A Survey on Return-Oriented Programming

HANIEH BAGHERI
hanieh.bagheri@os3.nl

May 28, 2013

Abstract

The focus of this research is studying Return-Oriented Programming (ROP), which is a technique for exploiting software vulnerabilities. The reasons and motivations behind proposing ROP, the mechanism of working, automation tools, detection and prevention methods and new advances in ROP attacks are some of the important topics covered in this report.

1 Introduction

Software vulnerability is an important source of security risk in every computer system. A simple software bug, such as improper boundary-checking, can give the control of a whole system to an attacker. Despite many advances in defense mechanisms against software exploitation, buffer overflow and other kinds of application threats are still big security issues.

Using unsafe programming languages like C/C++, which do not check the value boundaries by default, is the most important reason behind the vulnerable software. One obvious solution can be using safer programming languages like Java, but because of legacy reasons and good performance of C/C++, a wide range of software, including operating systems, still use these languages. As a result, every computer system runs many programs developed using such a C-like language, which makes the whole system vulnerable to software attacks.

By calling some specific widely-used functions in C (gets, scanf, etc.), if the programmer has forgotten to check the boundaries, a malicious user can feed the input with longer values than expected by the program and launch a buffer overflow attack. If an attacker exploits a buffer overflow vulnerability, he can run a shellcode to do whatever he wants.

Emergence of protection mechanisms, such as non-executable memory pages ($W \oplus X$), which marks memory pages either as writable or as executable, made the attackers to come up with a more complex software attack. Return-to-libc is a major attack that can bypass the non-executable memory protection. In this kind of attack, instead of injecting code to the stack, the attacker uses the existing functions in the dynamic libraries. Calling the existing functions one after another, enables the attacker to execute an exploit, but he
is not free enough to run any arbitrary code. The reason is that he is limited to executing sequential functions linked to the program and cannot use branches or jumps.

ASLR is another protection mechanism, which can defend against a return-to-libc attack. This task is done by randomizing the base address for the code segments.

Return-Oriented Programming (ROP) is a pretty novel exploitation technique, which can bypass both $W \oplus X$ and ASLR protections. Using ROP, the attacker will be able to execute any arbitrary program without any need to code injection. In ROP, any desired code can be constructed from the code chunks (ROP gadgets), which are already present inside the memory.

1.1 Research Objective

The main goal of this research is to study the return-oriented programming technique. To get a better understanding of the reasons and motivations behind development of this technique and the way it works, we briefly review some older techniques, such as buffer overflow, return-to-libc and borrowed code chunks. We also study the offered protections against these exploitation techniques and the methods that can bypass them. Then, ROP capabilities, features, working mechanism, automation tools, etc. are discussed more in detail. We continue by presenting the suggested defense mechanisms against ROP. Finally, the most recent ROP improvements are introduced, followed by the conclusion.

1.2 Research Questions

This research is mainly aimed to answer the following question:

“What is the state-of-the-art in return-oriented programming?”

The sub-questions that will be answered in this report will be:

- What is return-oriented programming (ROP)?
- Why and how is ROP developed?
- What protections are offered for ROP detection and prevention?
- How ROP is improved over the last few years?

1.3 Outline

The related works to this research, including buffer overflow, return-to-libc, borrowed code chunks and the proposed protections against them are discussed in Section 2. In Section 3, we will describe the ROP features, capabilities and working mechanism. Then, we will explain about the proposed defense mechanisms against ROP in Section 4 and 5. In Section 6, improvements in ROP attacks, which refine the original ROP functionalists, are introduced, followed by the conclusion of the report in Section 7.
2 Related Works

2.1 Buffer Overflow

Buffer overflow is the classic software vulnerability, introduced in “Smashing the stack for fun and profit” by Aleph One in 1997 [1].

In essence, the buffer overflow is a program bug, which happens as a result of inappropriate boundary checking in programming languages without automatic bound-checking. As we know, most of the operating systems are written in C/C++ and there are many applications in every system, which use dynamic C libraries. There are some functions such as gets(), strcpy(), strcat() and scanf(), which in case being called without proper input bound-checking, can cause a buffer overflow vulnerability.

The ultimate goal of a buffer overflow attack is to inject a shellcode to the memory and gain the control of the system. A shellcode is a stand-alone program, usually written in assembly language, which can be executed in memory without any other code dependency.

In a classic buffer overflow attack, the attacker overflows a buffer on the stack, writes a malicious code inside the memory and overwrites the return address of a called function with the address of the injected shellcode.

To see how a buffer overflow attack works, first, we need to explain what happens when a function is called. In order to be able to keep track of the program flow, the processor uses a stack. By calling each function a stack frame is pushed and when returning to the original function, the frame is popped from the stack.

![Figure 1: The Stack Frame](image)

Figure 1: The Stack Frame [6]
Figure 1 shows a pushed frame on the stack, when calling a function [6]. As we see, the stack grows downward (from higher values of memory towards lower values) and the stack pointer register always points to the top of the stack. As a result, the stack pointer always points to the lowest memory address in the stack. When a function is called, the order of pushed values onto the stack is as follows:

- function arguments
- return address, which is the address of the next instruction after the function
- saved base pointer
- local variables defined inside the function

Normally when the function is terminated, all of these values are popped from the stack and the value of the return address is loaded to the instruction pointer register, so the control of the program will go back to the caller of the function.

Figure 2 shows how a buffer overflow vulnerability can be exploited.

1. A vulnerable program is started by a legitimate user.
2. The vulnerable function asks the user to feed the program with the appropriate input.
3. The attacker enters a longer value than expected as the input to the program. The input value will be stored in a local buffer on the stack. The entered value is consisted of an arbitrary data that fills the stack to reach the return address, the address of the new return and the target shellcode.
4. After getting the input from the user, the function continues to operate in its normal way.
5. The function reaches the return instruction.

6. The return instruction redirects the control flow to the injected shellcode, because the attacker has changed the return address to point to the memory address of the injected shellcode (step 3).

7. The injected code is executed and does the intended malicious activity.

2.2 Protections against buffer overflow

One defense mechanism against buffer overflow attacks is using $W \oplus X$. Using $W \oplus X$, each page of memory can either be writable (W) or executable (X). Therefore, everything written to the stack by the attacker is non-executable. Since the attacker injects the shellcode to the stack, the code cannot be executed.

$W \oplus X$ can be implemented in both software and hardware. PaX patch for Linux and Data Execution Prevention (DEP) for Microsoft Windows are two examples of implementing $W \oplus X$ at software-level. Intel and AMD have added a per-page execute disable bit to their processors to support $W \oplus X$ at hardware-level (XD bit in Intel and NX bit in AMD) [7].

Although $W \oplus X$ is a powerful protection against buffer overflow exploits, there are still methods that can bypass $W \oplus X$ (Some of the software attacks that can bypass this mechanism will be discussed in the next sections of this report).

To provide a better protection against buffer overflow, some security extensions can be added to standard compilers. A useful extension can be using stack canaries. A canary value is placed near sensitive data and it is regularly checked. Any changes in the value of a canary can be considered as a sign of a buffer overflow attack. StackGuard [2] and ProPolice [3] are two well-known implementations of stack canaries. Although using compiler extensions can be a very secure solution, having lots of canaries puts a lot of overhead on the compiler and degrades the performance.

2.3 return-to-libc

In order to bypass the $W \oplus X$ mechanism, Solar Designer proposed the return-to-libc attack in 1997. This attack utilizes the existing functions in the memory, instead of injecting new codes. In return-to-libc attacks, the attacker tries to run functions from libc, the standard C library. He overflows the stack by overwriting the return address of the called function with the address of a libc function. This attack is called “return-to-libc”, because the called function is returned to a system’s library, which is authorized to execute the injected code.

System() and execute() are two functions which are usually used in return-to-libc. The former executes a new program within the current running program and the later stops the current program and starts a new shell. For example, system("/bin/sh") executes

\footnote{http://seclists.org/bugtraq/1997/Aug/63}
the “/bin/sh” file while the original program is still running. Another useful function in return-to-libc attacks is exit(), which is used to exit a program.

Figure 3 shows the needed steps to run a return-to-libc attack [6].

Compared to the buffer overflow attack, the main difference is in step 3: while overflowing the buffer, the attacker overwrites the return address with the memory address of the “system” function. In step 6, the called function tries to return to the caller, but since the return address is overwritten, the “system” function will be called with ”/bin/sh“ as its argument in step 7. In this step, the desired shellcode is executed, then in step 8, the exit() function is called to terminate the program.

Return-to-libc attacks are able to bypass $W \oplus X$, but still have some limitations: The attacker is limited to the available functions in libc and might not be able to run every arbitrary code by calling the existing functions. To be able to do so, other facilities such as conditional branches are needed. Another issue is that if the commonly used functions in return-to-libc are removed from libc, using this attack will become very hard.

2.4 Protections against return-to-libc

Address Space Layout Randomization (ASLR) is a technique, which can provide protection against the return-to-libc attacks. ASLR randomizes the base address of loaded codes in the memory. It means that the location of stacks, heaps and linked libraries is not a constant address in the memory and will change in different executions of the same program.

Although some limited form of address randomization was used in 1997, the term ASLR was for the first time used by PaX project in 2001. Nowadays, most of the operating systems, including Windows, Linux, OpenBSD, Android, etc. have implemented ASLR [4].
2.5 Borrowed code chunks

To protect the memory against return-to-libc attacks, some functions are removed from libc. Moreover, by introduction of 64-bit CPUs, the function arguments are loaded into registers, instead of being pushed on the stack. As a result, the attacker might not be able to execute a libc function to run a shellcode located in the memory. To become able to do this, first he needs to load the injected shellcode into a proper register.

To overcome the mentioned problems, Krahmer (2005) [8] started to use chunks of functions instead of calling a function. The designed attacks aimed to find some functions from the library, which pop instructions from the stack, followed by a “return” instruction to load values from the stack to the desired registers. By chaining multiple sequences ended with return, it is possible to run a program.

DEPlib, a library developed by Pablo Sole, uses this idea. This library automates the proposed idea, but it does not offer any facilities for performing loops and conditional branches.

3 Return-Oriented Programming (ROP)

3.1 What is ROP?

Return-Oriented Programming (ROP) is a technique introduced by By Shacham in 2007 [5]. This technique provides the attacker with the possibility of executing codes despite the presence of security defenses such as $W \oplus X$ and ASLR.

Similar to return-to-libc and borrowed code chunks, ROP is a code-reuse techniques [9]. It means, ROP uses the available codes in libc or any other code linked to the program, to redirect the control of a running program to another location of the memory. The main feature that makes ROP distinguished from the previous techniques, is that using ROP, the attacker is not limited to functions and the linked programs and can execute any arbitrary malicious code, without need to injection of any code, calling any function or modification of the existing code. In other words, ROP is as strong as a Turing-complete machine.

ROP uses small sequences of assembly instruction and chains them together to build the building blocks required for executing a program. There are 2-5 instructions in each sequence, where the last instruction of every sequence is the “return” instruction. This is why this technique is called “return-oriented programming”. A set of these sequences that can perform an atomic task like load, restore, an arithmetic operations, etc., is called a “gadget”. By finding the complete set of gadgets necessary for the basic operations in a system, an attacker will be able to execute any desired action, just by combining the appropriate gadgets.

\(^2\text{http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/}\)
3.2 ROP and different architectures

ROP was originally proposed by Shacham to be used for exploiting software vulnerabilities in an Intel x86 system [5]. After a while, Buchanan et. al. published a paper about extendability of ROP to SPARC [10].

Intel x86 and SPARC are from two extremely different families of architectures [10]:

- x86 uses memory-register Complex Instruction Set Computing (CISC), while SPARC uses load-restore Reduced Instruction Set Computing (RISC)
- x86 instructions are variable-sized and unaligned, while SPARC instructions are fix-sized and aligned
- x86 uses a limited amount of registers, but in SPARC many registers are accessible
- function calling in x86 is more flexible, compared to SPARC, which uses strict register-based function calls
- x86 pushes function arguments and the return address to the stack, while SPARC passes them to the registers
- In contradiction with x86, SPARC uses delay slots for transferring the control flow

It is an interesting observation that despite all the differences between Intel x86 and SPARC, an attacker can take advantage of ROP to attack both kinds of systems. Later on, it was proven that ROP can be used in a wide range of architectures, including as Atmel AVR [11], PowerPC [12], Z80 [13] and ARM [14]. Therefore, it can be claimed that ROP is an architecture- and operating system-independent exploitation mechanism.

3.3 How to execute an ROP attack?

Figure 4 shows the needed steps to run a return-to-libc attack [9].

![Figure 4: A return-oriented programming attack](image)
Most of the steps are similar to ones needed for the attacks discussed in the previous sections. The only differences are the followings:

In step 3, the attacker feeds the program with a long string as the input. The input overflows the buffer and overwrites the return address of the called function with the first return address, followed by the other ROP return addresses. Each return address points to the start address of an instruction sequence.

In step 7, the first intended instruction sequence, resided in memory address “ROP Return 1”, is executed. When the instruction sequence is terminated, the return instruction redirects the control flow to the instruction sequences located inside “ROP Return 2” memory address. Similar operations will be performed in step 8 and 9. Step 7, 8 and 9 can cause a malicious behaviour, which was not intended by the original program.

3.4 Automatic gadget finding

As discussed before, using ROP, one can find a set of gadgets as powerful as a Turing-complete machine. However, finding the needed gadgets inside a given program is a tedious work. So some automation tools are developed to make the gadget finding process more convenient.

The most well-known tools used for gadget finding are the followings:

- **ROPgadget**: This tool uses a binary file as input and looks for the useful gadgets until reaching a complete set of gadgets. Auto-roper feature provides the capability of building any arbitrary payload automatically with the found gadgets.

- **ROPEME**: ROP Exploit Made Easy or ROPEME is a proof-of-concept ROP automation tool. It contains a set of Python scripts to generate ROP gadgets and payloads.

- **Msfrop**: It is a metasploit tool that can collect, export, and import ROP gadgets from various file formats, including PE, ELF, Macho.

- **ROPC**: It is a developed tool based on a model named Q, proposed in a research done by Schwartz et. al [15]. According to ROPC’s github repository: “ROPC is an example of a Turing complete ROP compiler. It’s not supposed to be practical: generated payloads are too big to be useful (emulating a stack is very space consuming). Generated programs are pure: nothing is ever executed from executable memory.”

Among the discussed tools, ROPgadget seems to be the most comprehensive and well-developed tool for practical usage.

---

3http://shell-storm.org/project/ROPgadget/ retrieved on May 26, 2013
4 Protections against ROP

Since ROP does not need to inject a shellcode, it can easily bypass $W \oplus X$ and since it does not call any function, removing functions like `system()` from libc cannot be a problem for the attacker. However, there are some general methods proposed for ROP mitigation:

- **Address Space Layout Randomization (ASLR):** As discussed before, ASLR causes stacks, heaps and libraries to be loaded in random addresses of the memory. Similar to the case of return-to-libc attacks, if the ROP attacker does not know about the base address of the target codes, he will not be able to overwrite it. So in theory, ASLR is a very good defense against ROP attacks. However, in reality, the entropy of randomization is not high enough. Therefore, using a brute-force or a memory disclosure attack, it will be possible to find out the location of the target code. Moreover, all of the code segments are not using ASLR, so in practice the attacker can use the non-ASLR codes in order to perform an ROP attack [18].

Since in ASLR, just the base pointer is randomized, by discovering the start address of a code segment, the attacker can easily calculate the rest of the addresses. So recently some other methods are proposed to perform randomization at instruction-level [19, 20, 21]. However, instruction-level ASLR is vulnerable to memory disclosure attacks [22].

- **Compiler extensions:** One defense mechanism against ROP is using security extensions for the standard compilers. Similar to case of buffer overflow protections, using stack canaries for ROP can be very useful, but the usage is limited to open source codes. Moreover, these extensions force a considerable amount of overhead to the compiler, which in turn will cause performance degradation.

A compiler-based defense against ROP is offered by Jink Lu et. al [16]. This method recognizes return instructions at compile-time and completely removes them in a running OS kernel, using three key techniques: return indirection, register allocation, and peephole optimization. It is claimed that this method has a good performance and does not cause a considerable overhead.

G-Free [17] is another compiler-based method, which removes all the gadgets from the binary at compile-time. This is done by removing all the unaligned free-branch instructions inside an executable file.

- **Programming Languages:** Another defense can be using a programming language like Java, with built-in boundary-checking capabilities. The problem with this category of defense mechanisms is porting the legacy programs to the new chosen language. Another problem is related to performance reasons: most of the operating systems are developed using C and it does not seem to be a good idea to develop an operating system using a modern language such as Java.

- **Software diversity:** This method uses a diversification engine to extract n different compiled versions from the same source code. Using this method, in case of being exploited, just one of the instances is affected. However, a successful ROP attack needs to exploit all of n instances. This method does not seem to be practical,
because it needs to run n instances from every library function and linked program [23, 18].

5 ROP detection

The methods used for ROP detection can be classified into two groups [24]:

1) The ones that look for frequent usage of the return instruction: DROP [25] is an example of this type of detection mechanism. It considers three sequences of five or fewer instructions followed by return, as a suspicious behavior.

2) The ones that try to match every return with a call instruction to make sure that the stack behaves in LIFO manner. The methods of this class can be categorized into:
   a) compiler-based solutions
   b) instrumentation-based solutions
   c) the ones that use just-in time compilation
   d) hardware-facilitated solutions

A recent publication [26] has offered another detection approach, which tracks the position of the return addresses to discover the ones close to each other.

6 ROP without return

In a paper named “Return-Oriented Programming without Returns” published by Checkoway et. al. at 2010 [24], it is shown that in both x86 and ARM architectures, it is possible to perform ROP attacks without using return instructions. Instead, it is possible to use some other return-like instructions (jmp *%eax in x86 and blx r3 in ARM) to get the same functionality from the attack. Since this method does not use return instructions, most of the developed defense mechanisms are not able to detect or prevent the attack. An examples of a bypass-able detection method is the one which detects a lot of nearby return addresses [26]. It is proved that this attack has all the functionalists provided by a normal ROP attack, so it is also a Turing-complete machine.

The difficulties of this method are [18]:
   1) the used register has to be initialized beforehand
   2) Unlike return instruction, which automatically updates the stack pointer, using jump, it should be updated in case of need.

Protection against this kind of attack is very difficult and needs a general solution like Control-Flow Integrity (CFI) [27].

7 Conclusion

The main goal of this research was to answer the following question: “What is the state-of-the-art in return-oriented programming?” To be able to answer this question, the historical trend of software vulnerability exploitation was studied from
buffer overflow attacks, followed by return-to-libc and borrowed code chunks exploits. Meanwhile, the different proposed methods to defense against each attack were investigated.

The most important protection against software vulnerabilities are $W \oplus X$ and ASLR. Generally speaking, $W \oplus X$ was invented to defense against buffer overflow attacks. It marks every memory page as either writable or executable, so the attacker cannot execute the injected shellcode. To be able to defense against return-to-libc family of attacks, ASLR was offered, which randomizes the base address of code segments. To bypass ASLR, the attacker needs to predict the start address of the target code.

Return-oriented Programming (ROP) is a technique proposed to bypass the offered protections against the previous methods. Using ROP, the attacker will be able to execute any arbitrary program without need to code injection. ROP is as strong as a Turing-complete machine, it means any desired code can be constructed from the ROP gadgets, already present inside memory.

New advances in attack techniques, are followed by advances in detection and prevention mechanisms and this trend still continues.

References


