Hacking in Darkness: Return-Oriented Programming Attacks, Mitigation Strategies and its Benign Uses

1st Ms. Sonali (Mentor)
Assistant Professor
School of Computing
Indian Institute of Information Technology Una
Himachal Pradesh, India
sonalimahajan@iiitu.ac.in

2nd Monty Shyama
UG Student
School of Computing
Indian Institute of Information Technology Una
Himachal Pradesh, India
16111@iiitu.ac.in

***Abstract*—Return-oriented programming (ROP) is one of the most common attack methodologies to exploit software specific vulnerabilities in the modern-day operating systems. A threat actor can execute unintended code with the help of ROP chains despite the existence of various security mechanisms. The property that the ROP is resistant to static analysis has drawn researchers' attention to come out with solutions mitigating the ROP attacks. The same property is used to hide some important code and also results in a low overhead on program size. The paper discusses the ROP chaining methodologies. The various defense mechanisms based on randomization, frequency and control flow integrity are also discussed. At the end of the paper, the future of existent mitigation strategies and their effectiveness are also highlighted.**

***Index Terms*—Return-oriented programming, ROP mitigation strategies, Return-to-libc attack, Binary exploitation**

I. Introduction

The most prevalent forms of vulnerabilities in the modern-day operating system is memory leakage and data corruption vulnerabilities. Earlier, the attacker used to run shellcodes on the stack directly in order to exploit the vulnerable binaries. But, as the Data Execution Prevention (DEP) mechanism was evolved, the arbitrary shellcode execution method becomes no more of use. DEP uses a W XOR X security model in which the memory location can either be writeable or executable, but not both at the same time [1]. Despite the deployment of such a method, the attacker is still able to exploit a software vulnerability. Return-to-libc attack was the first method to circumvent the W XOR X protection, it utilises functions already present in the libc. In order to execute arbitrary code inside the binary, return-oriented programming (ROP) was proposed to attain the target. It is a well-known fact that each software is given an address space during its execution. ROP utilises instructions already present in the address space of binary and chains them into a meaningful attack vector. To chain these code fragments together, the adversary links them with an intermediate control transfer instruction, ret. Since ROP is Turing complete, so an attacker can use it to achieve any function. Till now, ROP is one of the most common attacks methodologies to exploit a software security vulnerability.

Since the effectiveness of a ROP attack is dependent on the address exposure of instructions, some general methods were suggested as a part of the remedy for preventing these attacks [3]. One such method is the Address space layout randomization (ASLR). In ASLR, the base address of the stack, heap, and external libraries are random and different for each execution of the program, so the attacker is not able to gain the exact addresses for either the libc functions nor the code fragments during the program execution. In addition to the randomization approaches, the researchers also found defense mechanisms based on instruction frequency and control flow integrity. These approaches mitigate ROP attacks to a much greater extent, but due to the overhead on the program, some of these mechanisms have not been applied in practice.

In the last decade, the situation of software security has become more severe and in-depth research has been carried out on ROP. These research outlines the attack methodologies, the corresponding mitigation strategies, and security impact. The rest of this paper is organised as follows. ROP are discussed in section II. In section III, the proposed mitigation strategies are outlined. In section IV, benign uses of return-oriented programming are discussed. In sections V and VI, the report is concluded.

II. ROP ATTACK TECHNIQUES

Return-oriented programming (ROP) was introduced by Shacham, a technique which provides an attacker with the capability to change the control flow of the vulnerable program and utilise it to gain unauthorized access to the system. ROP was originally experimented on Intel x86 architecture, but later on, it was extended on numerous other architectures including ARM and the Intel x86-64. It can thus be safely assumed that ROP chaining is an architecture-independent attack.

A. Return-oriented programming

In a classical buffer overflow attack, the vulnerability is generally exploited by overwriting the return address in a call stack. ROP is different in a way that it utilises registers for storing and manipulating values. The reason is that the stack cannot be overwritten if the Data Execution Prevention (DEP) security mechanism is kept in place. Hence, custom shellcodes can't be injected even if the software is vulnerable.

To circumvent the W XOR X security model, return-to-libc attack was introduced where the attacker overwrites the return address with the address of libc functions such as system() and execve(), which can be further used to spawn the shell and gain unauthorised access to the system containing the vulnerable binary. Return-to-libc is still helpful to bypass the W XOR X, but it has some limitations that it cannot be used to execute arbitrary code by calling predefined functions in the libc. To overcome this problem, Shacham proposed the use of already existing code fragments ending with ret; inside the running program and replaced the original shellcode. The adversary first calculates the address of these special code fragments using various search mechanisms and then combines these code fragments, ending with ret instruction to generate a chain of assembly code. These chains are called gadget by Shacham and can be found by using the Galileo algorithm. These gadgets are divided into two categories: intended gadgets and unintended gadgets. The intended gadgets are derived from function epilogue and the unintended gadgets come from unaligned instructions.

The first step in the attack is to identify the number of junk bytes required to overflow the stack and reach the return instruction. Once the padding is identified, the address for suitable ROP gadgets is extracted from the binary. Then, ROP chains are formed by writing a suitable algorithm. In Intel 64-bit architecture, instead of storing arguments for a system calls on the stack, they are put inside the registers. The first argument is always stored in the %rdi register. So, the attack strategy is to simply overwrite the %rdi register with the string "$/bin/sh$" and then call the system() which will eventually spawn a shell.

To achieve a successful ROP attack, the adversary needs three things: an overflow vulnerability, suitable code fragments, and address for ROP gadgets. Shacham also proved that ROP attack is Turing complete, which theoretically proves that ROP can achieve any attack. In order to prove that, Shacham proved that the gadgets in ROP attack can implement the following functions: Memory load/store, Arithmetic and logic, Control flow transfer, System calls and function calls.

B. *ROP Automation*

It is difficult to find all the suitable ROP gadgets in a binary manually; automation is used to make the attack easier and convenient. There are two major steps involved in ROP automation: 1) Finding the required gadget, which is known as gadget searching; 2) Chaining the gadgets together to achieve the attack, known as gadget compiling. Some of the popular tools used for gadget searching are ropper, ROPgadget, ROPme, etc. All these tools use galileo algorithm proposed by Shacham in 2007 for searching suitable ROP gadgets. Gadget compiling represent gadget in an Intermediate Language and then using instruction matching, determine gadgets for final execution.

C. *Extended ROP attack*

Shacham also proposed that ROP attack can be achieved without using ret instruction. The ret instruction poses some of the properties which make it suitable for use in ROP attack: 1) transfer control of instruction; 2) load an immediate value into the register; so a subsequent return will not cause the program to crash. Thus, jmp and call instruction can also be used for Return-oriented programming.

Stephen Checkoway also pointed out the possibility of achieving ROP chaining without using ret instructions. An update-load-branch sequence can be used in place of ret which will update the global state of the registers acting as a pointer to the return-oriented program's instruction pointer. Once the global state is updated, the instruction pointer will fetch the corresponding instruction from the right memory location; and it will then branch to that instruction. On the Intel x86 architecture, the author recommends using "pop x; jmp *x", where x can any general-purpose register. The method showed by Stephen showed the chances of using jmp and call to replace ret.

In call-oriented programming, the attacker uses special gadgets ending with jmp or call instruction instead of ret. The call instruction will push the next instruction address on the stack so that it remembers where to return after completing the function call. Although call ending gadgets are feasible for use in Return-oriented programming attacks, the unnecessary address on the top of the stack makes it difficult to change the control flow of the program as per attacker's need. The call-oriented programming was only in theories until AliAkbar Sadeghi came out with the idea of Pure-Call Oriented Programming(PCOP) in 2017 [2]. He used a strong trampoline gadget, like "pop eax; popad; cld; call eax;" to remove undesired return addresses pushed onto the stack by previous gadget's call instructions.

III. ROP Mitigation Strategies

Return-oriented programming has a high success rate in bypassing the W XOR X guard, and ROP attack has become a necessary step in exploiting a software vulnerability in modern-day operating systems. To reduce the damage due to ROP attacks, there are some prevention techniques proposed for ROP mitigation. In this section, mitigation strategies based on the frequency of gadget, control-flow integrity, and randomization are discussed.

A. *Frequency of gadgets*

In return-oriented programming, the gadget is the basic unit. A gadget is extracted from the existing code and consist of two or three instructions ending with a ret. During the attack, the program stream will have a number of instructions ending with ret. Such a high frequency of return instructions can become a method to detect ROP chaining on binary under attack. A defense system will hook the ret instruction and will count the length of this code fragment. It will also count the length of continuous code snippets ending with ret. The effectiveness of this defense mechanism depends on the length of the gadget used, and the number of consecutive times the gadget is being called inside the binary. By manipulating the two variables, the sensitivity of detection can be increased or decreased. However, high sensitivity means high false alarms while low sensitivity means ROP cannot be detected. Ping Chen et. al. proposed DROP, a ROP attack defense method based on this principle. Frequency approach is suitable at the binary level, but due to security reasons, it has not been implemented yet on modern-day operating systems.

The side effect of frequency-based approach makes it difficult to be used in practice, but the frequency approach combined with control flow integrity method achieves a better performance in gadget detection.

B. Randomization

One of the basic necessities in achieving a successful ROP attack is to know the exact addresses of the ROP gadget instructions. Randomization of addresses for each execution of program can be used to resist ROP attack. It mainly consists of a technique called Address Space Layout Randomization (ASLR). ASLR randomizes base addresses of the stack, heap, libc functions and others so that the adversary cannot predict the exact addresses of where these parts could be loaded during the program execution, and thus cannot get the address of gadgets to corrupt the stack frame. ASLR reduces risk of ROP attacks but cannot do away with it completely. In reality, not all parts of the binary are randomized. Only the base addresses of libc and other entities are randomized. Internally, the position of every function inside libc is the same for each round of execution for the program. If there is a memory leakage from any parts of the program during its execution, the attacker can calculate the offset required to reach to the correct address for libc functions. Hiser et al. came out with a method that inspected the vulnerable software offline to retrieve the actual locations of each instruction and the target address of indirect instructions, following that the method outputs a re-write rule for these indirect instructions. Compared with other defense mechanisms, the randomization technique has a minimal modification to the binary.

Among all the defense mechanisms, ASLR is the only method adopted by modern-day operating systems. ASLR is prominent because of its transparency to the software developers.

C. Control-flow Integrity

In return-oriented programming, the execution of code fragments is either picked from incomplete calls of functions or from unaligned instructions. Control-flow integrity is useful in detecting ROP attacks. Control-flow integrity is divided into two categories: compiler-based approach and dynamic approach. The compiler-based approach modifies the layout of code during program execution and injects checker codes additionally that analyzes the behaviour of free branch instructions [3]. If the free branch instruction is not at its specified address, then the defense system will launch an alarm or a program exception to halt the program execution. Some compiler-based approach rewrites the binary and reduces the number of unintended gadgets. Onarlioglu et. al. came out with an idea of using a random key to encrypt the return address. At the point when the function is returned, the return address is decrypted with the same random key. The dynamic approach can be used at the binary level. This method says that the number of ret instructions must be the same as the number of the call. Pappas et. al. came out with a dynamic approach to check control-flow integrity, which was called kBouncer. When an internal system call is launched, kBouncer analyzes whether every ret was located after the corresponding call site of the call function. Though dynamic approaches are suitable both at the binary level and the source code level, monitoring the process of binary is a significant overhead on the performance and resources.

Control-flow integrity is not implemented in real life because of its performance overhead issues. Moreover, recent research shows that CFI is not as safe as depicted in theories. Carlin et. al. proposed using control-flow bending to bypass control-flow integrity.

IV. Benign uses of ROP

Return-oriented programming has always been considered as an exploitation technique. However, recent research showed that ROP can also be applied in benign uses like program steganography, code integrity verification and software watermarking.

A. Program Steganography with ROP

The goal of steganography is to hide certain instructions that are non-existent until program execution. The reasons for combining ROP and Steganography are: 1) Most existing steganographic technique violates W XOR X security model or mandatory code signing security mechanisms; and ROP can be used to bypass DEP guard; 2) A disassembler will never pick up unintended gadgets in a binary; which can be thought of as a way to hide messages inside a program until its execution. The flaw in this secret hiding method is that it can be flagged as an attack attempt by the security mechanisms. RopSteg showed the possibility of adopting ROP in security use.

B. Code Integrity Verification using Return-Oriented Programming

Dennis Andriesse et. al. proposed Parallax, which is another non-attack use of return-oriented programming. The idea behind Parallax is that a single gadget error would result in the failure of the attack. Parallax augments the code that needs to be integrity verified and then uses these gadgets to generate verification function. Since a single instruction change in the verification function would lead to its failure, parallax can be used to identify whether the code is modified or not. If the verification function returns a failure signal, it would lead to a program crash and eventually will halt the program execution. Parallax thus converts the integrity of code into the integrity of the ROP chains.

The author claimed that Parallax is the original research which uses return-oriented programming techniques to check for tampering inside the program, and Parallax can protect up to 90% of the code bytes, including the control-flow instructions, and has a performance overhead of only under 4%. Also, parallax can be used at the binary level and can protect non-deterministic code regions. Unfortunately, the use of parallax is also viewed as an attack attempt by the existing security mechanisms for preventing ROP attacks.

C. Software Watermarking using Return-Oriented Programming

Previously, software watermarking uses special data structures and instruction patterns, which might create suspicion. Also, the code inserted in the watermarked program has independent control flow from the other parts of the program, so that an adversary can successfully use dependency inspection to find the watermarking. In order to solve these issues, Haoyu Ma et al. came out with the idea of code reuse in return-oriented programming. The practical implementation of this watermarking can be categorised into four steps: 1) Extracting gadgets which will be useful in watermark generation. The author uses shared libraries to find suitable gadgets instead of the code execution area because using gadgets in the code execution area may look suspicious for short length programs. 2) Generating suitable payload. The method breaks the watermarking payload into various functions of the program, and these functions are termed as “carriers”. A processor tracer is used to find functions that can act as “carriers”. Using carriers, the author is able to embed just a small piece of code in each carrier that controls few gadgets, significantly decreasing the suspicion raised since the inserted code is unimportant compared to the size of the original carriers. 3) Chaining these “carriers” with special gadgets. In order to chain the payload pieces, stack-shifting gadgets are appended at the end of payloads so that each of the segments is responsible to relocate the current stack frame in the right manner to the correct memory address of the corresponding instruction. 4) Triggering ROP with function pointer overwriting. The experiments depicted that this method is suitable for generating watermark semantics which is untraceable even by powerful static analysis tools, and the code-reuse method generates no suspicious data structures till current literature. Similar to other benign uses, this technique is also flagged as an attack attempt by the ROP defense mechanisms.

Although return-oriented programming was initially proposed as an attack methodology, it can also be used for benign purposes. ROP utilises code snippets already present inside the program during its execution. It has a strong resistance to static code analysis.

V. THE FUTURE OF ROP

In real-life scenarios, return-oriented programming is commonly combined with memory leakage. An arbitrary memory read caused by information leakage enhance the success rate of ROP attacks. In 2017, Ben Gras proposed the idea of bypassing ASLR with MMU [5]. Therefore, how to prevent memory leakage at software and hardware level is a recent research direction.

Although a large number of defense mechanisms have been proposed in recent years, only ASLR is adopted as a full-proof security mechanism in modern-day operating systems. All the existing defense mechanisms discussed above are entirely theoretical, and most of them are not transparent to the software developers which makes these prevention techniques not applicable in the real environment. Therefore, how to make these security measures implemented in a real software environment is also a research direction.

VI. CONCLUSION

Return-oriented programming was initially thought of as an attack exploiting buffer overflow vulnerability. In the last decade, in-depth research has been carried out on ROP attacks and mitigation strategies. In this paper, the concept of return-oriented programming and a brief classification of existing defense mechanisms are discussed.

Return-oriented programming in the current computer architecture and software environment is unable to be eliminated completely. ROP is also proved to be Turing complete which makes it more offensive. Therefore, it is necessary to explore the binary code reuse technology in-depth.

ACKNOWLEDGMENT

The Term Paper is completed under the guidance of Ms. Sonali, Assistant Professor, School of Computing, IIIT Una.

REFERENCES

[1] Y. C. Gao, A. M. Zhou, and L. Liu, “Data-execution prevention technology in windows system,” Information Security & Communications Privacy, 2013.

[2] A. A. Sadeghi, S. Niksefat, and M. Rostamipour, “Pure-call oriented programming (pcop): chaining the gadgets using call instructions,” Journal of Computer Virology Hacking Techniques, pp. 1–18, 2017.

[3] Y. Ruan, S. Kalyanasundaram, and X. Zou, “Survey of return-oriented programming defense mechanisms,” Security & Communication Networks, vol. 9, no. 10, pp. 1247–1265, 2016.

[4] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” Acm Transactions on Information & System Security, vol. 15, no. 1, p. 2, 2012.

[5] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “Aslr on the line: Practical cache attacks on the mmu,” NDSS (Feb. 2017), 2017.