International Journal of Scientific Research in Computer Science, Engineering and Information Technology
ISSN : 2456-3307 (www.ijsrcseit.com)
doi : https://doi.org/10.32628/CSEIT206545

Return Oriented Programming - Exploit Implementation using Pwntools

Jayesh Zala
Computer Engineering Department, A. D. Patel Institute of Technology, Karamsad, Gujarat, India

Article Info
Volume 6, Issue 5
Page Number: 194-198
Publication Issue :
September-October-2020

Article History
Accepted : 05 Oct 2020
Published : 14 Oct 2020

ABSTRACT

A common approach to leverage software vulnerabilities in the contemporary operating system has been the Return-Oriented Programming(ROP) attack. Although protection mechanisms are involved in the OS, an attacker may execute arbitrary code with the support of ROP. A decade ago, Return Oriented programming was designed to solve the buffer overflow exploit security mechanisms such as ASLR, DEP (or W⊕X) by reusing the machine code in the form of gadgets that are stitched together to render a full assault on Turing. And it will take more complex efforts to conduct a Turing complete attack, and very little data is possible to perform it with raw input. Therefore, in this project, we are systematizing the interpretation of the new findings that can be used to carry out a full ROP attack with the help of pwntools python library.

Keywords : Return-Oriented Programming, ASLR, DEP, Stack Cracking Attack, CDECL, STDCALL, FASTCALL.

I. INTRODUCTION

Return-oriented programming is an improved variant of the Stack Cracking Attack. Normally, when an attacker deceives a stack by reaping the benefits of an implementation vulnerability, often a buffer overrun, these sorts of threats arise. It is a feature that enables an attacker to trigger unreasonable behavior in a program by transferring the program control flow without embedding any code. A return-oriented program chains together brief procedure sequences that are already present in the address space of a program, each terminating in a return statement. There are various security measures, one of which is well recognized as Data Execution Prevention is a safety mechanism of operating systems and virtual machines. It is just a memory management strategy that requires any page in a processor kernel address space either to be writable or executable, but not both. The other is ASLR, which is an acronym for Address Space Layout Randomization as well as being a common safeguard against ROP attacks. This works by arbitrarily shifting a program's fragments around in memory, stopping the intruder from calculating useful gadget addresses. Address space layout randomization structure is based on the low probability of an intruder guessing the positions of randomly located sections. By-the search storage, security is enhanced. Thus, as more uncertainty is present in different offsets, random sampling of the main memory is much more efficient. As a result, unmounting the ASLR is a typical illustration at the time of commencement. In this process, we attempted to execute an ROP attack on a binary

program for our own simplicity by deactivating ASLR before writing our program and executing the exploit script with the help of *pwntools* which is the python framework for CTFs.

II. BACKGROUND

ROP was meant to overcome the shortcomings of Buffer overflow, whereby the attacker was able to insert and execute his arbitrary code into the stack fragment, this was avoided by keeping the stack section non-executable and making it more difficult to introduce malicious code by adding ASLR. As a result, ROPs were designed to exploit existing security mechanisms and reusing the code.

2.1 Buffer Overflow (BoF): is an exception when a program overwrites the boundaries of the buffer when writing data to a buffer and overwrites neighboring memory locations. Buffers are storage spaces designated for storing data when moved from one portion of a process to another, or between processes. Buffer overflows can also be induced by malformed entries; when one assumes that all input data are lesser than the specific size and the buffer is formed to be that size, an unusual operation that produces extra data can enable the buffer to write beyond the buffer end. When this overwrites adjoining information or program code, this can result in erroneous program behavior, which includes memory access failures, invalid performance, and collisions. Modern languages usually synonymous with buffer overflows include C / C++, which may not provide built-in storage access or duplicating data in any portion of the storage and may not automatically validate that the data written to the array is underneath the boundaries of that array. If this overwrites neighboring data or program code, this can result in varying program behavior, including memory access failures, invalid performance, and collisions.

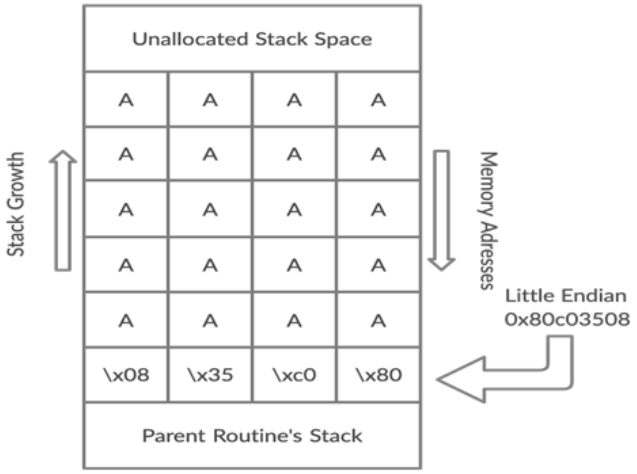


Figure 1: Stack buffer overflow at work

2.2 Calling conventions: This is a low-level scheme for how subroutines accept parameters from their caller and how they return the response. There are three main calling conventions that are used for C on 32-bit *x86* processors: CDECL, STDCALL, and FASTCALL.

2.2.1 CDECL: C declaration is the calling protocol that derives from the Microsoft C language parser which is used by many C compilers for the Intel processors. The caller clears the arguments from the stack in this protocol. Given below is the C pseudo-code:

```
int callee1(int a, int b, int c);
int caller1()
{
        return callee1(5, 6, 7) + 8;
}
```

Given below is the x86 assembly code of Intel syntax:

```
caller1:
  push   ebp
  mov    ebp, esp

  push   7
  push   6
  push   5
```

```
call    callee1
add     esp, 12
add     eax, 8

mov     esp, ebp
pop     ebp
ret
```

After the function call returns, caller1 cleans the stack.

2.2.2 STDCALL: The stdcall calling concept is a variation of the Pascal calling concept where the callee is in charge of fixing the stack, however the arguments are transferred to a stack in the R2L order, like in the C declaration calling procedure. The registers *EAX, ECX, and EDX* are designed for use through the procedure. The returned results are calculated in the EAX register. The called procedure cleans the stack, unlike c declaration. This implies that standard call does not accept lists of variable-length arguments.

2.2.3 FASTCALL: The FASTCALL call protocol also isn't fully common for all compilers, so that should be used with precautions. In the FASTCALL convention, its first 2 to 3 32-bit arguments were entered in the registers, the most widely used being *EAX, ECX,* and *EDX*. Here, below is the example of C function:

```
_fastcall int MyFunction4(int c, int d)
{
  return c + d;
}
y = MyFunction4(5, 6);
```

Will produce the following assembly code fragments for the called, and the calling functions, respectively:

```
MyFunction4:
  push    ebp
  mov     ebp, esp
  add     eax, edx
  pop     ebp
  ret

mov     eax, 5
mov     edx, 6
call    MyFunction4
```

2.3 Tools for gadget searching: In return-oriented programming, the main concept is to combine valuable instruction sequences from the code and chain these instructions.

2.3.1 Ropper: You may use ropper to display binary file information in various file types and search gadget to create sequences for various architectures (*x86 / x86 64, ARM / ARM64, MIPS*). The awesome Capstone System is used for disassembly of the ropper.

$ ropper --file <afile> --semantic "<any constraint>"

2.3.1 ROPgadget: ROPgadget allows PE, ELF, and Mach binary variants on the x86, x64, ARM, ARM64, and other architectures.

$ ROPgadget --binary <afile> --only "<gadget>"

III. WORKFLOW

The aim of this segment is to show the chaining of gadgets and with the help of *pwntools* framework of python exploit the *ELF* binary file.

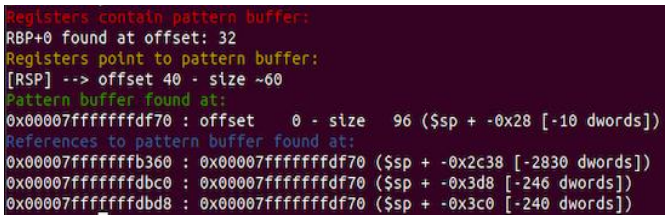
3.1 Source Code: Given below is the functional code of the binary file which we are going to exploit:

```
void main() {
        int buffer[32];
        memset(buffer, 0, 32);
        puts("Input your data");
        read(0, buffer, 96);
        return;
}

void win(void) {
        system("/bin/ls");
        return;
}
```

Here, we have to overwrite the buffer, call the win function and execute the system function with "/bin/sh" argument instead of "/bin/ls".

3.2 Find length of buffer and return address: We will use the *GDB-PEDA* to find the padding offset to the return address of the main function.



Here, we found that the offset value is *40* bytes, *x64* system checks the address is valid or not before popping it into *RIP*. Therefore, we found the offset with the help of the value presented in *RSP*.

3.3 Writing exploit script in python: We're going to exploit the *x64* system so our raw payload structure is offset + pop_rdi + bin_sh + system_addr. But we will ease our process with the *ROP API* of python *pwntools*.

```
from pwn import *

elf = context.binary = ELF('./binary')

rop = ROP(elf)

info("%#x system", elf.symbols.system)
system = p64(elf.symbols.system)

info("%#x /bin/sh", elf.symbols.binShString)
bin_sh = p64(elf.symbols.usefulString)

pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]
info("%#x pop rdi",pop_rdi)

rop.call(pop_rdi)
rop.call(system, [bin_sh])

payload =  b"A"*40
payload += rop.chain()

io = process(elf.path)

io.recvuntil('your data')
io.sendline(payload)
io.interactive()
```

We have used the ROP class of pwntools to create the instance rop and by utilizing its find_gadget method to find the address of pop rdi, ret instructions to add to the payload. After that, we called the call method to add the pop_rdi address and system address with bin_sh string as an argument. Finally, we called the chain method to chain together all the gadgets, added it with payload, and sent it to binary to get sh shell.

IV. CONCLUSION

Return Oriented Programming may have been a decade old and not many vulnerabilities have been documented using ROP, because this is a stealth feature that can not be detected by Intrusion Detection Systems or other Signature-based detection systems because it reuses the system's trusted library to execute malicious acts. ROP seems to be very limited, so in our project, we illustrated gadget chaining in the form of chaining pwntools functions, which is the same duplication that can be used to execute a Turing complete attack.

V. REFERENCES

[1]. "Smashing The Stack For Fun And Profit" by Aleph One
[2]. x86 calling conventions https://blog.csdn.net/Scotthuang1989/article/details/42969393
[3]. Calling conventions https://www.tfzx.net/article/6972315.html
[4]. "Penetration Testing with Shellcode" by Hamza Megahed.

[5]. “The advanced return-into-lib(c) exploits” by Ihsahn, "Alsvartr"
[6]. ROP Emporium https://ropemporium.com/
[7]. “Programming in ANSI C” by E. Balaguruswamy
[8]. pwntools documentation http://docs.pwntools.com/en/stable/
[9]. “Beginning Ethical Hacking with Python” by Sanjib Sinha
[10]. “Hacking: The Art of Exploitation” by Jon Erickson
[11]. “Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation” by Alexandre Gazet, Bruce Dang, and Elias Bachaalany
[12]. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes” by Chris Anley, Felix Lindner, and John Heasman
[13]. “When to use __fastcall” by Kent Reisdorph http://bcbjournal.org/articles/vol4/0004/When_to_use___fastcall.htm
[14]. GDB-PEDA https://github.com/longld/peda
[15]. “Beginning X64 Assembly Programming: From Novice to AVX Professional” by Jo Van Hoey

Cite this article as :

Jayesh Zala, "Return Oriented Programming - Exploit Implementation using Pwntools", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 6 Issue 5, pp. 194-198, September-October 2020. Available at doi : https://doi.org/10.32628/CSEIT206545
Journal URL : http://ijsrcseit.com/CSEIT206545