Clean the Scratch Registers:A Way to Mitigate Return-Oriented Programming Attacks

Zelin Rong, Peidai Xie, Jingyuan Wang, Shenglin Xu, Yongjun Wang

Abstract—With the implementation of W $\oplus$ X security model on computer system, Return-Oriented Programming(ROP) has become the primary exploitation technique for adversaries. Although many solutions that defend against ROP exploits have been proposed, they still suffer from various shortcomings. In this paper, we propose a new way to mitigate ROP attacks that are based on return instructions. We clean the scratch registers which are also the parameter registers based on the features of ROP malicious code and calling convention. A prototype is implemented on x64-based Linux platform based on Pin. Preliminary experimental results show that our method can efficiently mitigate conventional ROP attacks.

Keywords—binary instrumentation, calling convention, ROP attack, scratch register.

I. Introduction

SOFTWARE vulnerabilities are a serious threat to cyberspace security. Adversaries usually achieve their goals by hajacking the application's control flow with software vulnerabilities such as stack overflow [1], heap overflow [2], integer overflow [3], format string [4]. Once the adversary is able to hijack the control flow of the application, The adversary will execute malicious code and have the appropriate priviliges to perform malicious behaviors which may cause serious thearts to the system.

In the last few decades, operating systems and processor manufactures propose some solutions to mitigate these kinds of attacks such as W $\oplus$ X (Writable XOR Executable) security model [5] and ASLR (address space layout randomization) [6]. The W $\oplus$ X prevents adversaries from executing malicious code directly by making memory pages either writable or executable and ASLR randomizes the base address of program's memory so that the adversary can't get the addresses of instruction sequences directly. These memory protection mechanisms provide effective protections to a computer system.

Despite the employment of such methods, adversaries are still able to find ways to execute unintended code by making use of already existing code instead of injecting new malicious code which is called Return-Oriented Programming(ROP). ROP is a technique which chains together short existing instructions called gadget to perform malicious behavior, and it has been proved that ROP can perform arbitrary, Turing-complete computations [7].

Usually, the gadgets of ROP end with a return instruction which we called conventional ROP attacks. Call Oriented Programming (COP) [8] and Jump-Oriented Programming (JOP) [9] are the variations of ROP attacks without returns [10]. The variations use gadgets that end with indirect call or jump instruction. However, performing ROP attacks without return instruction in reality is difficult for the reason that the gadgets of COP and JOP that can form a completed gadget chain are almost nonexistent. Actually, adversaries prefer to use combinational gadgets to evade current protection mechanisms.

In the last few years, a number of defense mechanisms have been proposed to mitigate ROP attacks. These solutions can be categorized in compiler-based solutions [11]–[13]; dynamic approaches [14]–[19] and randomization-based solutions [20]–[22]. The compiler-based solutions try to eliminate gadgets of the program at compiling time itself. The dynamic approaches perform checking dynamically by monitoring the control-flow integrity of the program. The randomization-based solutions try to randomize the address or the instruction to mitigate ROP attacks. However, the existing solutions suffer from various shortcomings and practical deficiencies: the compiler-based solutions require side information such as debugging information and source code which are rarely provided in practice. The dynamic approaches usually detect ROP attacks based on the thresholds of return instructions which is not so reliable. What's more, they usually can't handle exceptional cases such as C++ exceptions, Unix signals which lead to suffering from false positives. And the randomization-based solutions can be bypassed or need side information.

Although there are some variations of ROP attacks, the conventional ROP attacks are still the main threat in reality [10]. In this paper, we focus on conventional ROP attacks. We find the features of ROP attacks: the gadgets try to write value to parameter registers and execute system calls, especially on x64 architecture. We also do a survey on calling convention and find that we can clean the scratch registers at return instruction that wouldn't affect the normal execution of program. Considering characteristics of them, we propose a scheme to mitigate ROP attacks based on return instruction. The scheme is implemented on x64-based Linux operating system based on binary instrumentation framework Pin which can dynamically clean the scratch registers storing parameters at return instruction [23], [24].

In summary, the main contributions of our work are:

- We analyze latest ROP malicious code, extract features of the ROP attacks and do a survey on calling convention.
- We verify that cleaning the scratch registers when the function returns does not affect the normal execution of the program.
- We propose a novel approach to protecting programs from

Zeling Rong is with National University of Defense Technology, Hunan, China (e-mail: zelin_rong@protonmail.com).

978-1-5386-7479-6/18/$31.00 ©2018 IEEE

convertional ROP attacks without access to source code.

- We design and implement a prototype on x64-based Linux platform and evaluate its security effectiveness and performance overhead.

The remainder of this paper is organized as follows. Section II provides background knowledge including an overview to ROP attacks, a survey of calling convention and festures of ROP attacks. The main idea of our approach and the architecture is provided at Section III. The details of our implementation is illustrated at section IV and Section V provides the evaluation result of the tool. We discuss our work in Section VI and conclude the paper in Section VII.

II. BACKGROUND

To better understand our prototype system, we provide the backgroud of our work in this section. First, we describe the ROP attacks and calling convention. Then we propose the features of ROP Attacks.

A. ROP Attack

Return-oriented Programming is a kind of code reuse attack technique, which executes gadget chains to perform tasks. Because it does't need to inject new malicious code, it is a effective way to defeat W $\oplus$ X protection mechanism.

Short instruction sequences that can be called as gadgets have two features: perform a particular atomic task such as load or store memory and then transfer the control flow to the next gadget with return instruction or indirect call or jump instruction. There are two sources to form a gadget. One is the instruction sequences come from the program itself, the other is unintended instruction sequences which begin in the middle of a valid instruction resulting in a new instruction sequences never intended by the programmer nor the compiler. Due to variable-length instructions and unaligned memory access, we can easily find such instructions on x86 and x64 architecture. For example, consider the following set of x64 instructions:

```
48 63 14 32 movsxd rdx, dword ptr [rdx+rsi]
29 c3       sub    ebx, eax
```

There is a "0xc3" byte in the instruction sequences which can be recongnized as a "ret" instruction. If a adversary hajacks the control flow and starts the execution from the middle of that sequences, then the sequences could be a gadget. The execution sequences will be:

```
32 29         xor    ch, [rcx]
c3            ret
```

Adversaries chain gadgets together and transfer the control flow from one gadget to another by writing appropriate values to memory, normally to stack. Once the adversary hajacks the control flow to the first gadget, arbitrary behavior can be done. It has been proved that ROP attack technique is Turing-complete [7].

Figure 1 provides a overview of conventional ROP attack. In step 1, the adversary hajacks the control flow and moves the stack pointer to the first gadget address with a memory-related vulnerability such as buffer overflow vulnerability. The first gadget address overwrites the original return address. After that the adversary will hajack the control flow to the first gadget upon the original function returns, as shown in step 2. The first gadget performs a small computation and is terminated by another return instruction which pops the second gadget address from the memory (step 3) and redirects execution to the next gadget (step 4). This procedure is repeated until the adversary achieves goal and terminates the program.

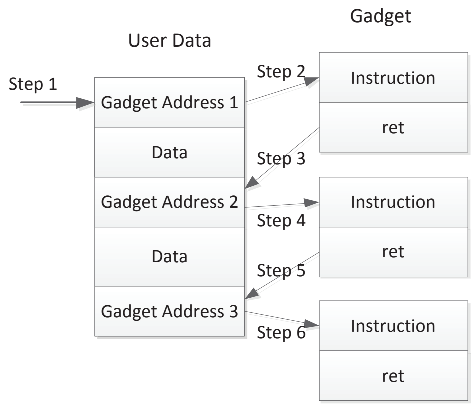


Fig. 1: Overview of conventional ROP attack

Compared to the ROP attacks with return instruction, some new variants of ROP attack without return instructions are proposed [8]–[10]. The variants usually use indirect jump or call instruction to chain together gadgets instead of return instruction. However, performing ROP attacks without return instruction in reality is difficult for the reason that the gadgets of COP and JOP that can form a completed gadget chain are almost nonexistent.

To evade current protection mechanisms, adversaries usually prefer to use combinational gadgets, which means using the combination of gadgets based return instruction and indirect jump or call instruction together. In this paper, we focus on the ROP attacks based on return instruction.

B. Calling Convention

In computer science, a calling convention is a convention between caller and callee. It is an implementation-level scheme for how subfunctions receive parameters from their caller and how they return a result, specifically, including where parameters, return values, return addresses and scope links are placed.

In some conventions, the caller is responsible for cleaning the arguments such as _cdecl. While in other conventions, cleaning up the arguments is the responsibility of the callee, such as stdcall and Microsoft fastcall. No matter whether caller or callee cleans up the arguments, what we really focus on is which registers are needed to retain their values after a subroutine call (preserved register) and which registers are considered to be volatile and, if volatile, need not be restored by the callee (scratch register). Which means, we need to figure out which registers can be directly used by the callee without being preserved and which registers should be preserved first and must be restored at return by the callee.

On x86 architecture, the registers EBX, ESI, EDI, EBP, DS, ES, and SS are preserved registers. If uses them, callee must save them first and restore them afterwards. EAX and EDX registers are used for return values, and thus should not be preserved. The other registers do not need to be saved by the callee function, which means if they are used by the callee function, then the caller function should save them before the call is made, and restore after the call is over.

X86-64 calling conventions take advantage of the additional register space to pass more arguments in registers instead of stack on x86 architecture. There are two calling conventions in common use: Microsoft x64 calling convention [25] and System V AMD64 ABI [26].

The Microsoft x64 calling convention is normally followed on Windows and pre-boot UEFI (for long mode on x86-64). The registers RCX, RDX, R8, R9 are the first four integer or pointer argument registers, while the first four floating point argument registers are XMM0, XMM1, XMM2, XMM3. If there are more arguments, the stack would be used to pass arguments. The RAX register is used for storing integer return value and XMM0 is used for floating point return value. The registers RAX, RCX, RDX, R8, R9, R10, R11 are srcatch registers. The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are preserved registers.

The calling convention of the System V AMD64 ABI is normally followed on Unix and Unix-like operating systems such as Solaris and Linux. The registers RDI, RSI, RDX, RCX, R8, R9 are the first six integer or pointer argument registers and the floating point argument registers are XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7. Similar to the Microsoft x64 calling convention, additional arguments are passed on the stack. The registers RAX and RDX are used for storing integer return value and XMM0 and XMM1 are used for floating point return value. The preserved registers are RBX, RBP, and R12-R15 and all other registers are scratch registers.

The table I [27]–[29] shows a quick overview of common calling conventions.

C. Syscall

It's worth noting that on Linux system, system functions such as "read" and "write" are normally the wrapper functions of syscall. Syscalls [30] are the interface between user programs and the Linux kernel. User programs use it to make the kernel perform expected system tasks, such as file access and networking.

On Linux system, "int 0x80" instruction is the interrupt of x86 architecture to enter the kernel and "syscall" instruction is the interrupt of x64 architecture to enter the kernel. Each syscall has a fixed number (note: the numbers differ between int 0x80 and syscall). For eaxmple, number 1 is the exit syscall number of "int 0x80" instruction and number 1 is the write syscall number of "syscall" instruction. The syscall number is passed by the EAX/RAX register. Most of syscalls take parameters to perform their task. Those parameters are passed by writing them in the appropriate registers before making the actual call. Each parameter index has a specific register. The table II provides the overview of the system call.

TABLE I: Overview of common calling conventions

platform	return value	parameter registers	additional parameters	scratch registers	preserved registers
System V i386	EAX, EDX	None	Stack (right to left)	EAX, ECX, EDX	EBX, ESI, EDI, EBP, ESP
System V x86_64	RAX, RDX	RDI, RSI, RDX, RCX, R8, R9	Stack (right to left)	RAX, RDI, RSI, RDX, RCX, R8, R9, R10, R11	RBX, RSP, RBP, R12, R13, R14, R15
Microsoft x64	RAX, RDX	RCX, RDX, R8, R9	Stack (right to left)	RAX, RCX, RDX, R8, R9, R10, R11	RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15

TABLE II: Overview of syscall

instruction	syscall number registers	parameter registers(1 to 6)	retrun value registers
int 0x80	EAX	EBX, ECX, EDX, ESI, EDI, EBP	EAX
syscall	RAX	RDI, RSI, RDX, R10, R8, R9	RAX

After all the parameters are set up correctly, user programes call the interrupt with "int 0x80" or "syscall" and the kernel would perform the corresponding task. The return/error value of a syscall is written to EAX/RAX.

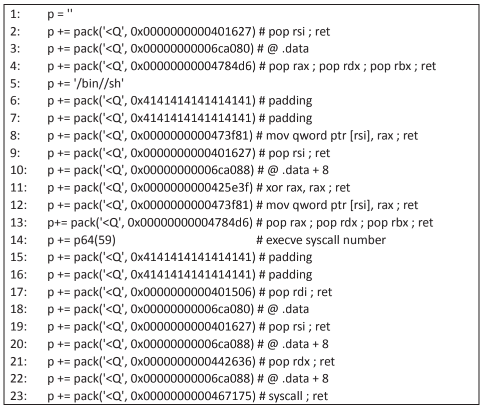
D. Features of ROP Attack

Because of the exsistence of W $\oplus$ X security model, An adversary can't execute shellcode directly. In essence, The purpose of an adversary uses ROP is to disable W $\oplus$ X with syscall or directly execute shellcode in the way of gadget chains.

A typical ROP gadget chains generated by ROPgadget [31] is shown in the figure 2. The function of the example is obtaining a command shell from which the adversary can control the compromised machine, actually, it executes "execve("/bin/sh", ["/bin/sh"], NULL)" in the program to get a shell. As we can see, the first parameter is passed to the RDI register in line 17 and the second is passed to the RSI in line 19 and the syscall number is 59 in RAX register in line 13.

Based on the practical experience of reading and writing ROP code, we find the features of ROP attacks as follows.

First, the destination of using gadget chains in usual is performing system call or system fucntion to perform malicious behaviour such as file access, network access and W $\oplus$ X disable. In most cases, the adversary would like to disable W $\oplus$ X. Because once W $\oplus$ X has been disabled, shellcode can be executed directly instead of rewritting shellcode to ROP chains which may cause some troubles for the adversary. In



```
1:      p = ''
2:      p += pack('<Q', 0x0000000000401627) # pop rsi ; ret
3:      p += pack('<Q', 0x00000000006ca080) # @ .data
4:      p += pack('<Q', 0x00000000004784d6) # pop rax ; pop rdx ; pop rbx ; ret
5:      p += '/bin//sh'
6:      p += pack('<Q', 0x4141414141414141) # padding
7:      p += pack('<Q', 0x4141414141414141) # padding
8:      p += pack('<Q', 0x0000000000473f81) # mov qword ptr [rsi], rax ; ret
9:      p += pack('<Q', 0x0000000000401627) # pop rsi ; ret
10:     p += pack('<Q', 0x00000000006ca088) # @ .data + 8
11:     p += pack('<Q', 0x0000000000425e3f) # xor rax, rax ; ret
12:     p += pack('<Q', 0x0000000000473f81) # mov qword ptr [rsi], rax ; ret
13:     p+= pack('<Q', 0x00000000004784d6) # pop rax ; pop rdx ; pop rbx ; ret
14:     p += p64(59)                        # execve syscall number
15:     p += pack('<Q', 0x4141414141414141) # padding
16:     p += pack('<Q', 0x4141414141414141) # padding
17:     p += pack('<Q', 0x0000000000401506) # pop rdi ; ret
18:     p += pack('<Q', 0x00000000006ca080) # @ .data
19:     p += pack('<Q', 0x0000000000401627) # pop rsi ; ret
20:     p += pack('<Q', 0x00000000006ca088) # @ .data + 8
21:     p += pack('<Q', 0x0000000000442636) # pop rdx ; ret
22:     p += pack('<Q', 0x00000000006ca088) # @ .data + 8
23:     p += pack('<Q', 0x0000000000467175) # syscall ; ret
```

Fig. 2: Typical example of ROP

upper example, the system call is number 59 which is "execve" system call.

Second, if the adversary performs ROP attacks using system call instruction, no matter on x86 or x64 architecture, the register would be used to pass parameter. Or if the adversary performs ROP attacks using system function such as "read" or "mprotect", on x64 system, the register would still be used to pass parameters, as mentioned in subsection B and C.

III. System Design

Based on the features of ROP attacks and calling convention, we propose a new way to mitigate conventional ROP attacks which is cleaning the scratch registers that are also the parameter registers when meeting return instruction.

A. Assumptions

We make the following assumptions in order to simulate the real environment:

1. We assume that operating systems and processor enable the W $\oplus$ X security model. In that way, adversaries can't execute malicious code directly.
2. We assume that the adversary performs ROP attacks based on the return instruction.
3. We assume that we have no access to side information such as source code and debugging information while defeating ROP, which is also rarely provided in practice.

B. Main Idea

The main idea of our method is that when meeting return instruction, clean the scratch registers that are also the parameter registers.

The reason why it can mitigate ROP attacks and have no affect on normal performance of programe shows as below. During the process of ROP attacks, cleaning the parameter registers can undermine the adversary's intention to use gadgets. When the gadget of system call have been called, the parameter registers have already been destroyed then the ROP attack will lapse. We do not need to worry about that cleaning the registers will affect the normal performance of the program because that the regsisters are also the scratch registers. The callee fuctions don't need to preserve the value of scratch regsisters, so cleaning the value at return would not affect the execution of the program at return instruction.

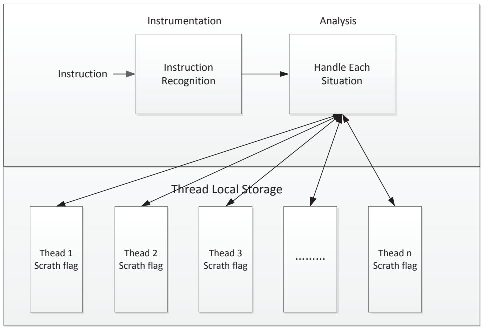


Fig. 3: Architecture of the system

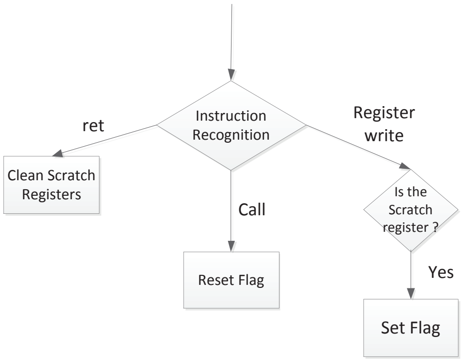


Fig. 4: Flow chart of the system

As shown in the figure 2, once we cleane the register RDI at the return in line 17 and the register RSI at the return in line 19, the system call "execve" will fail for it lose the first two parameters.

C. Work Flow

Base on the main idea, we make use of instrumentation to implement our system. In that way we can add extra code to monitor the behavior of the program without the need of source code.

The architecture of our system is shown in figure 3. We set a flag for each scratch register to indicate whether the register is used or not by the callee. To avoid that one thread accesses the flag of another thread, we set flag for each thread. Every instruction is intercepted before it is executed by the processor. In instrumentation component, we recongnize every instruction's type. And in analysis component, we do the corresponding operation for each instruction.

Figure 4 shows the flow chart of the system. Before a instruction is executed by the processor, we first check whether the instruction is a call instruction or not. If it is a call, we reset the flag of all the scratch registers. If it is not a call, then we check if the instruction is a return instruction, if it is true, we clean the value of scratch registers of which flag has been set. Otherwise, if the instruction writes value to scratch registers which means the registers don't need to be preserved at return, we set the flag of the registers.

IV. System Implementation

Section III provides the main idea and overall design of the system to prevent ROP attacks with cleaning the scratch registers. In this section, we discuss specific implementation details of our prototype.

We have developed the prototype implementation for the 64-bit version of Ubuntu 16.04. Our system is implemented on the dynamical binary instrumentation tool Pin (version 3.5) [23], [24].

We choose RDI, RSI, RCX as the registers we clean. These three regsiters are the scratch registers on system V x86_64, so cleaning these three registers will not affect the program behavior at return if their values have been changed in the callee. These three registers are the scratch registers which are also the parameter registers on system V x86_64 architecture. RDI register is the first parameter register and RSI is the second and RCX is the forth. Cleaning their values will cause the system call losing the proper parameters which then mitigate the ROP attacks. It is not necessary to clean all the scratch registers for the reason that it may cost too much resources and cause high performance overhead and some of them are not parameter registers.

A. Instrumentation Framework

We focus on dynamic binary instrumentation at runtime to avoid access to side information. After doing a research on binary instrumentation framework such as DynInst [32], Vulcan [33], DTrace [34], Valgrind [35] and Pin [23], [24], we decide to use Pin as our instrumentation framework.

Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures, which requires no source code and can support instrumenting programs dynamically. It allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. There are many tools built based on Pin, such as VTune Amplifier XE, Inspector XE, Advisor XE and SDE.

There are two kinds of working mode in Pin, probe mode and just-in-time (JIT) mode. In JIT mode, Pin allows users to inject instrumentation before the instruction is executed by the processor. Probe mode is a method which insert probes at the start of specified routines. A probe is a jump instruction which is placed at the start of the target routine and redirects the control flow to the instrumentation code. As we need to instrument unintened instruction, we use JIT mode to implement our system.

Conceptually, instrumentation consists of two components: instrumentation and analysis code. Instrumentation code decides where and what code is inserted and analysis code decides what code to execute at insertion points. Pin provides these two components with Pintool. Pintools can be thought of as plugins of Pin which can modify the code generation process. Normally, there are two types of routines in Pintool: instrumentation callback routine and analysis routine. Each one represents the corresponding component. Pin also provides a rich API allowing users to develop their own Pintool.

We designed and implemented our own Pintool to mitigate ROP attacks with the Pin framework.

B. Instrumentation and Analysis Routines

We need to figure out two things in the system: where to insert code and what code to execute at insertion points.

As shown in section III, before a instruction is executed by the processor, we need to recongnize the instruction's type in instrumentation routines to check whether the instrution is call or return or the operation of writting a value to scratch registers. We use the inspection routines INS_IsCall(INS ins) and INS_IsRet(INS ins) provided by the Pin API to recognize the instruction is a call or return instruction. We use INS_RegWContain(INS ins, REG reg) to check whether the instrution change the value of scratch registers or not.

In analysis routines, when the instruction is a call, we reset all the flag of scratch registers. To avoid that one thread accesses the flag of another thread, we use the thread local storage (TLS) from the Pin API, so that each thread has its own flag. When the instruction is a return, we clean the value of the registers of which flag has been set. And when the instruction is a operation of writting value to the scratch registers, we set the corresponding flag of the register.

V. Evaluation

In this section, we test the security effectiveness of our system. We also test the false negative of it to check whether it will affect the normal execution of program. Finally we evaluate the performance overhead of the system. The evaluation is performed on an Intel(R) Core(TM) i7-4790 3.60GHz machine with 8GB memory and 64-bit Ubuntu OS which kernel is 4.4.0.

A. Security Effectiveness

We first test whether our system can really mitigate conventional ROP attacks.

First, We collected some programs that appeared in the CTF contest [36] and are categorized as "pwn" which means they can be exploited to get a shell. We used the programs that can be exploited by ROP attacks. We classfied the programs into five categories according to their types of vulnerability and ways of compiling. We ran the programs under the instrumentation of the system and then ran the exploit scripts to attack them to see whether we can get the shell. The result is shown in table III.

To further test the security effectiveness of our system, we viewed the shell-storm [37] website and rewrote some shellcodes in the way of ROP, and tested them with our system.

We used a simple target program that had a strcpy vulnerability. We compiled the program with gcc-5.4.0 in static way.

There are 35 shellcodes for x64 architecture on shell-storm websit. After reading all the shellcodes, we picked up 7 typical shellcodes and rewrote them in ROP with the gadgets which generated by ROPGadget [31]. Then we attack the vulnerable program with the shellcodes. The result shows as table IV.

As demonstrated, the experimental results show that our system could mitigate all these ROP attacks without false positive. So we believe our technique can be used to mitigate conventional ROP attacks.

TABLE III: The test result of ROP attacks on CTF contest

vulnerability classification	size(kb)	without instrumentation	under instrumentation
stack overflow (dynamic compiled)	8.9	get shell	failed to get shell
stack overflow (static compiled)	806.1	get shell	failed to get shell
format vulnerability (dynamic compiled)	20	get shell	failed to get shell
format vulnerability (static compiled)	844.8	get shell	failed to get shell
heap vulnerability (dynamic compiled)	13.8	get shell	failed to get shell

TABLE IV: The test result of ROP attacks on shellcode

shellcode	instruction count	gadget count	without instrumentation	under instrumentation
add map in /etc/hosts	28	35	succeed	failed
read /etc/passwd	24	30	succeed	failed
bind-shell with netcat	35	41	succeed	failed
shutdown -h now	22	28	succeed	failed
reverse tcp bindshell	50	30	succeed	failed
tcp bindshell	70	41	succeed	failed
setuid(0) + execve(”/bin/sh”)	19	20	succeed	failed

B. False Negative

To see the false nagative of our system, we tested the system over the set of GNU Coreutils, and a set of other real-world applications and server programs including firefox and httpd. We also coded program with all kinds of funtions from glibc and checked the return values of the funtions.

We tested out system over the set of GNU Coreutils including almost 113 programs such as “head” and “sha1sum”. All the program behaviored well under the instrumentation of our system. We also monitored 8 real and big programs: chrome, firefox, gdb, httpd, libreoffice, python, supertux2 game. None of them crashed under the instrumentation. They all performed well.

We also checked the return values of functions from glibc. We classfied functions into 14 categories according to their features [38] which include almost 250 functions. As we can see from table V, the situations under the instrumentation are the same as without the instrumentation. It is noteworthy that our system can handle the situations like setjmp, signal, exception and lazy binding that some dynamic detecting solutions can’t handle.

TABLE V: The test result of ROP attacks on functions

functions	result of under instrumentation
signal relative	work fine
setjmp relative	work fine
exception relative	work fine
socket relative	work fine
IO std relative	work fine
IO file relative	work fine
string relative	work fine
memory relative	work fine
math relative	work fine
time relative	work fine
user and group privilege relative	work fine
process relative	work fine
file permission relative	work fine
C++ and Class	work fine

TABLE VI: Performance overhead

program	size(kb)	benchmark	without instrumentation(s)	under instrumentation(s)	overhead
bzip2-1.0.6	31.3	uncompress a 469 KB file	0.038	0.538	14.1X
gzip-1.6	98.2	compress a 76.2 MB file	3.037	14.925	4.9X
grep-2.25	211.2	find pattern in 76.2 MB file	0.045	0.633	22.5X
gcc-5.4.1	915.7	compile a 1.3KB source code	0.032	0.795	24.8X
firefox-58.0	199.9	browse website	4.072	87.221	21.4X
chrome-64.0	123063	browse website file	3.312	66.513	20.0X
tar-1.28	383.6	compress a 76.2 MB file	0.046	0.597	13.0X
gocr-0.49	421.8	process JPG file	0.121	3.031	25.0X
cat-8.25	52.1	cat a 487.7KB file	0.016	0.307	19.1X
hexdump	31.2	hexdump a 487.7KB file	0.179	1.546	8.7X
Average			1.090	17.611	16.2X

C. Performance Evaluation

To evaluate the performance of our system, we measured the time consumption that running normal applications needed. The evaluation was tested on programs. For each program, we tested the performance overhead when the program runs without instrumentation and under the instrumentation of the system, as shown in table VI.

The average performance overhead is nearly 16.2 times. The performance overhead of our system is relatively high which is a shortcoming. The recognition of instruction’s type and routines relative to TLS cost the main performance overhead, especially the recongnition of writting value to scratch regsisters and setting the flag.

VI. Discussion

Our system can mitigate conventional ROP attacks based on the theory of calling convention. It doesn't need any side information like source code which most of compiled ways need. Compared with existing dynamic detect methods, it can mitigate ROP attacks with no dependency on thresholds and no false negative.

Although it is effective at mitigating ROP attacks, there are some limitations. First, the system can only mitigate ROP attacks based on return instruction, which means that it can only mitigate the conventional ROP attacks and some mixed attacks. It has no effect on attacks based on the instruction without returns such as COP and JOP. Second, the system is implemented by using the binary instrumentation framework Pin and causes an average slowdown of 16.2 times. The performance overhead is relatively high.

VII. Conclusion

In this paper, we analysed calling convention and Return-Oriented Programming which is a powerful attack that bypasses current security mechanisms. We also extracted the features of ROP attack : adversaries usually try to call fucntions like disabling W $\oplus$ X or file access with gadgets.

Based on the observation, we proposed a solution to mitigate conventional ROP attacks: clean the scratch registers which are also the parameter registers. Then we designed and implemented a prototype system. After evaluation, it has been proved that our method can mitigate ROP attacks based on return instruction with no false negative.

Our system induces a performance overhead by a factor of 16.2x which is relative high and it also can't handle ROP attacks without returns which are the next question we focus on to solve.

Acknowledgment

We thank the anonymous reviewers for their constructive comments that guided the final version of this paper. We thank National University of Defense Technology for providing essential conditions to accomplish this paper. This work is supported by NSFC No.61472439, National Natural Science Foundation of China under Grant.

References

[1] A. One, "Smashing the stack for fun and profit (1996)," *See http://www. phrack. org/show. php*, 2007.

[2] Anonymous, "Once upon a free()..." *See http://phrack.org/issues/57/9.html*, 2001.

[3] Blexim, "Basic integer overflows," *See http://phrack.org/issues/60/10.html*, 2002.

[4] R. Gera, "Advances in format string exploitation," *See http://phrack.org/issues/59/7.html*, 2002.

[5] Microsoft, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in, 2017.

[6] P. Team, "Pax address space layout randomization (aslr)," 2003.

[7] H. Shacham, "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*. ACM, 2007, pp. 552–561.

[8] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[11] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with return-less kernels," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 195–208.

[12] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[13] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 353–362.

[14] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *International Conference on Information Systems Security*. Springer, 2009, pp. 163–177.

[15] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[16] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security Symposium*, 2013, pp. 447–462.

[17] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in *International Conference on Information Systems Security*. Springer, 2010, pp. 140–155.

[18] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. Deng *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[19] I. Fratric, "Runtime prevention of return-oriented programming attacks, 2012," *URL https://ropguard. googlecode. com/svn/trunk/doc/ropguard. pdf*.

[20] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[21] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 571–585.

[22] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in *International Conference on Network and System Security*. Springer, 2013, pp. 293–306.

[23] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. ACM, 2004, p. 22.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[25] Microsoft, "x64 architecture," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture, 2017.

[26] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface," *AMD64 Architecture Processor Supplement, Draft v0*, vol. 99, 2013.

[27] D. Lawlor, "Function calling, and preserved registers," https://www.cs.uaf.edu/2012/fall/cs301/lecture/09_10_functions.htmlhttps://www.cs.uaf.edu/2012/fall/cs301/lecture/09_10_functions.html".

[28] Microsoft, "Register usage," https://msdn.microsoft.com/en-us/library/9z1stfyw.aspx, 2015.

[29] ——, "Caller/callee saved registers," https://msdn.microsoft.com/en-us/library/6t169e9c.aspx, 2015.

[30] F. S. Foundation, "System calls," https://www.gnu.org/software/libc/manual/html_node/System-Calls.html.

[31] J. Salwan, "Ropgadget - gadgets finder and auto-roper," http://shell-storm.org/project/ROPgadget/, 2011.

[32] B. Buck and J. K. Hollingsworth, “An api for runtime code patching,” *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[33] A. Edwards, H. Vo, and A. Srivastava, “Vulcan binary transformation in a distributed environment,” 2001.

[34] B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, “Dynamic instrumentation of production systems.” in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

[35] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[36] C. team, “Ctf? wtf?” https://ctftime.org/ctf-wtf/.

[37] JonathanSalwan, “Shellcodes database for study cases,” http://shell-storm.org/shellcode/.

[38] PennyHot, “Linux c function(),” http://net.pku.edu.cn/~yhf/linux_c/.