other threads to concurrently and reliably modify the signal handling thread's saved context. And neither of them they believe is easy since signal handlers rarely have complex code and usually do not run for an extended period of time [24].

However, it is still possible for attackers to conduct attacks by leveraging signal handlers to modify the signal frame. In this paper, we deeply investigate the signal handlers of real-world programs and find that many of them can be leveraged by attackers. Although they usually do not contain normal vulnerabilities, we can leverage them by doing some minor changes, such as overwriting the GOT table entry. Hence we propose eSROP (enhanced sigreturn-oriented programming) attack. It is a more dangerous variant of SROP attack. It can survive under defenses that are proposed to prevent traditional SROP attacks, such as signal cookies [3] or control flow integrity [1].

Our attack method is as follows. We send a signal when the program is executing some security-critical instructions. The signal handler leaks the currently executed instruction address. The attacker decides whether the interrupted instruction is the target one. If it is, the attacker sends the second signal. The same signal handler responses. This time, it overwrites the signal frame on the stack. Then the signal returns to the interrupted instruction to continue execution. However, with a different register context, the instruction will do completely different work. The core idea is still overwriting the signal frame. However, we do not directly overwrite the control data, such as the *rip* register's value. We only modify other general registers' values. So the control flow is preserved, but the data flow is changed. We do not violate backward edge protection, so strong defenses such as shadow stack [6] can hardly detect our attack.

Depending on the type of interrupted instructions, there are three cases. When the interrupted instruction (interrupted by the signal) is a *syscall* instruction, the attacker can invoke an arbitrary system call instead of the original one. Because the syscall number is stored in the *rax* register, which has been manipulated by our attack.

When the interrupted instruction is a DOP gadget [16], the attacker can leverage the gadget to implement Turing-complete computation. Original DOP attack [16] requires dispatcher gadget to connect DOP gadgets. But our attack removes this pre-requisite for using the DOP gadget. We use signals to connect different gadgets and implement complex semantics.

When the interrupted instruction is the CFI verification code, the attacker can even bypass fine-grained label-based CFI protection [1]. Since the signal handler can manipulate the verification result by modifying the general registers, the program will finally be fooled to jump to an attacker-controlled area. Most attacks [10,14] on CFI leverage the implementation defect of it. However, we show that the more fatal problem of CFI is that it cannot defend against signal attacks. Because signal can cause unexpected control flow transfer.

We build two end-to-end exploits to demonstrate the concrete techniques to construct eSROP attacks. In our first Proftpd attack, we show how to attack, show how to leverage the original signal handler to obtain arbitrary system calls invocation primitive, and finally get a shell. Our second Wu-ftpd attack exhibits how to obtain a root shell when we do not directly use the original signal handler. We manipulate the program memory and register a new signal handler for usage. Both of our exploits work reliably with ASLR and DEP defenses turned on and assuming the coarse-grained CFI defense [31] in place. As a consequence, we suggest that defenders should consider the security of signal processing more seriously.

2 Background and Assumptions

2.1 ROP Attack

Return Oriented Programming [25] attack reuses code snippets in the victim program to perform malicious behavior instead of injecting malicious code. It is a kind of control flow attack which diverts the victim program's execution flow. It has been popular for decades and has lots of variants.

2.2 SROP Attack

Sigreturn Oriented Programming [5] is one of the variants of ROP attacks. It is an attack that is related to signals. It leverages the *sigreturn* system call which is invoked when the signal returns. The attacker sets up a fake signal frame on the stack and invokes *sigreturn*. When the *sigreturn* gets executed, it treats the fake signal frame as the signal interrupted context and uses it to restore all the registers' values.

2.3 Attack Assumptions

In this paper, we assume a powerful yet realistic threat model. We assume the attacker can obtain arbitrary read/write primitive through the vulnerability. In practice, many vulnerabilities equip the attacker with this kind of ability, such as CVE-2017-7184, CVE-2017-0143, CVE-2016-4117, and CVE-2015-0057. This assumption is the same as other ROP attacks [11,13,27,28]. Then we assume the attacker can send signals to the victim program. It is easy to achieve in local attacks. In this paper, we mainly discuss local attacks, but our method can also

be applied to remote attacks. We leave this as future work. This assumption is similar to other signal attacks [7,32]. Once we can send signals, we can finally obtain a root shell.

We also assume that the system is under the following protections. Data execution prevention [12] is deployed in the system. The process is running with non-executable data and non-writable code. Address Space Layout Randomization [20] is also applied in the target system. 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 segments are always mapped to the same memory addresses. At the same time, coarse-grained CFI defense [31] is in use. An indirect call or jump can go to any function's starting address. Returns should go back to any return address, i.e. an instruction following a call. When we discuss our attack to bypass label-based CFI defense [1] in Sect. 3.3, we assume fine-grained CFI [1] is in use.

Finally, we assume the program is dynamically linked and the GOT table is not under protection.

3 eSROP Attack Method

In this section, we illustrate the techniques of enhanced Sigreturn-oriented Programming attack.

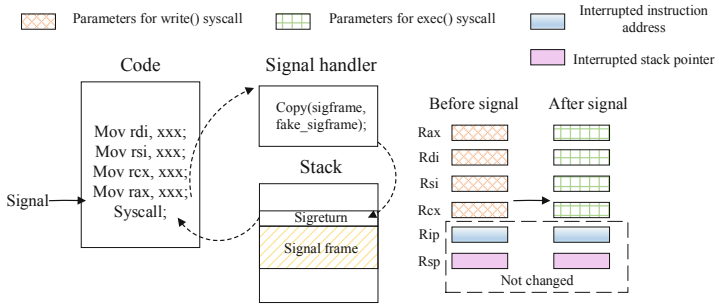

Fig. 1. Attack procedure on modifying the signal frame. When the signal returns, the program execute *exec*() system call instead of *write*() system call.

3.1 Invoke Arbitrary System Call

In the following, we will demonstrate how to invoke arbitrary system call under coarse-grained CFI defense [31].

Our attack is based on the following facts. First, in a normal program, library functions will always be invoked several times. Some of these library functions will invoke the system call. Second, signals can arrive at any time during the program's execution. If the victim process is executing in userspace (not executing kernel code), and the signal is not blocked by the process. The signal can

immediately interrupt the process, and divert the control flow to execute the corresponding signal handler. Third, signal handlers are executed in userspace. They can modify the values on the user stack. Forth, the signal frame is stored on the user stack. The signal frame is used to store the process context (all the register values) that was interrupted by the signal. when the signal returns, the process uses the values in the signal frame to restore all the registers and continues to execute the interrupted instruction.

As shown in Fig. 1, we exploit the timing of a program executing instructions right before the *syscall* instruction and right after the register setting instructions. This timing is transient but it widely exists in a program's execution. We need to capture the timing to send the signal and make the signal arrives just at the point. Then, in the signal handler, we overwrite the signal frame that was stored on the stack to modify the system call number and all the parameters reserved in general registers. When the signal returns, the program will continue to execute the *syscall* instruction. Now we successfully tamper with the system call that the program originally intended to execute and invoke an arbitrary system call.

Compared with SROP: SROP directly modifies the return address to the address of the *sigreturn* function and prepares a fake signal frame on the stack. When the vulnerable function returns, it will invoke the *sigreturn*. This is not the normal execution flow, so it can be easily detected. However, in eSROP, we leverage the signal handler to tamper with the on-stack signal frame. eSROP sends the signal and returns to the sigreturn as usual. At the same time, we do not directly overwrite the control data, such as the *rip* register value. The program returns to the point where it is stopped by the signal. The reason is if we modify the *rip* register value, it can be easily detected by other strong defenses such as shadow stack [6], which backs up the return address in a safe area. So eSROP is more stealthy than SROP attack. We do not cause abnormal control transfer during signal processing. Meanwhile, our attack is still powerful, it gives attackers the ability to invoke arbitrary system calls by only modifying the data that the program process to influence the execution result.

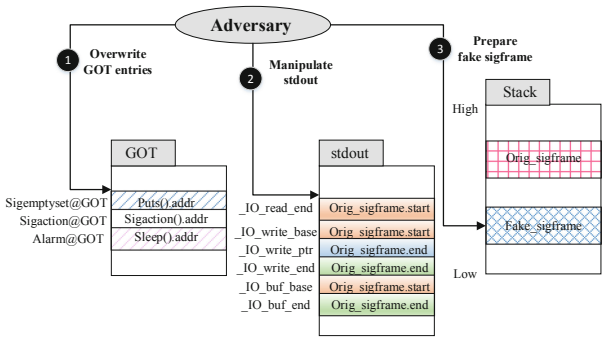

Fig. 2. Attack steps that adversary takes to leverage original signal handler. Note that, the *stdout._IO_write_ptr* is first set to *Orig_sigframe.end*. After the *puts*() function's first execution, it will automatically adjust to point to *Orig_sigframe.start*.

Details: In this attack procedure, there exist some problems to be solved. First, how to capture the timing of sending the signal. Maybe we need to implement a highly precise timer to stop the program at any instructions. Second, we need a signal handler to overwrite the signal frame which is usually not the intended work that a signal handler will do.

We solve the first problem by sending a lot of signals to the target process. Since the timing is transient and difficult to catch in one shot, we send lots of signals. After a few seconds, one of the signals will hit the target instruction. The signal handler will decide whether the instruction pointer that was interrupted by the signal equals the target. If it is, the signal handler covers the signal frame on the stack. When the signal returns, the program will execute the *exec*() system call. Finally, we will successfully obtain a shell.

To solve the second problem, people may think we must register a new signal handler to implement the semantics. The original signal handler is not designed for this purpose, so it is useless. However, when we thoroughly investigate the signal handler of many real-world programs. We surprisingly find that many of them can be leveraged by attackers, and some of them can even be directly used. This means that we need not install a new signal handler, just leverage the one that the program originally registered. It seems difficult, but we show that by exploiting the *puts*() function, we can overcome all the difficulties (as shown in Fig. 2).

In the following, we will demonstrate two approaches for two different cases. In the first case, the signal handler in the program can be directly used. In the second case, the signal handler in the program cannot be used by the attacker. We show how to conduct attacks in each of these cases.

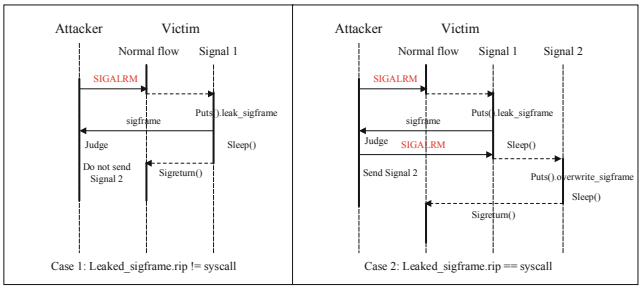

Fig. 3. Attack procedure on sending signals. The attacker decides whether to send the second signal depending on the *leaked_sigframe.rip* value.

Approach 1. In this case, the signal handler can be directly used. People may wonder why this happens. We will show that attackers can achieve this by doing some minor changes to the GOT table entries. At the same time, the signal handler must meet some conditions.

First, in the signal handler, there must be a GOT function that has at least one parameter and the first parameter is an address. The address must point to

an area that the attacker can modify. This GOT function will later be replaced by the *puts*() function. Second, in the signal handler, there must be another GOT function that has at least one parameter and the first parameter is an integer. This GOT function will later be replaced by *sleep*() or *usleep*() function.

These conditions are easy to meet. Since some normal signal handlers always re-install the same signal handler for usage the next time, they will invoke the *sigaction*() function to register the current function again. Before the *sigaction*() function's execution, the program usually prepares a *sigaction* structure and calls another function *sigemptyset*(). This function meets our first condition. It has one parameter and the parameter points to the data on the stack. The data is stored on the lower stack. It has not been used and is not initialized. Attackers can easily control the value in it.

The second condition is also easy to meet. For example, in the signal handler of *SIGALRM* signal, it is normal to invoke the *alarm*() function. This function is a library function. The program invokes it by looking up GOT table. It has one parameter and the parameter is an integer. We can modify the *alarm*() function's address in the GOT table to change it to the *sleep*() function. When the program invokes the *alarm*() function, it executes the *sleep*() function.

As a whole, we need the signal handler to execute the *puts*() function and *sleep*() function. And *puts*() is executed before *sleep*() function. The reason we choose these two functions is that we leverage the *puts*() function to first leak the signal frame on the stack. The leaked information contains the current interrupted instruction pointer, the attacker observes this value and decides whether it is the target instruction (*syscall* instruction). The *sleep*() function is used to wait for the attacker's second signal. The attacker will send the second signal to the program when he finds that the current interrupted instruction is the target. Once the second signal arrives, the same signal handler responds. This time *puts*() function overwrites the signal frame with a fake signal frame (stored in the *puts*() function's parameter). When the signal returns, the program will execute the interrupted system call. But this time, all the general registers' values have changed and the program executes a different system call (as shown in Fig. 3). More details can be found in the evaluation section.

Approach 2. In this case, the original signal handler cannot be leveraged by the attacker. So we have to register a new signal handler.

When the signal handler does not meet the requirements shown in approach 1, the attacker can turn to approach 2 to overcome the problem. In this approach, we don't make any assumptions about the original signal handler. We just do not use it and leverage *sigaction*() function to register a new signal handler. However, the *sigaction*() is usually called before the program begins handling requests. So most of the time, we cannot directly leverage the *sigaction*() function. We can leverage other frequently called functions, which have a similar parameter type and number with *sigaction*(). We modify their GOT entries and change their parameters to invoke *sigaction*() to install a new signal handler.

People may wonder whether this kind of GOT function exists. In practice, we find that it is not difficult to find this kind of function. For example, the

sigprocmask() function has similar parameters as the *sigaction*(). The value of *SIG_UNBLOCK* is *0x1*, which corresponds to the *SIGHUP* signal number. The *SIGHUP* is a signal that can be registered by users. The second parameter is usually a global variable. With the arbitrary write primitive, we can easily manipulate its value. We can construct a *sigaction* structure on it.

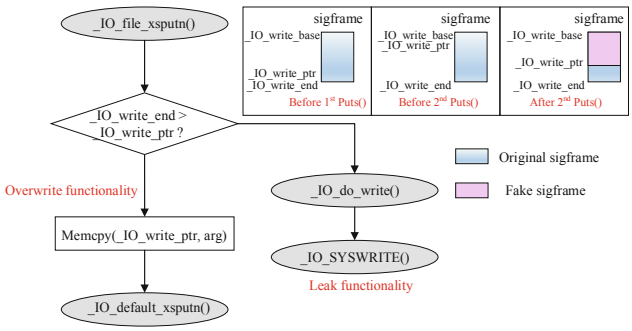

Fig. 4. Puts() function's functionality. The puts() function's first execution implements leakage, and its second execution implements overwritten.

Discussion: Here we leverage signal nesting to achieve our goal. We use the *puts*() function twice but exploits its different functionalities. For the first time, *puts*() only leaks the signal frame. For the second time, it will overwrite the signal frame. The reason why it can have different functionalities in two continuous executions is that the parameter is different in two executions and the write buffer pointer is changed automatically. For the first time, we make the *puts*() function's parameter point to some zero space, it won't copy anything into the signal frame. We precisely adjust the pointers in *stdout* object's FILE structure (as shown in Fig 4) and make it disclose the current signal frame. For the second time, we make the *puts*() function's parameter point to a fake signal frame. After the output, the *puts*() function automatically adjust the current write pointer to the start of the first signal frame. The area between *_IO_write_ptr* and *_IO_write_end* are used to buffer data. When the second time *puts*() is executed, the overwritten happens. The original signal frame is covered with fake values.

The attack shown above can bypass coarse-grained CFI [31]. First, we never overwrite any return addresses on the stack, even if the signal returns to the interrupted instruction. So we do not violate the backward edge protection. At the same time, we do not violate the forward edge protection. We do modify some control data, such as GOT entries. However, coarse-grained CFI cannot ensure only one target for indirect jumps. We find that the program invokes GOT table functions by using *jmp* instructions. This kind of instruction can have a large number of valid targets. The GOT table is dynamically updated when the program is executed. So the defense cannot differentiate GOT table functions from each other. We leverage this feature to conduct our attack.

3.2 Search and Execute DOP Gadgets

To implement more complex attack semantics, we need to use gadgets.

DOP Gadgets. As shown in the DOP [16] attack, DOP gadgets are short instruction sequences and are connected sequentially to achieve the attacker's desired functionality. DOP gadgets can implement arithmetic/logic calculation, assignment, load, and store operations. Different from original ROP gadgets, DOP gadgets require to deliver operation results with memory. We find that DOP gadgets are more suitable to be our attack targets. Because DOP gadgets need not be executed one after another. Since the DOP gadgets' operation results have been preserved in memory, the program can execute several other instructions between two adjacent DOP gadgets. This also works in our eSROP attack.

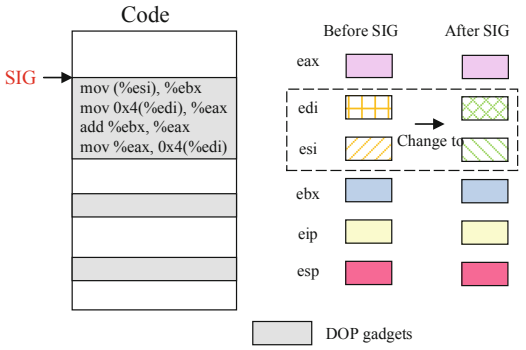

Fig. 5. Attack procedure on simulating Turing-Complete computation. The attacker can send a signal before the DOP gadget's execution and modify the data flow. When the signal returns, the gadget will do the computation on attacker-controlled data rather than the original data.

Chaining DOP Gadgets without Dispatcher Gadget. The DOP attack has investigated that the DOP gadgets widely exist in most programs. We can easily leverage them. Different from the original DOP attack, our attack removes the need for dispatcher gadgets. The dispatcher gadgets are used to chain the DOP gadgets. It equips the attacker with the ability to repeat gadget invocations. It plays a crucial role in DOP attacks. Because without it, the DOP attack fails. It also limits the DOP attack. Because dispatcher gadgets are rare and sometimes do not exist in a victim program. Even if it exists, it constrains the DOP gadgets that the attacker can use. Because only the DOP gadgets that are reachable by the dispatcher gadgets can be used. At the same time, the DOP attack leverages the vulnerability to modify data flow. This requires that the DOP gadgets are also reachable by the vulnerability.

We show that with the help of the eSROP method, all these limitations can be solved. First, we eliminate the need for dispatcher gadgets. Any DOP gadgets in the program can be used by eSROP. Second, we leverage the signal handler

to overwrite the data flow instead of using the vulnerability to modify the data flow. This is more flexible. The DOP gadgets that are not reachable by the vulnerability can also be used.

The way to implement this is similar to what we used to implement arbitrary system calls. This time, the target instruction is a DOP gadget instead of a *syscall* instruction (as shown in Fig. 5). We first leak the program code to identify DOP gadgets. Then we choose the target gadgets we want to execute. When the program executes just before the target gadget, we send the signal. In the signal handler, we modify all the general registers to prepare a new environment for the gadget. When the signal returns, the program continues to execute the DOP gadget, but now the data have been changed. The program does completely different work.

Since that, we can use all the DOP gadgets, and DOP gadgets can simulate Turing complete computation, we can conclude that eSROP is Turing complete.

3.3 Bypass Fine-Grained Label-Based CFI

Except for invoking arbitrary system calls and doing Turing complete computation, an eSROP attack can be even more dangerous. It can bypass fine-grained CFI.

The milestone paper [1] proposed by Abadi et al. uses the label-based approach to ensure the program's execution is within its control flow graph (CFG). It is a fine-grained CFI defense. Later, many other label-based approaches [4,30] have been proposed to further develop this method. But we observe that these label-based fine-grained CFI methods are vulnerable when they encounter some unpredictable events, such as signals. Because the CFG represents the intended behavior of the program, it cannot deal with unpredictable events that may happen at run time [22]. The CFG assumes that the control transfer only happens at the end of each basic block. In the real world, signals can be sent to the program at any time. A signal's arrival can cause exceptional flow transfer. This kind of control transfer is not triggered by any instruction but will interrupt the normal execution (even in the middle of a basic block) and force the control to move to the signal handler. Hence the signal handler can be used to hijack the control flow. Although the signal handler is usually small and does not contain stack overflow or other memory vulnerabilities, our eSROP attack can help to accomplish this work.

Method Description: The fine-grained label-based CFI modifies the compiled binary to insert unique IDs at the beginning of each basic block. Before each indirect branch, it inserts a few instructions to check if the destination contains the pre-computed basic block's ID. Normal control flow tampering will cause the check to fail because the destination ID will not match the label ID stored in the program. However, attacks are still possible if a signal arrives at the right moment.

The above code is used to ensure that the *jmp ecx* instruction reaches the code starting with label ID *12345678h*.

```
1 cmp [ecx], 12345678h
2 jne violation
3 lea ecx, [ecx+4]
4 jmp ecx
```

Fig. 6. Fine-grained label-based CFI verification code.

With a memory vulnerability, suppose an attacker has already tampered with the *ecx* content with a malicious code address. A signal arrives between the *cmp* and *jne* instructions, and the processor status word (*PSW*) is pushed on top of the stack in the signal frame. By leveraging our eSROP method, we can maliciously overwrite the *PSW* in the signal handler. So the *ZP* flag is reset, and when returning, the violation is bypassed and the malicious code address is reachable by the attacker.

The above attack needs to capture the timing. Once the signal does not arrive on time, CFI will detect the attack. Another more stable attack is possible. We do not modify *ecx* beforehand. We just send the signal after the violation check and before the *jmp* instruction. In the signal handler, we modify the *ecx* register's value in the signal frame. Once returns, the program will jump to our malicious code instead of the intended location. The advantage of this attack is if our signal does not arrive on time, the modification just doesn't take place, we avoid the program crash. We can wait for the next chance.

Other label-based instrumentation is also vulnerable to our attack, such as control flow locking [4].

3.4 Attack Prevention and Defense

To prevent the eSROP attack, it is important to move the signal frame to some safe area instead of storing it on the user stack. For example, Virtual Ghost [9] saves the signal frame within the VM's internal memory.

Existing techniques that prevent eSROP attack includes, CCFI [21], CPI [18] and GOT protections [17]. CCFI [21] called cryptographic CFI, uses MACs to protect control flow elements, such as return addresses, function pointers, and vtable pointers. CPI [18] is a defense that guarantees the integrity of all code pointers in a program (e.g., function pointers, saved return addresses). Both CCFI [21] and CPI [18] protect the integrity of code pointers.

Some CFI defenses that prohibit overwriting the GOT table can also prevent our attack, such as Modular CFI [23], LLVM cross-DSO [26], PiCFI [24] and uCFI [15].

GOT protection defenses prevent GOT table overwritten attacks. For example, Full Relro (Relocation Read only) [29] arranges the GOT section as read-only at program startup. However, it causes nontrivial loading overhead and does not apply to libraries. People also propose a CFI-based protection scheme [17] against the GOT overwrite attack.

However, these defenses have not been widely adopted in real systems. So eSROP attacks and GOT overwritten attacks still threaten the application security.

4 Evaluation

4.1 Experimental Setup

We conduct two experiments, Proftpd and Wu-ftpd. Both of them are in a VirtualBox virtual machine. The host machine has an Intel Core i7-10510U with 2 cores @ 1.80 GHz 2.30 GHz and 16 GB DRAM. Both experiments are in 32-bit environments. The operating system of the virtual machine is Ubuntu 16.04.

For Proftpd, we add the parameter "–with-mod_tls" when we configure the program. For the other program, we use the default settings.

In the attacks, the DEP and ASLR defenses are deployed. The programs are not compiled with position-independent code options. So the code and data sections are not randomized. However, library sections are randomized by default.

4.2 ProFTPD

The vulnerability CVE-2006-5815 in *Proftpd 1.3.0* is a stack buffer overflow. The vulnerable function is *sreplace*() function. There is an off-by-one error in this function. The attacker can leverage this error to copy a large buffer onto the stack. The buffer will overwrite the local data. It can also overwrite the return address. However, in our attack, we don't cover the return address. Some of these covered local data are security-critical, they are the parameters of the *strcpy*() function. They can implement arbitrary copy primitive. With this primitive, we can further implement our attack.

The following list is an excerpt from *Proftpd 1.3.0*. The function *sig_alarm*() is the signal handler for the *SIGALRM* signal, which is registered by the program itself.

Step 1: Modify GOT Table Entries. We modify *sigemptyset*() function's GOT table entry to change it to the *puts*() function's address. We leverage memory copy primitive to copy the *puts*() function's GOT table entry to the *sigemptyset*() function's GOT table entry. The *puts*() function will later do the memory leakage and memory overwritten for us. Then we modify *alarm*() function's GOT table entry to change it to the *sleep*() function's address. The *sleep*() function is also important in this attack because it gives the attacker enough time to decide whether to send the second signal depending on the leaked information.

Step 2: Modify stdout's Output Buffer. In our attack, the *puts*() function plays a crucial role. We use it to implement memory leakage and memory overwritten in a signal handler. To leverage the *puts*() function, we should first modify some pointers in the FILE structure of *stdout*. The *puts*() function has an output buffer, we redirect the output buffer onto the stack to point to the signal frame.

To manipulate the FILE structure, we need first modify the *_flags* field of the *stdout* object. The *_IO_write_base*, *_IO_write_ptr*, and *_IO_write_end* represent the base write pointer, current write pointer, and end write pointer respectively. The area between *_IO_write_base* and *_IO_write_ptr* is used to buffer the written

```
1  static RETSIGTYPE sig_alarm(int signo) {
2    struct sigaction act;
3
4    act.sa_handler = sig_alarm;
5    sigemptyset(&act.sa_mask);
6    act.sa_flags = 0;
7
8    #ifdef SA_INTERRUPT
9    act.sa_flags |= SA_INTERRUPT;
10   #endif
11
12   /* Install this handler for SIGALARM. */
13   sigaction(SIGALRM, &act, NULL);
14
15   #ifdef HAVE_SIGINTERRUPT
16   siginterrupt(SIGALRM, 1);
17   #endif
18
19   recvd_signal_flags |= RECEIVED_SIG_ALRM;
20   nalarms++;
21
22   /* Reset the alarm. */
23   _total_time += _current_timeout;
24   if(_current_timeout) {
25     _alarmed_time = time(NULL);
26     alarm(_current_timeout);
27   }
28 }
```

Fig. 7. Proftpd 1.3.0 code snippet.

data that has not been flushed out. We can leverage this area to leak the data. Note that in order to successfully leak the data, the *_IO_read_end* must equal to the *_IO_write_base*. The area between *_IO_write_ptr* and *_IO_write_end* is used to store the parameter data of the *puts*() function. We can manipulate these two pointers to control the overwritten.

If we want to leak and overwrite the same area, for example, the signal frame in our attack. Meanwhile, we have no chance of adjusting the buffer pointers between leakage and overwritten. The modification will be a little bit complicated. As shown in Fig 4, we first make the *_IO_write_ptr* points to the end of the signal frame. We make the *_IO_write_end* equal to the *_IO_write_ptr*. The program realizes that the current write pointer is at the end of the buffer, it will first flush out the buffer. Then it adjusts the *_IO_write_ptr* to point to *_IO_write_base* and copy new data into the buffer. This time we can leak the target object and at the same time overwrite it.

Step 3: Prepare a Fake Signal Frame. The attacker leverages the arbitrary copy primitive to write some parameters in the process memory, such as "/bin/sh". Then the attacker prepares a fake signal frame that is stored in the area pointed by the *puts*() parameter. The fake signal frame contains all the parameters and the system call number that the attacker chooses to execute. When the *puts*() function is executed, it will copy the data pointed by its parameter to the write buffer. Now the write buffer points to the real signal frame, so the overwritten can take place.

Step 4: Send the First Signal. Now the attacker can send a signal to the victim. The original signal handler responds. In it, the *puts*() function is executed instead of the *sigemptyset*() function. The *puts*() function will leak the original signal frame. Then, the *sleep*() function will be executed. The program will sleep for a few moments. In this period, the attacker can observe the output signal frame.

Step 5: Decide Whether to Send the Second Signal. There is critical data in the output signal frame, current interrupted instruction pointer. The attacker can determine whether the interrupted instruction is a *syscall* instruction. If it is, we get the point. We can send the second signal immediately. This time, the same signal handler responds. We use the *puts*() function to overwrite the signal frame. Otherwise, we do not send the second signal and wait for the *sleep*() function's completion. The signal frame won't change and we avoid possible crashes.

The signal-sending process cannot promise success in one shot. So it sends lots of signals to the program. The program's signal handler decides whether the condition meets. If the condition meets, the signal handler does the coverage. Otherwise, the signal handler doesn't make the coverage.

Our attack is implemented in a ruby script. It is invoked by the Metasploit framework. We can obtain a shell. The attack can be completed within 6 min.

4.3 Wu-ftp

The vulnerability CVE-2000-0573 is a format string vulnerability in *Wu-ftpd 2.6.0*. The format string vulnerability gives the attacker the ability to leak and overwrite arbitrary memory. The *Wu-ftpd* misuses the user input as a format string to pass it to the *vsnprintf*() function. The attacker can construct a specific format string to cover on-stack values and finally leak and overwrite arbitrary memory.

The above code is abstract from the *Wu-ftpd 2.6.0*. The attacker can leverage the program code to obtain a root shell. There is an interesting function in the code, called *ftpd_popen*(). It invokes *seteuid*(0) and *execv*() function. We leverage this function to achieve our goal.

We send the signal when the program is executing the *ftpd_popen*() function. Make sure the signal arrives before the *execv*() function's execution and after the *seteuid*(0)'s execution. When the signal arrives, we leverage the signal handler to overwrite the *execv*() function's parameter *gargv*. When the signal returns, the program continues to execute the *execv*() function. And this time, the parameters have all been changed. The attacker obtains a root shell.

Here we use our second approach. We register a new signal handler by using *sigprocmask*() function.

Step 1: Modify GOT Table Entries. We modify the GOT table entry of *sigprocmask*() function to the address of the *sigaction*(). The two functions have similar parameters. The *sigprocmask*() function's first parameter is

```
FILE *ftpd_popen(char *program, char *type,
  int closestderr)
{
  ...
  i = geteuid();
  delay_signaling();
  seteuid(0);
  setgid(getegid());
  setuid(i);
  enable_signaling();
  execv(gargv[0], gargv);
  _exit(1);
}

int enable_signaling(void)
{
  if(delaying != 0){
    delaying = 0;
      if(sigprocmask(SIG_SETMASK,
      &saved_sigmask, (sigset_t *)0) < 0){
        syslog(LOG_ERR, "sigprocmask: %m");
        return (-1);
      }
      return (0);
  }
}
```

Fig. 8. Wu-ftpd 2.6.0 code snippet.

SIG_SETMASK. Its value is *0x2*. The corresponding signal is *SIGINT*. It can be registered by the attacker. The second parameter of the *sigprocmask*() function is *saved_sigmask*. The *saved_sigmask* is a global variable. Its value can be manipulated. We can construct a *sigaction* structure on it.

Step 2: Choose a Suitable Signal Handler. Actually, we can use an arbitrary function in the program as the signal handler. However, the function must meet some conditions. First, the function must have no parameter or only one parameter and the parameter is a little integer. Because the signal handler has only one parameter which is the signal number. Second, the function must meet the requirements shown in approach 1. Here the newly registered signal handler is *pwd*(). It is a function that already exists in the victim program. It has no parameters which can match the normal signal handler. It invokes the function *getcwd*(). We modify the *getcwd*() function's GOT table entry to the address of *puts*() function.

Step 3: Prepare Fake Data. We modify the content of the *getcwd*() function's parameter "path". We change it to point to the fake *gargv* structure. The fake structure contains the "\bin\sh". We modify the *stdout* object's FILE structure and change the output buffer to point to the *execv*() function's parameter *gargv*. The *gargv* is stored on the stack.

Step 4: Trigger the Function and Send a Signal. The way to trigger *ftpd_popen*() function is by calling *NLST* command. The program will invoke *send_file_list*(), and finally invoke *ftpd_popen*() function. We send the signal when

the program is executing the *ftpd_popen*() function. When the signal arrives, the signal handler (*pwd*() function) will invoke *puts*() function instead of the *getcwd*() function. The *puts*() function's execution will cause the *execv*() function's parameter *gargv* be covered. Finally, we can obtain a root shell.

Our attack is implemented in a C program. It can also be implemented in a python script in which attackers can leverage the pwntools to assist the attack. We can obtain a root shell. Our attack can be completed in 5 min.

Security Analysis. In our attack, we do not violate the signal cookies [3] defense. Because we only partly overwrite the signal frame (we leave the *rip* and *rsp* register unchanged), we can avoid touching the signal cookie field in the signal frame. A stronger defense such as computing a hash for the signal frame may help to prevent our attack.

We also do not violate shadow stack [6], because we do not manipulate control flow elements on backward edges. Most shadow stack defenses do not protect the integrity of general registers in the signal frame, except for the PACStack [19].

5 Related Work

Since we have described eSROP-related defenses in previous sections, we only discuss eSROP-related attacks here.

The attacks that are most related to ours are the Control Flow Bending attack [8] and the Control Flow Interrupt attack [22]. In Control Flow Bending, they leverage the functions, such as *memcpy* and *printf* to do some self modifications. They show that *printf*-like functions can do Turing complete computation. In the eSROP attack, we leverage the *puts* function to both leak and overwrite the same memory space. We show that by carefully adjusting the pointers in the FILE structure, we can make the same function automatically complete two different works continuously. In Control Flow Interrupt, they show that the unexpected trigger of an interrupt and the sudden execution of an Interrupt Service Routine can circumvent CFI-based defenses. Similarly, we show that a signal can help the attacker bypass the fine-grained label-based CFI defenses. In their work, they do not show the experiments and the detailed techniques to overcome the difficulties. But we show the attack methods and experiments for real-world programs.

6 Conclusion

In this paper, we propose the eSROP attack. It is a kind of attack that leverages the vulnerable signal-handling process. We build two end-to-end exploits to show how to invoke arbitrary system calls and perform Turing Complete computations without violating DEP, ASLR, and coarse-grained CFI defenses. Both of our attacks can be completed within 10 min. The findings in this paper emphasize the importance of signal-related security. More research on defending against signal attacks is needed. We will leave it as future work.

Acknowledgement. This paper is supported by Fundamental Research Funds for the Central Universities (No. B220202073), Natural Science Foundation of Jiangsu Province (No. BK20220973), CCF-Huawei Innovation Research Plan (No. CCF2021-admin-270-202101), China Postdoctoral Science Foundation (No. 2022M711014), Jiangsu Planned Projects for Postdoctoral Research Funds (No. 2021K635C).

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inform. Syst. Secur. (TISSEC) **13**(1), 1–40 (2009)
2. Baratloo, A., Singh, N., Tsai, T.: Transparent {Run-Time} defense against {Stack-Smashing} attacks. In: 2000 USENIX Annual Technical Conference (USENIX ATC 00) (2000)
3. Bauer, S.: Srop mitigation: Signal cookies. Linux Mailing List: https://lwn.net/Articles/67486**32** (2016)
4. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 353–362 (2011)
5. Bosman, E., Bos, H.: Framing signals-a return to portable shellcode. In: 2014 IEEE Symposium on Security and Privacy, pp. 243–258. IEEE (2014)
6. Burow, N., Zhang, X., Payer, M.: Sok: Shining light on shadow stacks. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 985–999. IEEE (2019)
7. Cai, X., Gui, Y., Johnson, R.: Exploiting unix file-system races via algorithmic complexity attacks. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 27–41. IEEE (2009)
8. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: On the effectiveness of control-flow integrity. In: 24th {USENIX} Security Symposium ({USENIX} Security 15), pp. 161–176 (2015)
9. Criswell, J., Dautenhahn, N., Adve, V.: Virtual ghost: protecting applications from hostile operating systems. ACM SIGARCH Comput. Architect. News **42**(1), 81–96 (2014)
10. Evans, I., et al.: Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 901–913 (2015)
11. Farkhani, R.M., Jafari, S., Arshad, S., Robertson, W.K., Kirda, E., Okhravi, H.: On the effectiveness of type-based control flow integrity. In: Proceedings of the 34th Annual Computer Security Applications Conference (2018)
12. Gao, Y.c., Zhou, A.m., Liu, L.: Data-execution prevention technology in windows system. Information Security & Communications Privacy (2013)
13. Gawlik, R., Kollenda, B., Koppe, P., Garmany, B., Holz, T.: Enabling client-side crash-resistance to overcome diversification and information hiding. In: NDSS (2016)
14. Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: 2014 IEEE Symposium on Security and Privacy, pp. 575–589. IEEE (2014)
15. Hu, H., et al.: Enforcing unique code target property for control-flow integrity. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1470–1486 (2018)

16. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 969–986. IEEE (2016)
17. Jeong, S., Hwang, J., Kwon, H., Shin, D.: A cfi countermeasure against got overwrite attacks. IEEE Access **8**, 36267–36280 (2020). https://doi.org/10.1109/ACCESS.2020.2975037
18. Kuznetzov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: The Continuing Arms Race: Code-Reuse Attacks and Defenses, pp. 81–116 (2018)
19. Liljestrand, H., Nyman, T., Gunn, L.J., Ekberg, J.E., Asokan, N.: {PACStack}: an authenticated call stack. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 357–374 (2021)
20. Marco-Gisbert, H., Ripoll Ripoll, I.: Address space layout randomization next generation. Appl. Sci. **9**(14), 2928 (2019)
21. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: Cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 941–951 (2015)
22. Maunero, N., Prinetto, P., Roascio, G.: CFI: Control flow integrity or control flow interruption? In: 2019 IEEE East-West Design & Test Symposium (EWDTS), pp. 1–6. IEEE (2019)
23. Niu, B., Tan, G.: Modular control-flow integrity. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 577–587 (2014)
24. Niu, B., Tan, G.: Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 914–926 (2015)
25. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inform. Syst. Security (TISSEC) **15**(1), 1–34 (2012)
26. Rohlf, C.: Cross dso cfi-llvm and android (2020)
27. Rudd, R., et al.: Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In: NDSS (2017)
28. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: IEEE Symposium on Security and Privacy, pp. 745–762
29. Sidhpurwala, H.: Hardening elf binaries using relocation read-only (relro). Red Hat-We make open source technologies for the enterprise (2019)
30. Zhang, C., et al.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy, pp. 559–573. IEEE (2013)
31. Zhang, M., Sekar, R.: Control flow integrity for {COTS} binaries. In: 22nd {USENIX} Security Symposium ({USENIX} Security 13), pp. 337–352 (2013)
32. Zhou, J., Vigna, G.: Detecting attacks that exploit application-logic errors through application-level auditing. In: 20th Annual Computer Security Applications Conference, pp. 168–178. IEEE (2004)