J Comput Virol Hack Tech
DOI 10.1007/s11416-017-0299-1

ORIGINAL PAPER

Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions

AliAkbar Sadeghi[1] · Salman Niksefat[1] · Maryam Rostamipour[1]

Received: 27 January 2017 / Accepted: 26 April 2017

Abstract Return-oriented programming (ROP) and jump-oriented programming (JOP) are two well-known code-reuse attacks in which short code sequences ending in ret or jmp instructions are located and chained in a specific order to execute the attacker's desired payload. JOP, comparing to ROP, is even more effective because it can be invoked without any reliance on the ret instruction and therefore it can bypass new defense mechanisms against ROP. In this paper, we continue this line of work by proposing Pure-Call Oriented Programming (PCOP). In PCOP, we drive the control flow by proposing special gadgets that all end in a call instruction rather than ret or jmp. We then propose techniques for chaining gadgets that removes the side-effects arise from the call-ending gadgets. The idea of having call-ending gadgets with the term Call Oriented Programming has been noted in some previous work but using call gadgets in these works, due to side-effects of the call instruction, was limited to one or two call-ending gadgets between other ret/jmp gadgets. Our work is the first that shows real code-reuse attacks solely based on call gadgets. We also show that our proposed approach is Turing-complete, meaning that any functionality can be driven by PCOP. We have successfully identified some call-oriented gadgets inside GNU libc library. Our experiments with the example shellcode show the practicality of the proposed approach. Finally, we propose a variant of PCOP named TinyCOP which resists detection by recent code-reuse defense mechanisms.

Keywords Code-reuse attack · Pure-Call Oriented Programming · Return oriented programming · Jump oriented programming · Exploitation

1 Introduction

In this paper, we propose new techniques for code-reuse attacks. The main idea in code-reuse attacks is to use existing code in a process address space to execute arbitrary payloads. This is a promising approach since it can bypass the write-xor-execute(W$\bigoplus$X) defense mechanism against classic code injection attacks and also makes it harder for intrusion detection systems to detect such attacks on the network. This is due to the fact that the payload is not injected from outside and the attacker uses the existing pieces of code in the memory to launch the attack.

The basic building blocks in code-reuse attacks are gadgets. Gadgets are sequence of instructions ending in an instruction that diverts the control flow to the next gadget or some other structure that is used for chaining the gadgets. The last instruction of a gadget can be instructions like ret, jmp or call. In return-oriented-programming (ROP) [14], all gadgets are ended with the ret whereas in jump-oriented-programming (JOP) [1,3] the gadgets are ended with jmp or sometimes call instructions. In JOP, the reliance on the ret instruction has been removed to bypass detection mechanisms that are sensitive to frequent use of the ret instruction.

In this paper, we argue that the code-reuse attack can be launched by solely relying on call-ending gadgets. Using of

This work is supported by APA research center (http://apa.aut.ac.ir) at Amirkabir University of Technology, Tehran, Iran.

✉ Salman Niksefat
niksefat@aut.ac.ir

AliAkbar Sadeghi
aliakbar.sadeghi@aut.ac.ir

Maryam Rostamipour
rostamipoor@aut.ac.ir

[1] APA Research Center, Amirkabir University of Technology, No. 424, Hafez Ave, Tehran, Iran

Published online: 15 May 2017

call instruction as the last instruction of gadgets has been noted before [1,2,10], but in all previous work the call-ending gadgets are intermixed by jmp or ret gadgets and none of the previous work presented a real attack that only relies on call-ending gadgets. In this work, we propose some new types of gadgets that allow us to chain the call-ending gadgets and remove the side-effects arisen by frequent use of the call instruction.

Our contributions in this paper are as follows:

- We propose new types of gadgets that allow chaining of call-ending gadgets. Among those are the strong trampoline gadget for removing the problematic values from the stack and the kernel trapper gadget for invoking a system call by engaging minimum number of CPU registers.
- We propose two modified attack models using our proposed gadgets. An attack model based on the trampoline gadget and another model based on the dispatcher gadget.
- We propose the TinyCOP technique, which uses minimum number of gadgets to prevent detection by defensive mechanisms that relies on the number of attack gadgets.
- We show the proposed PCOP attack is Turing-complete.
- We present a real shellcode that shows the practicality of our approach.

1.1 Paper organization

The paper is organized as follows. In section 2, we provide a background on code-reuse attacks and current defense mechanisms. Next, in section 3 we present our proposed approach. In this section, we first argue why the current code-reuse techniques are ineffective in performing a PCOP attack and then we propose our new gadgets. Using the proposed gadgets, two new attack models are presented. In section 4, the TinyCOP method is presented. In section 5, we discuss the gadget discovery phase. In section 6, we show that the PCOP method is Turing complete. In section 7, we show the practicality of PCOP by showing the possibility of converting existing shellcodes to PCOP shellcodes. In section 8, the related work comes. Finally in section 9, we summarize the paper.

2 Background: code-reuse attacks and current defense mechanisms

By the advent of mitigation techniques such as DEP or write-xor-execute (W$\bigoplus$X) against buffer overflow attacks, attackers have demonstrated subtle ideas and new techniques to exploit software vulnerabilities. One such a technique is code-reuse attacks in which an adversary exploits the vulnerability by chaining small pieces of code already present in the memory of the vulnerable program instead of injecting malicious code from outside. These code-reuse attacks have instituted as so-called return-into-libc attacks [8,17] and have been later generalized to return-oriented programming (ROP) [14] and jump-oriented programing (JOP) [1,3] attacks. Since ROP and JOP are directly related to our forthcoming ideas, in this section we first take a closer look at these techniques and their underlying ideas.

2.1 Return-Oriented Programming (ROP)

Since in the primitive return-into-Libc technique [8,17] the attacker was limited to running specific functions and cannot perform arbitrary computations, a new attack called return-oriented programming was proposed [14]. The new technique, aimed at using the available machine code in the target system, uses a chain of available instructions called gadgets that all end with a ret instruction. From the attacker point of view, this technique has two steps:

1. Finding a chain of “ret”-ended instructions, called gadgets, that simulates the desired attack payload.
2. Overflowing the memory and filling it out with the start addresses of gadgets as well as needed parameters of the target system calls.

To better understand the logic behind the return oriented programming, one should first study the effects of the ret instruction on the execution of a program. In the Intel x86 architecture, upon execution of the ret instruction, at first the top element of the stack, pointed by esp, is copied into the instruction pointer. Then the esp value is incremented by 4 (bytes) to point to the next element in the stack. Considering the ROP technique, each gadget contains a limited number of instructions ended with a ret. Execution of the ret instruction at the end of each gadget, causes the gadgets to chain together and emulate the execution of the attacker desired payload. The interesting point here is that no external payload is injected by the attacker into the vulnerable program’s memory and the attack merely uses the program’s own instructions. One of the sources that an attacker can use to find gadgets is the libraries linked to the program at the run-time (dynamic libraries) or compile-time (static libraries). An example of such libraries is the libc which links to most executable files in the Linux operating system. If a ROP technique can simulate all possible payloads using the available instructions in the process address space, then it is Turing-complete. It means that every possible payload can be executed by the attacker in the vulnerable application using the program’s own instructions. The difference between this technique and Return-into-Libc is that in the latter the control flow is redirected to some functions in the C library while in the former we can return to any arbitrary instruction in the program address space.

2.2 Jump Oriented Programming (JOP)

The next state-of-the-art technique in the area of code-reuse attacks is the Jump Oriented Programming (JOP) [1,3]. The idea is to bypass the proposed defenses against ROP by avoiding the reliance on the ret op-codes and to some extent the stack. In JOP, the attacker simulates the ret op-code behavior using other arbitrary registers. For example, the gadgets can be chained with "pop reg; jmp reg;" instructions [3]. JOP has some variants and here we review the two most popular: JOP based on a trampoline gadget [3] and JOP based on a dispatcher gadget [1]. In the first variant, the ret op-codes are replaced with either pop+jmp instructions or a single jmp instruction and a trampoline gadget is used to chain the gadgets. Rest of the attack logic is similar to the ROP. For example, the gadget addresses and the system call parameters are all maintained in the stack.

To remove the reliance on the stack and to bypass defense mechanisms which monitor the stack to detect code-reuse attacks, another JOP attack was proposed which is based on a so-called dispatcher gadget [1]. In this method, the attacker is not limited to use stack or the esp register for pointing to the gadget addresses. To do this, a special dispatcher table is created that lists the addresses and other data needed for gadgets. There's also a "virtual program counter" which can be any register that points into the dispatch table. At each step, the dispatcher moves forward the virtual program counter, and executes the associated gadget. This leads to a code-reuse attack that does not rely on ret instructions and therefore can bypass countermeasures that monitor the stack or frequent ret instructions.

2.3 Defense mechanisms against code-reuse attacks

A number of defense mechanisms have been proposed to detect or prevent code-reuse attacks. ROPdefender [7] proposes an instruction monitoring based technique which uses a binary instrumentation framework to check the validity of each return address by comparing it with its corresponding value stored in a shadow return address stack. DROP [4] and DynIMA [6] monitor the execution of short instruction sequences ending with a ret. The return-less approach [3] recognizes the need of ret for the gadget construction and chaining and develops a compiler-based approach to remove the presence of the ret instruction. By removing the reliance on the ret instruction in JOP-based attacks [1,3], more elaborate techniques was proposed to detect code-reuse attacks. JOPalarm [18] uses a scoring system which is based on the distance between jmp/call instruction addresses and the jump target address. If this score exceeds the defined threshold, then JOPalarm raises an alarm. Scraps [11] works by counting the number of consecutive gadgets to detect JOP attacks. If the number of chained gadgets is more than 4, each having a maximum length of 7 instructions, SCARP raises an alarm. Ropecker [5] is another run-time defense mechanism which is activated whenever a sensitive system call is called or when an out of scope instruction is executed that is out of the defined sliding window boundaries. With the launch of the ROPecker, a "Past and Future Payload Detection" mechanism is activated which counts the number of executed gadgets using the LBR registers and the number of future gadgets by emulation. If this number is equal or more than 10, and the length of each gadget is at most 6 instructions, then ROPecker raises an alarm and terminates the process.

3 PCOP (Pure-Call Oriented Programming)

Our proposed method is based on using the call instruction and some other tricks to create the chain of gadgets. We call it Pure-Call Oriented Programming or PCOP because here the gadgets are all ended with the call instruction instead of ret or jmp instructions. To better illustrate the logic and also the challenges behind the PCOP attack, note that each call invocation has the following effects on the program flow:

1. Jump to the address specified in the call register operand.
2. Push the address of the next instruction after call into the stack.

Using call-ending gadgets in code-reuse attacks is not straightforward, because the second operation modifies the stack and makes it quite hard for the stack to be used as the start address of the gadgets. This seems to be the reason that current code-reuse exploits avoid using call-ending gadgets. Although researchers have pointed to the possibility of using call-ending gadgets [1,2,10], using these kinds of gadget is currently quite limited in exploits because of the above-mentioned side effects and we have not seen any working exploit that solely relies on call-ending gadgets.

In the following, after introducing the assumed threat model, we first show why the current known techniques used in previous code-reuse attacks [1,3] are ineffective to launch PCOP attacks. Then we proceed with our ideas to make PCOP feasible.

3.1 Threat model

In this work, like the similar code-reuse attacks, we assume the adversary can put a payload (e.g., gadget addresses or the dispatch table) and other needed attack data into memory and gain control of a number of registers, such as instruction pointer. We also assume that the vulnerable program is protected by a code integrity mechanism such W$\bigoplus$X that prevents the traditional code injection attacks to be launched.

3.2 Making PCOP gadgets using currently known techniques

To show the side-effects arisen from using call-ending gadgets in code-reuse exploits, we first discuss the possibility of using gadget chaining techniques already explained in the related papers to chain call-ending gadgets.

3.2.1 Ineffectiveness of JOP method of [3] to chain call-ending gadgets

In the JOP trampoline method of [3], a trampoline gadget is used between the functional gadgets to make the overall payload works. The trampoline gadget of [3] have a “pop + jmp” structure. However, in a PCOP attack, we want gadgets that end with a call instruction. Considering this, we analyzed the libc library to find trampoline gadgets that have a “pop+call” instruction sequence. In the Table 1, two sample trampoline gadgets extracted from the libc library are shown.

Now, let’s try chaining gadgets based on the trampoline gadgets of Table 1 which are like gadgets of [3] with the difference that they end in a call rather than a jmp instruction. In Figure 1, an assumed PCOP attack based on these trampoline gadgets is depicted.

Here we show why the trampoline gadgets of Table 1 aren’t successful in performing a PCOP attack. First, consider the gadget depicted in Table 1A. As soon as the control flow is diverted, the first gadget being executed is G1. Since the gadgets of the PCOP attack are ended in call, the last instruction of G1 is a call. Execution of this instruction results in pushing of the next instruction address after call into the stack and jumping to G2 trampoline gadget. The first instruction of the trampoline gadget G2 is a pop that should pop up the next gadget address from the stack and jump to it. But since the call instruction of the G1 gadget has modified the stack, the G2 trampoline gadget pops-up a wrong address from the stack and does not jump to the correct address of the next gadget. This results in a failure of the PCOP attack based on the trampoline gadget of the Table 1A.

Now considering the next trampoline gadget depicted in Table 1B, first note the effect of executing popad instruction on the system. This instruction pops up 32 bytes from top of the stack and stores in general-purpose registers [12]. The order of registers being filled by popad instruction is depicted in Figure 2.

Table 1 Two sample call-ending trampoline gadgets in the libc-2.19

pop eax;	popad;
cld;	cld;
call dword [eax]	call dword [eax|ecx|edx|esi]
(A)	(B)

As soon as the control flow of the process is diverted, the first gadget being executed is G1 which ends in a call instruction. Like the previous case, execution of this call pushes the address of the instruction after call into the stack and causes a jump to the G2 trampoline gadget. The first instruction of G2 is a popad that should pop-up the next gadget (G3) address from the stack and jump to it. By execution of the popad, 32 bytes are popped from the stack and stored in general-purpose registers. The first 4 bytes of these 32 bytes, stored in edi, is in fact the value that had been pushed into the stack due to the previous call invocation. This makes the edi register unusable for gadget chaining. Now assume that the attacker uses another register such as esi for gadget chaining and stores the next gadget address from the stack into this register. As the attack proceeds, with the execution of more gadgets (e.g. G4) and because of the final call instruction in each gadget, more trouble-making return addresses are pushed into the stack. This further reduces the number of usable registers.

In fact, the main limitation here is the small number of available registers to be used in a code-reuse attack. An analysis we performed on the ShellStorm data-set [13] shows that 83.5% of shellcodes need at least 4 registers eax, ebx, ecx and edx to perform the desired functionality such as executing system calls. So from the total of 8 general purpose registers, the following registers remain to be used in a trampoline-based attack: esp, ebp, esi and edi. The esp register is the stack pointer and cannot be controlled by the attacker. Also the ebp register that points to the current stack frame, cannot store any values out of the data segment addresses. Therefore, the attacker has only access to esi, edi and ebp (limited) registers to perform the attack. As we showed before, these registers are overwritten by unwanted return addresses that make them unavailable to be used in the attack. Our analysis based on the gadgets available in the libc library, reveals that a PCOP attack based on Table 1 gadgets is not feasible. In fact we need a subtle way to increase the number of available registers that can be used in the PCOP chaining and control flow management. Our proposed method of Section 3.3 solves this problem.

3.2.2 Ineffectiveness of JOP dispatcher gadget of [1] to chain call-ending gadgets

In this section, we observe the feasibility of using the dispatcher gadget of [1] to chain call-ending gadgets. The proposed idea of [1] is to use a mixture of gadgets ending in jmp/call instructions instead of ret instruction which most defensive systems are sensitive to it. The dispatcher gadget is used between the functional gadgets to chain the gadgets and perform the attack. By reviewing the libc library, we only find one dispatcher gadget ending in call (rather than jmp) that seems to be suitable for chaining gadgets (Table 2).

Fig. 1 Using the trampoline method of [3] to chain PCOP gadgets

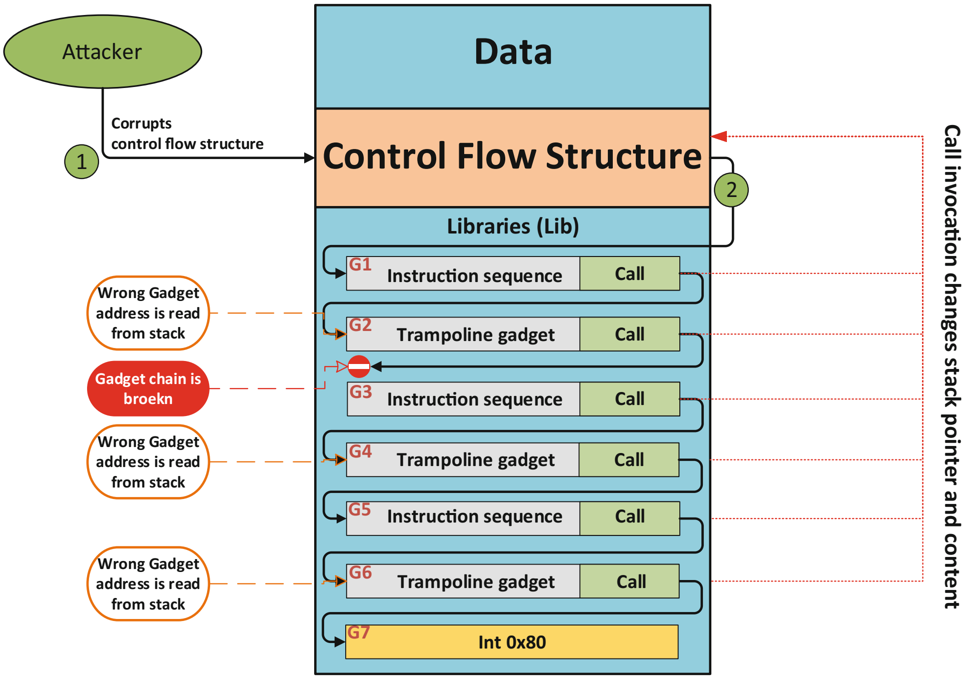


Fig. 2 The order of registers filled by popad instruction

```
|-------|   High
|  eax  |
|-------|
|  ecx  |
|-------|
|  edx  |
|-------|
|  ebx  |
|-------|
|  esp  |
|-------|
|  ebp  |
|-------|
|  esi  |
|-------|
|  edi  |
|-------|   Low
```

Table 2 A call-ending dispatcher gadget in libc-2.19 library

add eax, esi;
call dword [eax];

The design of a JOP attack based on the dispatcher gadgets is already discussed in section 2.2. Due to the lack of suitable call-ending gadgets and also inherent effects of invoking a call instruction, one cannot chain the functional and dispatcher gadgets without using any auxiliary gadgets. Note that one side effect of using the call-ending dispatcher gadgets is that we need more number of general-purpose registers to perform the attack. As we discussed in previous section, only edi, esi and ebp registers are available to do internal stuff of a code-reuse attack such as maintaining the dispatch table and program counter addresses.

Considering the dispatcher gadget of Table 2, two registers are used within the dispatcher gadget itself. Therefore, only 1 register remains for maintaining the dispatcher gadget address which is not enough for making real code-reuse attacks based on this gadget. Another limitation is that we need some specific registers for making system-calls in the shellcode. For example, the eax register should store the system call number whereas in the Table 2, eax is already used inside the dispatcher gadget. In conclusion, finding suitable call-ending gadgets and launching a practical code-reuse attack is not feasible using the currently known gadget types discussed in [1].

3.3 New gadgets and their instantiations in Linux operating system

As we discussed in previous section, the limitation on the number of available CPU registers and suitable gadgets, makes performing the PCOP attack using known techniques infeasible or at least very hard. To overcome these limitations and launch a successful PCOP attack, we propose a number of new gadgets that help in neutralizing the side-effects of call instructions and allow chaining of call-ending gadgets. In this section, we discuss the details and the instantiation of these new gadgets.

3.3.1 Strong trampoline gadget

The idea is to design a gadget that not only loads and calls the next functional gadget, but also removes the undesired return addresses pushed into the stack by previous gadgets' call instructions. To do this, this gadget should contain instructions that pop up the undesirable values from the stack. The pop and popa instructions in the x86 architecture which pop 4 and 32 bytes from the stack can help to achieve this goal. We call such a gadget, the *strong trampoline gadget*.

In a PCOP attack that consists of n functional gadgets, we need $n - 1$ strong trampoline gadgets between the functional gadgets to chain the gadgets and launch a successful attack.

Now to design an effective strong trampoline gadget, consider a general PCOP attack with 3 functional gadgets chained with 2 strong trampoline gadgets (*FG1* => *STG1* => *FG2* => *STG2* => *FG3*). Here FG stands for functional gadget and STG stands for strong trampoline gadget. The execution of the first functional gadget (*FG1*) causes the 4 bytes return address of the instruction after (*FG1*)'s final call to be pushed into the stack and jump to STG gadget. If *STG1* contains instructions "*pop x*; *pop y*; *call y*", by invocation of "*pop x*" the previous return address is loaded into register x. Then the next gadget address is popped from the stack and loaded into the register y. The "*call y*" instruction in the gadget causes the control to be diverted to *FG2* and also a new return address is pushed into the stack. Note that in this state two troublesome (from the attacker's point of view) return addresses are in the stack. The *STG2* gadget pops-up these two values into variables x and y respectively. Its final call instruction, instead of redirection to the next functional gadget, redirects the control into the next instruction after the *STG1*'s call instruction which apparently breaks the chaining of functional gadgets and causes the failure of PCOP attack.

Therefore, we observe that an ideal strong trampoline gadget should remove at least 4 bytes and at most 8 bytes from the stack to neutralize the side effects of gadgets' call instructions. For this, we need a strong trampoline gadget that contains the following instructions sequentially: "*pop x*; *pop y*; *pop z*; *call z*". If the second trampoline gadget contains these instructions, the troublesome return values are placed in x and y variables and the next functional gadget's address is placed into z and the attack proceeds. However, by analysis of the libc library, we found no gadget with such structure. Hence, we observed that the only way to overcome this problem is to use gadgets with *popa* instruction.

Note that if the *STG2* gadget contains "*popa*; *call x*" instructions, then execution of this gadget pops-up the two problematic return addresses from the stack and puts them into the edi and esi registers which makes them unavailable for PCOP's gadget chaining. Since, as we discussed earlier, only edi, esi, ebp and esp registers are available for maintaining control in the PCOP attack, there only remains the ebp register which its value should be in the range of program's data segment. These limitations makes performing PCOP attack using the above-mentioned strong trampoline gadget very hard.

Table 3 The only strong trampoline gadget in libc-2.19

0x00170ff4;	pop eax;
	popad;
	cld;
	call eax;

So to make the PCOP attack feasible, we observe that an ideal strong trampoline gadget should include "*pop x*; *pop y*; *popa*; *call z*" instruction sequence. We call it an ideal gadget because the two pop instructions before popa remove the problematic return values from the stack. This allows us to use esi, edi and ebp register to maintain the control flow and gadget chaining in a PCOP attack. Our analysis on the libc-2.19 library revealed that an exact sequence of instructions cannot be found but a similar gadget exists in this library which is depicted in Table 3.

The strong trampoline gadget depicted in Table 3 is not an ideal one in the sense that we discussed earlier, but it can remove the problematic return values from the stack. However, since this gadget has only one pop instruction, the edi register remains unusable because one of the problematic return instructions remains in the stack and is popped by popad into the edi register. So with the above-mentioned gadget, the only available registers for the PCOP's control maintenance and gadget chaining are esi and ebp registers. Designing real PCOP attacks using these registers is still hard, so we need additional techniques to increase the number of available registers. In the next section, we introduce the kernel trapper gadget that makes the eax register to be available for PCOP's gadget chaining. Using the eax register besides ebp and esi registers allows us to performa successful PCOP attack.

3.3.2 Kernel trapper gadget

The code-reuse shellcodes, depending on their functionality, need to invoke one or more system calls. To invoke a system call, first the appropriate values are loaded into the CPU registers and then int 0x80 or systenter instructions are called. In Linux one might use $call * \%gs : 0x10$ to invoke a system call too. This instruction invokes kernel_vsyscall in linux-gate.so.1 library which itself causes the invocation of int 0x80 or systenter. All the instructions in Intel x86 architecture that can transfer the control flow from the user space to the kernel space are depicted in Table 4 [14].

The shellcodes that use only one system-call may use any of the instructions listed in the Table 4 as the last instruction

Table 4 All system-call related instructions in Intel-x86 architecture

int 0x80
sysenter
call *%gs:0x10

Table 5 Gadgets for syscall invocation in shellcodes with two or more system calls

(A)	int 0x80
	ret
(B)	sysenter
	ret
(C)	call *%gs:0x10
	ret

of their payload. Others which use more system calls need a mechanism to transfer back the control to the rest of the shellcode. To achieve this, the attacker can use any of the gadgets depicted in Table 5.

In order to invoke a system call, the target syscall number should first be stored in eax, then the parameters for that specific syscall stored in other registers and finally the syscall invoked using one of the instructions depicted in Table 5. To store the target syscall number in eax, the attackers generally use gadgets which creates the appropriate value using logical or mathematical operators such as XOR, AND, ADD and SUB. Using this technique for creating the target syscall number is troublesome because it occupies at least 2 registers for storing the flag, mask and pointer to the next gadget.

To overcome this problem, our idea is to use a gadget that without using additional CPU registers, stores our intended syscall number into the eax and then invoke the desired syscall to transfer the control from the user to the kernel space. After thorough analysis of the libc library, we discovered some gadgets that store the appropriate system call number into eax register before calling the syscall instruction. The discovered gadgets do not end in ret, therefore they are appropriate for shellcodes that use one system call only. We also searched other libraries in /lib folder, but it seems that these kinds of gadgets are only found in the libc library. Since the libc library is linked against all executable files in Linux, the attacker can virtually use these gadgets in any vulnerable program to launch a PCOP attack. Since these gadgets first put the syscall number in eax register and then invoke the syscall instruction, we named these gadgets the Kernel-Trapper gadgets. In Table 6, 15 kernel-trapper gadgets are listed that we discovered in the libc library.

3.3.3 Intra stack pivot gadget

Our next proposed gadget is the intra stack pivot gadget that is another solution to handle the undesirable effects of invoking call (Push of a return address into the stack and decrease of the esp register). This new gadget increments the esp register by a fixed value and therefore it's like that no value has been pushed into the stack. The intra stack pivot gadget can be placed at any position between other gadgets in order to make a correct chaining of the gadgets.

In Table 7, a number of these gadgets that we have discovered in the libc library are shown. All of the gadgets in Table 7 are subtraction gadgets, but we need the esp value to be added. To overcome this problem, we come up with the

Table 6 Discovered kernel-trapper gadgets in the libc-2.19 library

Syscall	Gadget offset and instructions	Syscall	Gadget offset and instructions
sys_execve	b673b:mov $0xb,%eax call *%gs:0x10	sys_exit	da6c6:mov $0x1,%eax call *%gs:0x10
sys_chmod	db0ba:mov $0xf,%eax call *%gs:0x10	sys_dup2	dc0ba:mov $0x3f,%eax call *%gs:0x10
sys_read	db707:mov $0x3,%eax call *%gs:0x10	sys_unlink	dd696:mov $0xa,%eax call *%gs:0x10
sys_iopl	ec546:mov $0x6e,%eax call *%gs:0x10	sys_sethostname	e480a:mov $0x4a,%eax call *%gs:0x10
sys_rmdir	dd726:mov $0x28,%eax call *%gs:0x10	sys_setdomainname	e48ba:mov $0x79,%eax call *%gs:0x10
sys_reboot	e4cbd:mov $0x58,%eax call *%gs:0x10	sys_unmount	ec83a:mov $0x34,%eax call *%gs:0x10
sys_write	db7af:mov $0x4,%eax call *%gs:0x10	sys_sync	e4ba0:mov $0x24,%eax call *%gs:0x10
sys_kill	12c0a2:mov $0x25,%eax call *%gs:0x10		

Table 7 Discovered intra stack pivot gadgets in the libc-2.19 library

Gadget address	Instructions

0x0016aa45	sub esp, edi;
	call dword [ebp-0x3A0003D7];

0x0016a8cd	sub esp, edi;
	call dword [ebx];

0x0016a989	sub esp, eui; call esp;

Table 8 Discovered loader gadgets in the libc-2.19 library

	Gadget address	Instructions

(A)	0x000030de	popad;
		call dword [ecx];

(B)	0x00170ff5	popad;
		clc;
		call eax;

(C)	0x0016b3fc	popad;
		cld;
		call dword [esi+0x28FFFC69];

idea to use the two's complement subtraction which has the same effect as doing addition.

3.3.4 The loader gadget

The loader gadget is used to load the parameters needed in the PCOP attack from stack into the general purpose registers and it is generally the last gadget before invoking a system call. In x86 architecture, the popa and popad instructions can be used for this purpose. The popa and popad instructions pop up 8 values from the top of the stack and store them into general purpose registers. The order of storing the values into registers is as follows: edi, esi, ebp, ebx, edx, ecx and eax. Table 8, shows some PCOP loader gadgets that we have discovered in the libc library.

We should note that some conditions must be met for a loader gadget to be functional and therefore some of the above loader gadgets are inoperable in practice. These conditions are as follows:

1. The first instruction of the gadget should be popad, so the desired parameters are fetched from stack and saved into general purpose registers.
2. The gadget should not contain any instruction that modifies ebx, ecx or edx registers, so the registers are not tampered with before invoking the system call.
3. The destination register for the call's instruction should not be any of ebx, ecx, edx and edi registers, so general purpose registers are available for use in system call invocation. About the restriction on using the edi register; note that the previous gadget is ended with a call. This call instruction adds 4 undesired bytes on the top of the stack which is popped and stored into edi register when the loader gadget is executed. Therefore, the edi register is not under control of the attacker.

By the above explanations the loader gadget (A) in Table 8 cannot be used in a PCOP attack because the destination register of the call instruction is edx.

Next, we propose our ideas on using the new gadgets to design and implement real PCOP attacks.

3.4 PCOP attack 1: using strong trampoline gadget to perform PCOP

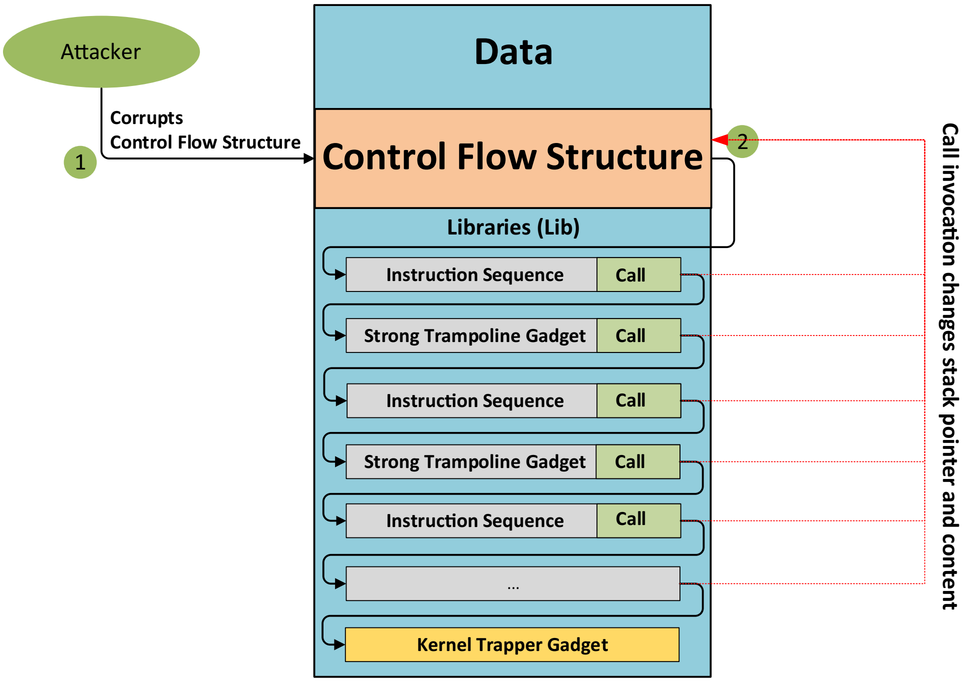
Here we propose the first PCOP attack which is based on our proposed types of gadgets: strong trampoline and kernel trapper. We discussed the details and instantiation of these gadgets in the previous section. In summary, the role of the strong trampoline gadget is to naturalize the side effects of the previous call instructions. Its functionality is the same as the normal trampoline gadget but it starts with a number of instructions which first pops the problematic values pushed into the stack by previous call invocations. The role of the kernel trapper gadget is to invoke a system-call with engaging minimum number of CPU registers.

Using these new gadgets, we can overcome the complexities arisen from call-ending gadgets and successfully launch a code-reuse attack. Figure 3 shows the design of the PCOP attack using our proposed strong trampoline and kernel trapper gadgets. We place a strong trampoline gadget after each functional gadget to neutralize the effect of previous call invocation. Finally, we end the chain with a kernel trapper gadget that invokes the desired system-call.

3.5 PCOP attack 2: using a dispatcher gadget to perform PCOP

The dispatch-table approach of [1] is beneficial in the sense that it removes the reliance on stack to some extent. In fact, the gadget addresses have not to be stored in the stack and they can be maintained elsewhere in the memory in a structure called dispatch-table. However, the attacker still needs the stack for some specific computations such as system calls. As we discussed in the section 3.2.2, it is not possible to launch a successful dispatch-table attack simply by chaining call-ending gadgets. This is due to a number of limitations including lack of available CPU registers, and modification of data and control structures (e.g. stack) as a side effect of invoking call. To overcome these limitations, we propose a new design that makes use of our proposed new gadgets.

Fig. 3 PCOP attack based on the proposed strong trampoline gadget



Using these 3 new gadgets, it is possible to launch a successful PCOP attack. Our proposed method for using dispatcher gadget in PCOP is depicted in Figure 4.

Using these new gadgets, we can overcome the complexities arisen from call-ending gadgets and successfully launch a code-reuse attack. Figure 3 shows the design of the PCOP attack using our proposed strong trampoline and kernel trapper gadgets. We place a strong trampoline gadget after each functional gadget to neutralize the effect of previous call invocation. Finally, we end the chain with a kernel trapper gadget that invokes the desired system-call.

Here the functional gadgets (instruction sequences) are chained together by the use of the dispatcher gadget. The functional gadgets' addresses are maintained in the dispatch table instead of stack. Therefore, unlike the previous model, it's not necessary to frequently remove problematic return values pushed into stack due to call invocations. However, whenever the attacker needs to load some values from the stack, or a system-call has to be invoked, the stack should be sanitized from the added return addresses. To do this, the Intra Stack Pivot gadget is used which modifies the esp register and increments it by a factor of 4 bytes so the side-effect of arbitrary number of previous calls are reversed. Then to load the desired values from the stack, the attacker can use loader or strong trampoline gadgets. If a loader gadget is used, then the call instruction of the intra-stack pivot is loaded into edi and makes this register unavailable in the rest of the stack. However, if a strong trampoline gadget is available to be used, the edi register would be available too. Finally, the strong trampoline or loader gadgets have to be used before the system call, so the required parameters for the syscall placed into the general purpose registers.

4 Tiny Call Oriented Programming (TinyCOP)

The main idea behind the TinyCOP is to use techniques to minimize the number of gadgets in a PCOP attack. The less the number of gadgets in a code-reuse attack is, the lower the probability of being caught by defensive mechanisms will be. When the number of consecutive gadgets is small, the PCOP behavior is similar to a normal program and thus it is quite hard to detect the attack based on techniques that count the number of consecutive gadgets chained together. In our proposed TinyCOP attack, the following gadgets are executed respectively: the initializer gadget as the first gadget, a number of functional gadgets to launch the attack and prepare the parameters for the final syscall and finally the kernel trapper gadget to store the target syscall number in eax and invoke the syscall.

Figure 5 depicts our proposed TinyCOP attack. By invoking the initializer gadget all the needed data in the PCOP attack are popped from the stack and stored in the general purpose registers. Then by execution of the call instruction in the initializer, the next functional gadget is invoked. We proceed by chaining a few functional gadgets with minimal side effects. Finally, by invoking the kernel-trapper gadget as the last gadget of the chain, the desired system call is executed with the proviso that all parameters are loaded correctly in the appropriate registers. It's noteworthy to say that in the

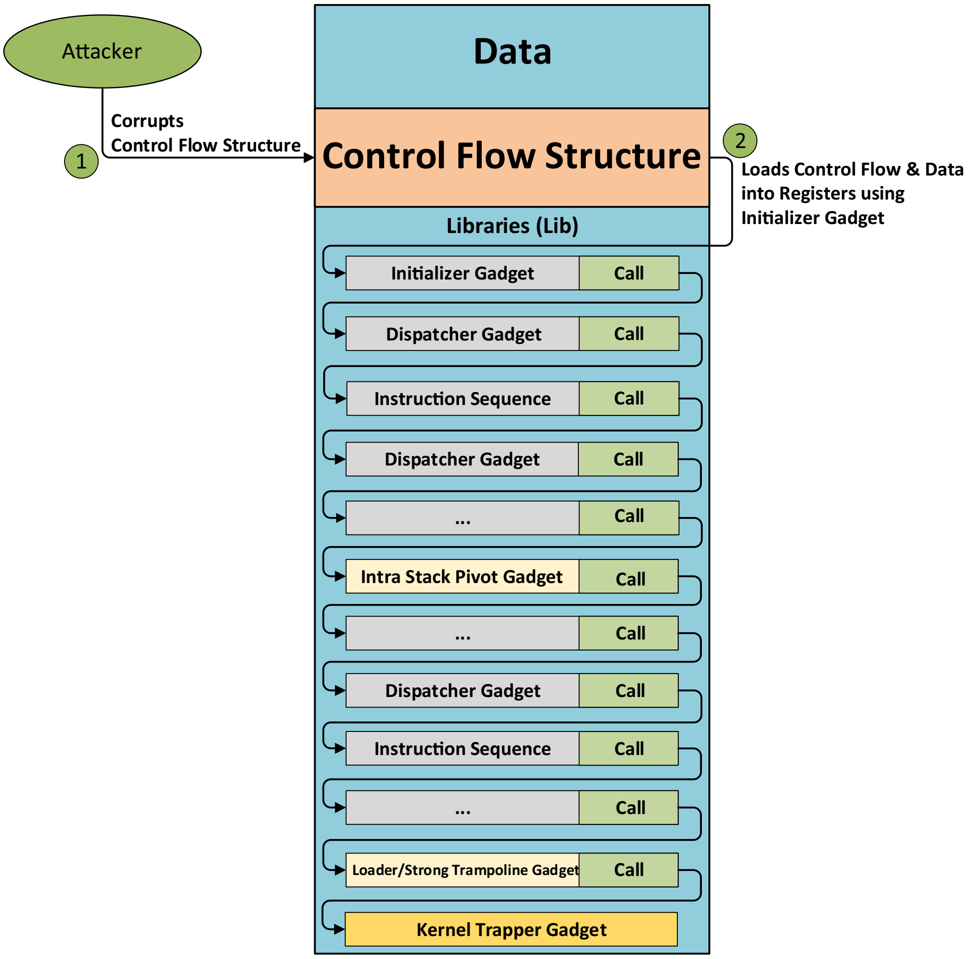


Fig. 4 PCOP attack based on the dispatcher gadgets

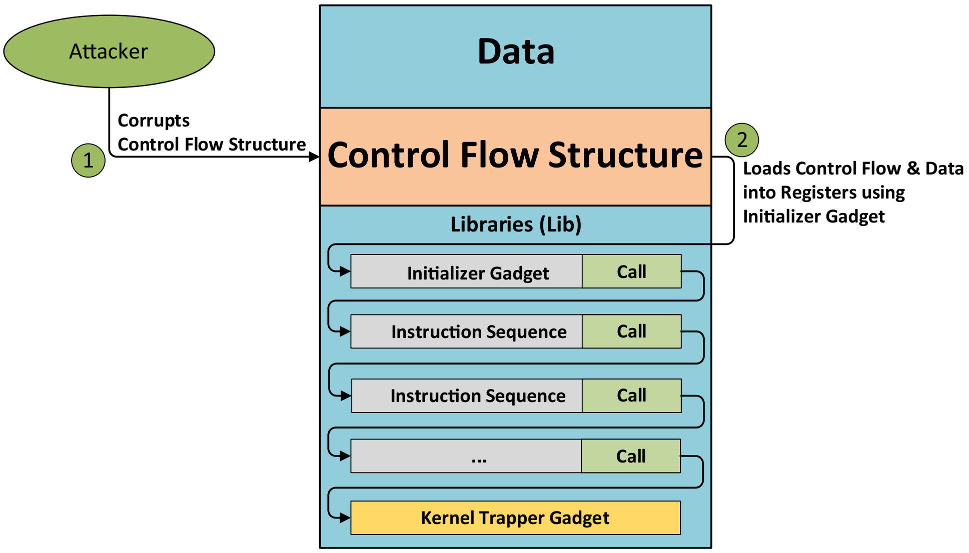


Fig. 5 The design for the proposed TinyCOP attack

TinyCOP, the attack is so minimal that we do not need to adjust the stack by removing problematic values.

4.1 A TinyCOP shell-code (spawning a Linux shell)

In this section, we present a shell code developed using the TinyCOP model. The shellcode is a typical spawn-shell that takes advantage of *execve*() system call. It's a syscall that can be used to invoke executables or scripts. The system call number for *execve*() is $0x0b$. As shown below *execve*() takes three parameters as input.

int execve(*const char* $*$ *filename*, *char* $*$ *const argv*[], *char* $*$ *const envp*[]);

filename is the name of the file that is going to be executed. *argv* passes argument strings and *envp* passes environment variables into the function. In our attack scenario, we are not going to pass any argument variable or environment string to execve().

Based on the above discussion, in our proposed shell-code, the appropriate parameters should be placed in general purpose registers as depicted in Table 9 and then the $execve()$ system call be invoked.

Our proposed shell-code is depicted in Figure 6 which only has two gadgets. Initializer is responsible to load ebx, ecx and edx with the addresses of unintended bytes of string $/bin/sh$ that are in the libc library, and two available NULL Dwords respectively. Then, the Kernel-Trapper will load eax with $0x0b$ and make a system call.

Table 9 Appropriate values of CPU registers for spawning a shell

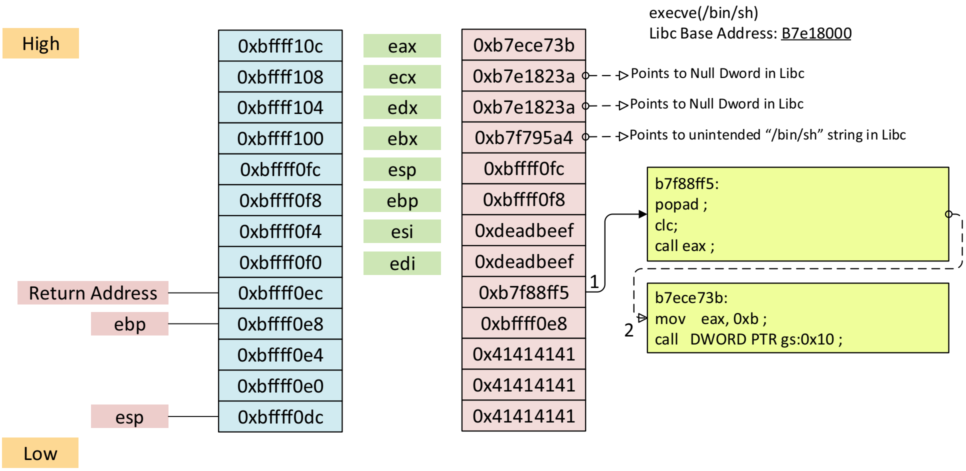
Registers	eax	& ebx	& ecx	& edx
Values	0x0b	/bin/sh	NULL	NULL

4.2 TinyCOP and current code-reuse defense mechanisms

In this section, we discuss why the current defense mechanisms against code-reuse attacks are ineffective against our proposed TinyCOP method. Since both JOP and PCOP attacks are based on frequent jumps in the code segment, the same ideas for defending against JOP can work for the PCOP attack too. Therefore, we focus on the following three defense mechanisms against JOP and discuss the behavior of TinyCOP in their presence: JOP-alarm [18], SCRAP [11] and ROPecker [5]. These systems have three properties in common. (1) All of them work under the Linux operating system. (2) They do not rely on any specific operating system features to work. (3) They do not need the program source code or binary rewriting to defend against code-reuse attacks.

Our proposed TinyCOP shellcode, depicted in Figure 6, spawns a Linux shell using the execve() system call. Its length is 36 bytes consisting of 2 gadgets and 5 assembly instructions. We begin the analysis with JOP-alarm. To detect code-reuse attacks it uses a scoring system which is based on the distance between jmp/call address and the jump target address. To do this, if the distance of jump between two gadgets is more than 4096 bytes, the score parameter is incremented by 20 and with each non-jump instructions 1 is decremented from the score parameter. Finally, if the score parameter reaches the value of 120, JOP-alarm raises an alarm. Now, considering our proposed TinyCOP shellcode, the first call instruction in the first gadget increases the score parameter to 20. Then, with the execution of the first instruction in the second gadget, the score is decremented by 1 and equals 19. Finally, by the instruction Call Dword PTR gs:0x10, the system call is executed and the Linux shell spawns without being detected by JOP-alarm. Note that upon execution of the system-call, the score is far less than the defined threshold of 120.

Fig. 6 A TinyCOP shellcode that invokes /bin/sh



```
b7f88ff5:
popad ;
clc;
call eax ;
```

```
b7ece73b:
mov    eax, 0xb ;
call   DWORD PTR gs:0x10 ;
```

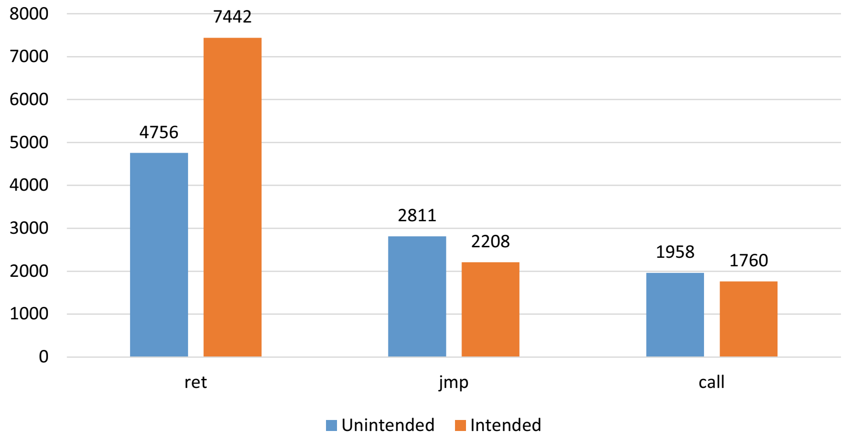


Fig. 7 The frequencies of call-ending gadgets in comparison with ret and jmp gadgets

SCRAP [11] works by counting the number of gadgets to detect JOP attacks. If the number of chained gadgets is more than 4, each having a maximum length of 7 instructions, SCARP raises an alarm. Thus our proposed TinyCOP attack with 2 gadgets and 5 instructions is undetectable by the SCARP framework.

Finally, the ROPecker detection mechanism is launched whenever a sensitive system call is called or when an out of scope instruction is executed that is out of the defined sliding window boundaries. With the launch of ROPecker, a Past and Future Payload Detection mechanism is activated which counts the number of executed gadgets using the LBR registers and the number of future gadgets by emulation. If this number is equal or more than 10, and the length of each gadget is at most 6 instructions, then ROPecker raises and alarm and terminates the process. Now considering TinyCOP, it can successfully bypass the detection by ROPecker because it only contains 2 gadgets and 5 instructions. In section IX.C of the ROPecker paper [5], it is mentioned that the ROPecker cannot detect attacks with 1 or 2 gadgets and that the attacks with 1 or 2 gadgets cannot contain any malicious activity. Our proposed TinyCOP attack shows that this claim is in fact not true.

5 Gadget discovery

PCOP needs gadgets that end in an indirect call instruction. To discover such gadgets, we used the Galileo algorithm presented in [14]. Note that every x86 binary has some unintended code sequences that can be used by jumping to an offset not on the original instruction binary [1] and the Galileo algorithm takes such gadgets into account. By a slight modification of this algorithm, we extracted all intended and unintended PCOP gadgets.

By running the Galileo algorithm on the libc-2.19 library, the total number of suitable gadgets ending in indirect call, indirect jmp and ret instructions is 20935.

Figure 7 depicts the frequencies of intended and unintended call, jmp and ret instructions in the libc-2.19 library. As seen, the number of call instructions is lower than ret and jmp instructions and this makes designing PCOP attacks harder than JOP and ROP attacks.

Next, we do a more detailed analysis on intended and unintended gadgets that end in a call instruction. The purpose of this analysis is to find which register is used more frequently as the target address of call instructions. As depicted in Figure 8, the eax register is used in 2210 gadgets as the call's target address which is more frequent than the other general purpose registers. This seems to be problematic, because if we use gadgets ending in indirect calls with eax register as the target address, then eax register would not be available for some other important tasks such as invoking system calls. Note that for invoking a system call, the system call number should be stored in eax, and therefore we should look for a technique that allows us to use eax both for gadget chaining and system calls. The kernel trapper gadget that we discuss in section 3.3.2, allows us to do this.

5.1 Gadgets on more recent targets and other libraries

Libc version 2.19 has been initially released on February 2014, however since it has been used in the long-term support (LTS) version of some major Linux distributions such as Ubuntu 14.04-LTS, it is still being maintained by Linux vendors and used widely on many Linux-based machines. As an example, the Libc-2.19's latest patch on Ubuntu 14.04 is "2.19-0ubuntu6.11" released on March 21, 2017 which is quite new. It's noteworthy to say that Ubuntu-14.04-LTS will have maintainance support till February 2019 [16] and hence the Libc-2.19 will be available on many Ubuntu-based production servers.

However, to assess the feasibility of our approach on the most recent versions of the Linux operation system, we conducted a number of more experiments. Our new experiments have been conducted on the latest LTS version of Ubuntu dis-

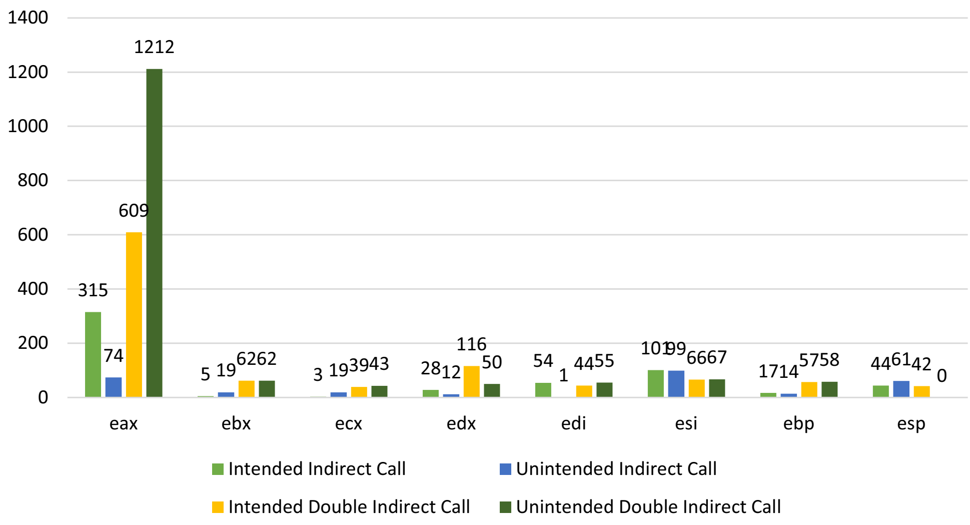


Fig. 8 Frequency of using general purpose registers as the call's target address

Table 10 Availability of PCOP gadgets in recent LTS versions of Ubuntu

	Libc-2.19 (Ubuntu 14.04 LTS)	Libc-2.23 (Ubuntu 16.04 LTS)
Strong trampoline	✓	✗
Intra stack pivot	✓	✓
Kernel trapper	✓	✓
Loader gadget	✓	✓

tribution which is Ubuntu 16.04-LTS (Released on February 2017). We first started by searching for our proposed gadgets in the Libc-2.23 library which has been used in this version of Ubuntu. As depicted in Table 10, except for the strong trampoline gadget, all other gadgets can be found in Libc-2.23.

To find strong trampoline gadgets in Ubuntu 16.04 LTS, we wrote a script to search through all available libraries in /lib folder (total of 398 libraries) and we successfully identified a number of strong trampoline gadgets in these libraries which has been depicted in Table 11. There are lots of more libraries in /usr/lib folder (around 3532 libraries) and our experiments showed that a high number of these libraries contain the strong trampoline gadget as well. A number of such libraries and their corresponding strong trampoline gadgets have been depicted in Table 12.

It's noteworthy to say that another reliable source for finding strong trampoline gadgets, is the the binary code of the vulnerable program or service.

5.2 PCOP gadgets on other operating systems

Launching a PCOP attack on mobile-based operating systems such as Android and IOS is not feasible using the proposed gadgets because of major differences in hardware architecture and instruction set. Also the kernel trapper gadgets are not available in the Microsoft's Windows family of operating systems because Windows lacks the virtual dynamic shared object capability. Therefore performing PCOP attacks on the current versions of Microsoft Windows is not feasible using the methods proposed in this paper. However, it should be noted that launching PCOP attacks in these operating systems are an interesting field for future research.

Regarding the defensive mechanisms in other operating systems, consider the recent Microsoft Control Flow Guard (CFG). Microsoft CFG is a defense mechanism in the Windows family of operating systems and it's not available on the Linux systems. However, to assess its effectiveness against the PCOP attack, assume that a similar mechanism has been

Table 11 Strong trampoline gadgets found in /lib libraries on Ubuntu-16.0.4-LTS

	Library	Gadget address + instructions
1	libslang.so.2.3.0	0x00060ab3: pop eax; popad; cld; call dword [eax-0x18];
2	libslang.so.2.3.0	0x00059c3d: popad; pop ebp; cld; call dword [eax-0x18];
3	libdevmapper.so.1.02.1	0x0004a2e5: pop ecx; add byte [eax], al; inc esp; popad; cld; call dword [eax];
4	libdevmapper.so.1.02.1	0x0004a2e4: pop esp; pop ecx; add byte [eax], al; inc esp; popad; cld; call dword [eax];

Table 12 A number of strong trampoline gadgets discovered in /usr/lib libraries on Ubuntu-16.0.4-LTS

	Library	Gadget address + instructions
1	libmergelo.so	0x02393ef2: pop edx; popad,; call dword [eax+0x6A]
2	libllvm-3.8.so.1	0x022d1505: pop ebp; popad,; add ah, bh; dec ecx; pop esi; call eax;
3	libxul.so	0x02c72cf4: pop ebx; inc dword [edi+edi*8-0x007C009B]; popad; call dword [eax-0x73009F01];

implemented in the Linux operating system. First of all, Microsoft CFG is a compiler-level protection system and hence it is needed that all binaries get recompiled for the protection to work. This may not be possible for the production systems or for applications with no access to their source codes. Secondly, Microsoft CFG is not a bullet-proof defense mechanism and researchers have proposed a number of techniques to bypass it [9,15,19]. For example, if an application uses libraries that have not been compiled with CFG linker flag, it is possible to extract suitable PCOP gadgets from such libraries and launch a successful PCOP attack. A similar technique has been used in [9] to bypass Microsoft CFG.

6 PCOP is turing-complete

An important parameter in evaluating a code-reuse method is to see if it has the ability to execute any arbitrary operations, i.e., being Turing-complete. To show that our proposed PCOP method is Turing-complete, we must show that all desired functionalities in a shellcode can be driven by chaining the proposed call-ending gadgets. These functionalities include storing and loading data into memory, arithmetic and logical operations, conditional and non-conditional jumps, and system call invocation. In this section, we show the Turing-completeness of PCOP by identifying appropriate functional gadgets solely from the Libc library which is linked to all executable files in Linux operating system. For each operation the gadget offset in the libc-2.19 library as well as its instructions is depicted.

6.1 Methodology

The proof for the Turing-completeness property, was performed based on an analysis on the Libc library (version: 2.19 size: 1718 KB) which is linked to all executable files in Linux operating system. We used rp++ tool to extract gadgets and used regular expression to search for the desired gadgets. The Intel semantic is used to express instructions (e.g. mov destination, source) and we've used semicolon to separate instructions.

Table 13 Gadgets for loading registers

Register	Gadget address + instructions
eax	0x0015a12e; pop eax; call dword [edi+0x4656EE7E];
ebx	0x0015e6e9; pop ebx; call dword [esi+0x67FFF258];
ecx	0x0018ce01; pop ecx; call dword [esi+0x00];
edx	0x0017ff08; pop edx; sub bh, ch; call dword [eax];
ebp	0x0018cebd; pop ebp; call dword [eax+0x00];
esi	0x0016b3f1; pop esi; cld; call dword [eax+0x5F];
edi	0x0018cf2d; pop edi; call dword [eax];
All general purpose registers	0x000030de; popad; call dword [ecx];
	0x00170ff4;pop eax; popad; clc; call eax;

6.2 Gadget repository

In this section, for each category of basic instructions, we present the equivalent PCOP gadgets that have been extracted solely from the libc library.

6.2.1 Loading registers

The loader gadget is used in PCOP exploits to load values form stack into CPU registers. In x86 architecture, there are 8 general purpose registers that can be used for storing and loading data. To store a value in registers from stack, the pop and popa instructions can be used. If no problematic return value is in stack, one can simply use the pop instruction; otherwise the popa instruction can be used. In the worst case, the edi and esi registers contain the problematic values. To store the desired values into this registers, we can put the values in other registers then use mov or xchg instructions to save the values into esi/edi registers. Table 13 depicts the gadgets extracted from the libc-2.19 library for loading values into various CPU registers.

6.2.2 Loading and storing from memory

In x86 architecture, the load and store instructions are used to load values from system memory into CPU registers and

Table 14 Gadgets for loading from and storing to memory

Type	Gadget address + instructions

LOAD (eax)	0x00124d50; mov eax, dword [ebp-0x54];
	call dword [edi+0x000001AC];

LOAD (edi)	0x001248d6; mov edi, dword [ebx-0x000000E0];
	call dword [edi+0x000001B4];

STORE (edx)	0x00124af1; mov dword [ebp-0x58], edx; call eax;

STORE (eax)	0x00110198; mov dword [esp+0x04], eax;
	call dword [edi+0x14];

vice versa. To use these instructions in a PCOP shellcode, we need some gadgets that use these instructions. In Table 14, a number of such gadgets that we extracted from libc-2.19 library is depicted.

As an example, for loading values from memory into eax register, we found a gadget that loads the memory address pointed by (ebp-0x54) in the eax. To use this gadget the attacker has to only store the appropriate address in the ebp. Moreover, for storing values into the system memory a number of gadgets has been discovered. For example by storing an appropriate address in ebp, the value of edx can be stored in the desired address.

6.2.3 Arithmetic gadgets

The gadgets for performing arithmetic operations are listed in Table 15. The basic arithmetic operations are add and sub instructions. These instructions can do addition and subtraction in three forms. In the first case, the operation is performed on two CPU registers specified as instruction operands. In the second case, one of the source or destination operands are in memory and in the third case one of the operands are a static value and the second operand is either a register or a memory address. For each of these cases, Table 15 depicts a suitable gadget. The gadgets for other instructions like inc, dec and neg are also depicted in this table.

6.2.4 Logical gadgets

The instructions AND, OR, XOR and NOT perform the primitive logical operations in x86 platform. In order for a PCOP shellcode to use these operations, we need appropriate gadgets that include these instructions. A number of such gadgets are listed in Table 16. The AND, OR and XOR gadgets take their operand values from register or memory addresses and store the result into a register or memory address. One of the widely used instructions for zeroing the memory is XOR. When the source and destination of this instruction is equal, the result would be zero. In Table 16, we have listed a gadget for zeroing the value in esi register. This value can be stored in other registers as well as the memory by taking advantage of other gadgets. Moreover for doing 1'completement (NOT) of a value, one can use the NOT instruction as depicted in Table 16.

Table 15 Gadgets for arithmetic operations

Type	Gadget address + instructions
ADD	0x0019ef79; add esi, edi; call dword [eax];

ADD	0x0015e9f4; add dword [edi-0x0E],ebp;
	call dword [ecx+0x6D];

ADD immediate	0x0000370b; add eax, 0x572808A8;
	call dword [esi+0x6F];

SUB	0x0019ff15; sub edi, esi; call dword [ebp+0x00];

SUB	0x0018cf2c; sub dword [edi-0x0E], ebx;
	call dword [eax];

SUB immediate	0x0016acd5; sub eax, 0x2D65FFFC; cld;
	call dword [ebp-0x2A0003D3];

INC	0x0018ee69; inc ebx; call dword [eax+0x00];
DEC	0x0015ac3a; dec ebp; call dword [esi+0x30];

NEG	0x00073d55; neg edx; mov dword [esp], edx;
	call dword [eax];

NEG	0x0019bcb0; neg dword [eax-0x0A];
	call dword [eax];

Table 16 Gadgets for logical operations

Type	Gadget address + instructions
AND	0x00180c49; and edi, ebp; call dword [edx];
AND	0x0015db28; and dword [ebx-0x10], ecx; call edx;
OR	0x0015cd78; or ebx, esi; call dword [eax];

OR	0x0018cf5c; or dword [edi-0x0E], ebx;
	call dword [eax];

XOR	0x0019a9c5; xor esi, esi; call dword [eax];
XOR	0x00171955; xor ecx, edi; call dword [eax-0x13];

XOR	0x000d4b1b; xor dword [ebx-0x01], ecx;
	call dword [eax-0x0008CC17];

NOT	0x0016e60c; not ecx; call dword [eax+0x03000152];

6.2.5 Branching gadgets

Since in code-reuse attacks, the gadget addresses are stored in stack a branch is done by changing the stack pointer [3]. This is contrary to a normal program which a branch is done by changing the instruction pointer to an absolute or relative address. In Table 17, an unconditional jump gadget is depicted which modifies the esp value by performing a subtraction or two's complement addition. Then using gadgets

Table 17 Gadgets for branching

Type	Gadget a ddress + instructions
Unconditional	0x0016a8cd; sub esp, edi; call dword [ebx];
Conditional	0x00171955; xor ecx, edi; call dword [eax-0x13];
	0x00170a66; clc; call dword [eax];
	0x001947dd; cmp esi, ebp;call dword [ecx+eax+0x00];
	0x0015ce51;sbb esi, ebx; call dword [eax];
	0x00117b9b;mov dword [esp], esi;call dword [eax+0x04];

like strong trampoline or loader, the next gadget address is read from the stack and the control flow is diverted to it. Therefore, with combining an unconditional and loader gadget, the unconditional operation can be done.

Another type of jump is conditional jump. The x86 architecture contains a number of instructions for conditional jump, but since these instructions modifies the eip register instead of esp register, they are not suitable to be used in code-reuse exploits. We take the same strategy as in [3] that changes the stack pointer conditioned on the word stored in memory at a known address. Our proposed branching gadgets are depicted in Table 17.

7 Experiments with converting classic shellcodes into PCOP shellcodes

In this section, we show the practicality of PCOP by showing the feasibility of converting the shellcodes available in Shell-Storm library [13] into their equivalent code-reuse PCOP shellcodes. Till 28 July 2016, the Shell-Storm library contains 247 shellcodes. As shown in Figure 9, 200 of these shellcodes are plaintext and the rest are polymorphic. Our evaluation is based on these 200 plaintext shellcodes that have different sizes and functions. The most important factor for converting these shellcodes into their equivalent PCOP shellcodes is the number of used syscalls. This is important since our proposed TinyCOP method only supports shellcodes with one system call.

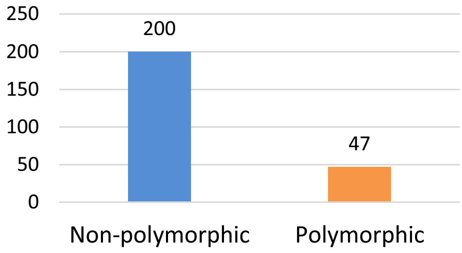


Fig. 9 The frequency of shellcodes in Shell-Storm library on July 2016

By disassembling non-polymorphic (plain) shellcodes, we counted the number of system-calls in each shellcode. As shown in Figure 10, 68 shellcodes invoke only 1 system call, and 53 shellcodes invoke 2 system-calls. The maximum number of system calls invoked in the analyzed shellcodes is 14. According to the high number of shellcodes with 1 system-call only (68), the proposed TinyCOP method seems to be quite practical, since it can be potentially used to convert these shellcodes into their equivalent PCOP shellcodes.

This allows us to convert 67 out of 68 shellcodes with 1 system-call, into their equivalent PCOP shellcodes [1]. Therefore, this evaluation shows that 42% of the Shell-storm library shellcodes can be used in a PCOP attack.

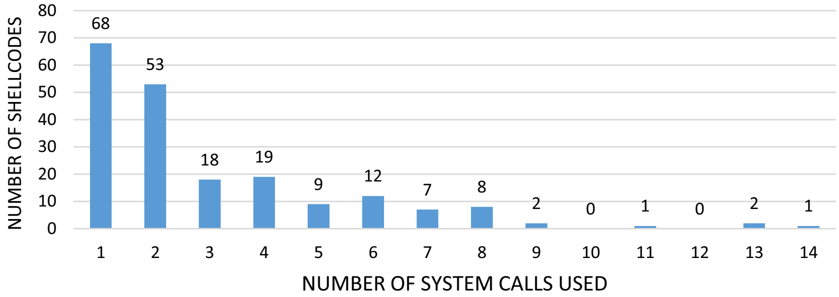
8 Related work

In this section, we review a number of works that are closely related to our work. In 2010, Bletsch, et.al introduced the jump oriented programming (jop) method [1]. They proposed a so-called dispatcher gadget as well a dispatch table and claimed that in JOP control flow can be driven without relying on the stack. But by analyzing their proposed sample attack in [1]'s technical paper, we observed that JOP uses the stack at least for loading the appropriate pointers to the dispatch table. They also mentioned (probably for the first time) that the call-ending gadgets can be used in a code-reuse attack. However, since in their method no mechanisms is incorporated to naturalize the side effects of call-ending gadgets (e.g. removing the undesired return values from the stack), the call-ending gadgets can only be placed at the end of the gadget's chain, so they do not have any negative effects on the control flow. In fact, having intermediate call-ending gadgets if not impossible is quite hard. Their sample JOP attack depicted in Fig. 5 of [1]'s technical report shows this fact too.

In 2014, Goktas et.al, proposed to use call gadgets to bypass the CCFIR [20] detection method and call functions in Windows operating system [10]. They introduced entry point (EP) gadgets that are blocks of instructions starting at a function's entry point and ending with an indirect call or jump. By chaining such gadgets, they can invoke arbitrary function in Windows. Using the Goktas's proposed EP gadgets in Linux operating system to build a chain of call-ending gadget is mostly infeasible due to some reasons. First, the large number of instruction in the EP gadgets rises the probability of disrupting the control flow structure of the attack

[1] One question that might come into mind is that why one of the 1-syscall shellcodes cannot be converted into its equivalent PCOP shellcode. The reason is that this shellcode code used in Forkbomb attack, uses the sys_fork syscall (No. 0x02) and In the libc library (ver.2.19), there is no kernel-trapper gadget with sys_fork syscall.

Fig. 10 Number of shellcodes with specific number of system-calls



(e.g. in the stack frame). Second, it has a similar limitation as of return-to-libc because it can only call functions not any desirable functionality. And finally, this mechanism cannot be used in Linux operating system because the EP gadgets are not available in the beginning of functions. By analyzing all the functions in libc-2.19 library, we found that all the gadgets that begin from the start of a function are ended with a direct call instead of indirect call or jump. And since the target address in such calls are direct (specified in compile time), performing a PCOP attack using such EP gadgets are impossible or at least quite hard.

In 2014, Wagner and Carlini, proposed an idea for using call-ending gadgets and called it call oriented programming [2]. Their idea is to use gadgets that end in double indirect calls (The target address is in a memory address which is pointed by the register specified in the call's operand). They also mentioned that a PCOP attack does not need a dispatcher gadget and it suffices that each of the functional gadgets points to the next functional gadget. Moreover, they explained that to launch a successful code-reuse attack, one cannot solely rely on call-ending gadgets and ret-ending gadgets are still needed. Note that limiting the PCOP gadgets to those with double indirect calls, lowers the number of available gadgets and thus lowers the possibility of designing and launching a successful PCOP attack. Moreover, the invocation of the call instruction pushes a return address to the stack, and disrupts the PCOP's control structure, but no solution has been proposed in this paper to address this problem. Because of these limitations, the authors conclude that launching a PCOP attack without using the classic ret-ended gadgets is quite hard and no proof-of-concept PCOP exploit has been proposed in the paper.

9 Conclusion

In this paper, we presented techniques for pure-call oriented programming (PCOP), a new class of code-reuse attacks that solely relies on call-ending gadgets. We discussed the side-effects of frequent use of the call instruction, and proposed new gadgets that allow us to neutralize these side effects and chain the gadgets. Using the proposed gadgets, we redesigned two attack models to take advantage of the proposed gadgets and perform PCOP attacks. We then presented TinyCOP, a form of a PCOP attack that uses minimal number of gadgets to avoid being detected by defensive mechanisms. We also introduced a real PCOP shellcode that spawns a Linux shell using only two call-ending gadgets. Finally, based on an analysis of the Shellstorm library and the fact that PCOP is Turing-complete, we showed that our proposed PCOP attack model is quite practical by demonstrating that many classical shellcodes can be converted into their equivalent PCOP shellcodes.

References

1. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 30–40. ACM
2. Carlini, N., Wagner, D.: Rop is still dangerous: breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 385–399
3. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 559–572. ACM
4. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: Drop: detecting return-oriented programming malicious code. In: International Conference on Information Systems Security, pp. 163–177. Springer
5. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., DENG, H.: Ropecker: a generic and practical approach for defending against ROP attack. In: Network and Distributed System Security Symposium (NDSS14)
6. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing, pp. 49–54. ACM
7. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 40–51. ACM
8. Designer, S.: Return-to-libc attack. Bugtraq, Aug (1997)
9. Falcn, F.: Exploiting cve-2015-0311, part ii: bypassing control flow guard on windows 8.1 update 3. https://www.coresecurity.com/blog/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3

10. Gktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 575–589. IEEE
11. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.: Scrap: architecture for signature-based protection from code reuse attacks. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013), pp. 258–269. doi:10.1109/HPCA.2013.6522324
12. Rose, J.R., Steele, G.L.: Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference (1999)
13. Salwan, J.: Shellcode Database. http://shell-storm.org/shellcode/
14. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561. ACM
15. Tang, J.: Exploring Control Flow Guard in Windows 10. http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-inwindows10.pdf
16. Ubuntu: Ubuntu Release End of Life. https://www.ubuntu.com/info/release-end-of-life
17. Wojtczuk, R.: The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, Volume 0x0b, Issue 0x3a (2001)
18. Yao, F., Chen, J., Venkataramani, G.: Jop-alarm: detecting jump-oriented programming-based anomalies in applications. In: 2013 IEEE 31st International Conference on Computer Design (ICCD), pp. 467–470. doi:10.1109/ICCD.2013.6657084
19. Yunhai, Z.: Bypass Control Flow Guard Comprehensively. Black Hat, BH US (2015)
20. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: IEEE Symposium on Security and Privacy (SP), pp. 559–573. IEEE