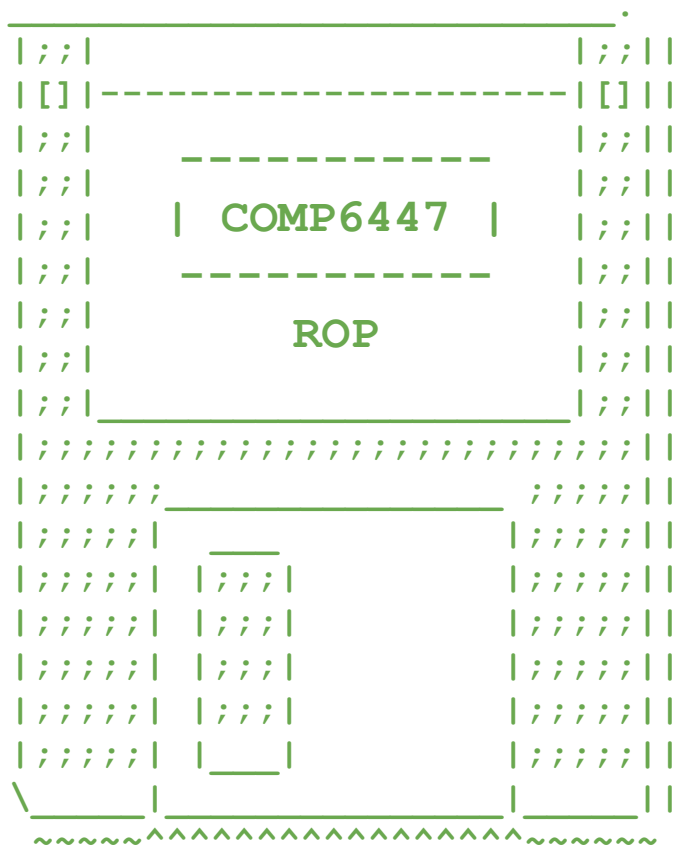

COMP6447
ROP

# Assignment

- Good luck on assignment (talk to tutors if you need help)
  - **Any questions / faq ask on Ed**
- Some people don’t have groups yet.
  - Fix this today - You will get zero in the midpoint submission otherwise

# Return Oriented Programming

- What is ROP
- Overview of what a **function** is
- Overview of what an **instruction** is
- How to ROP

# recap

So far we know

- Reverse Engineer Binaries
- Audit Source Code
- Exploit Buffer Overflows
- Bypass Stack Canaries
- Write and Execute Shellcode
- Exploit Format Strings
- Defeat PIE/ASLR

hopefully


YOU FIND A BUFFER OVERFLOW


THERE'S NO WIN FUNCTION


ROP

# What is rop

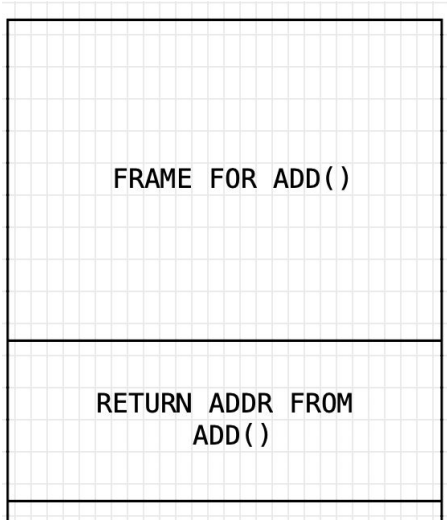
- Return Oriented Programming
- A turing completing method of writing programs without actually writing any code
  - **weird machine** is a computational artifact where additional code execution can happen outside the original specification of the program
- Instead of relying on shellcode/win functions
  - Take advantage of multibyte x86 instruction alignment
  - Chain together tiny functions to do a certain task
- Use the code already in the program

Why?

- Defeats NX / Code Signing protections

# How do we call a function?

call add()



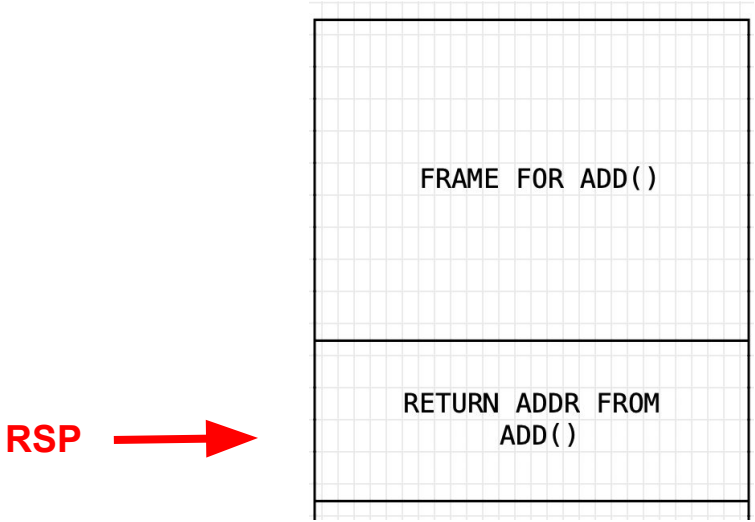

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}

int main()
{
   add(3, 4);
}
```

**ARGS**

**RDI: 3**
**RSI: 4**

# Finished executing. Now RSP is here

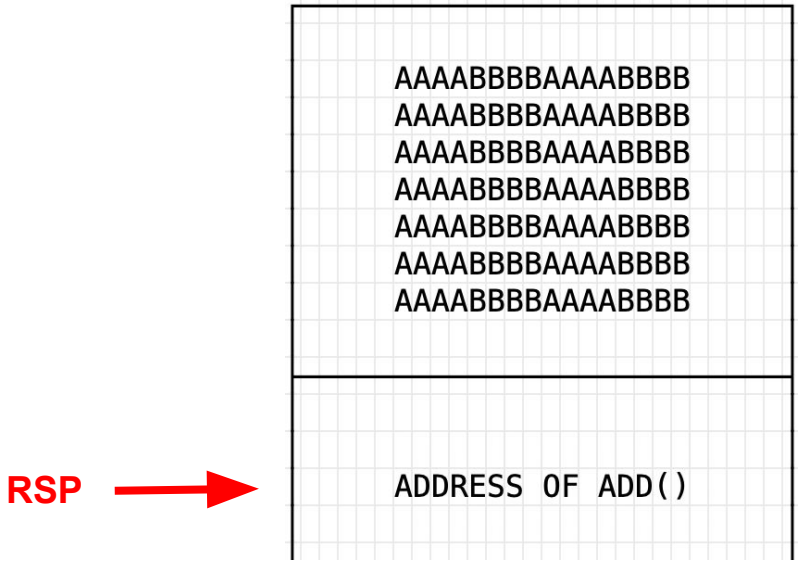



```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}

int main()
{
   add(3, 4);
}
```

# How can we call a function with a BoF?

**call add(01010101,02020202)**

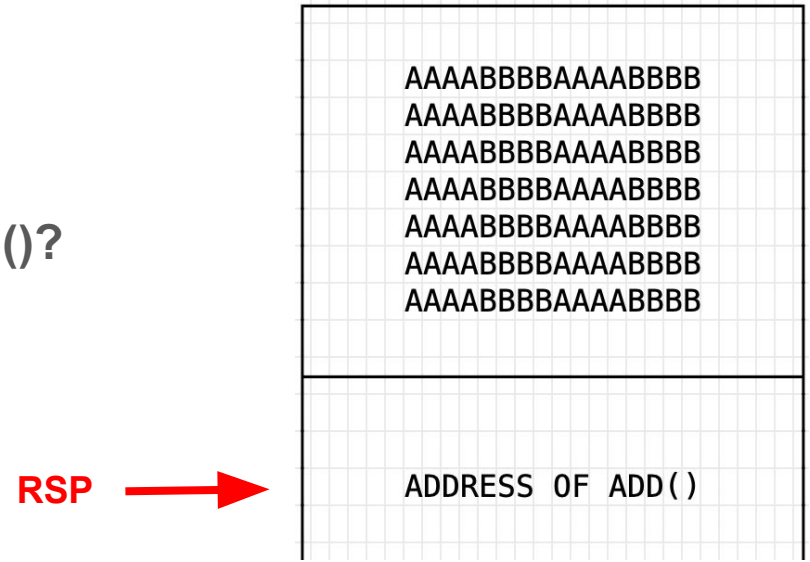


# How can we call a function with a BoF?

**call add(01010101,02020202)**

**How do we pass args?**

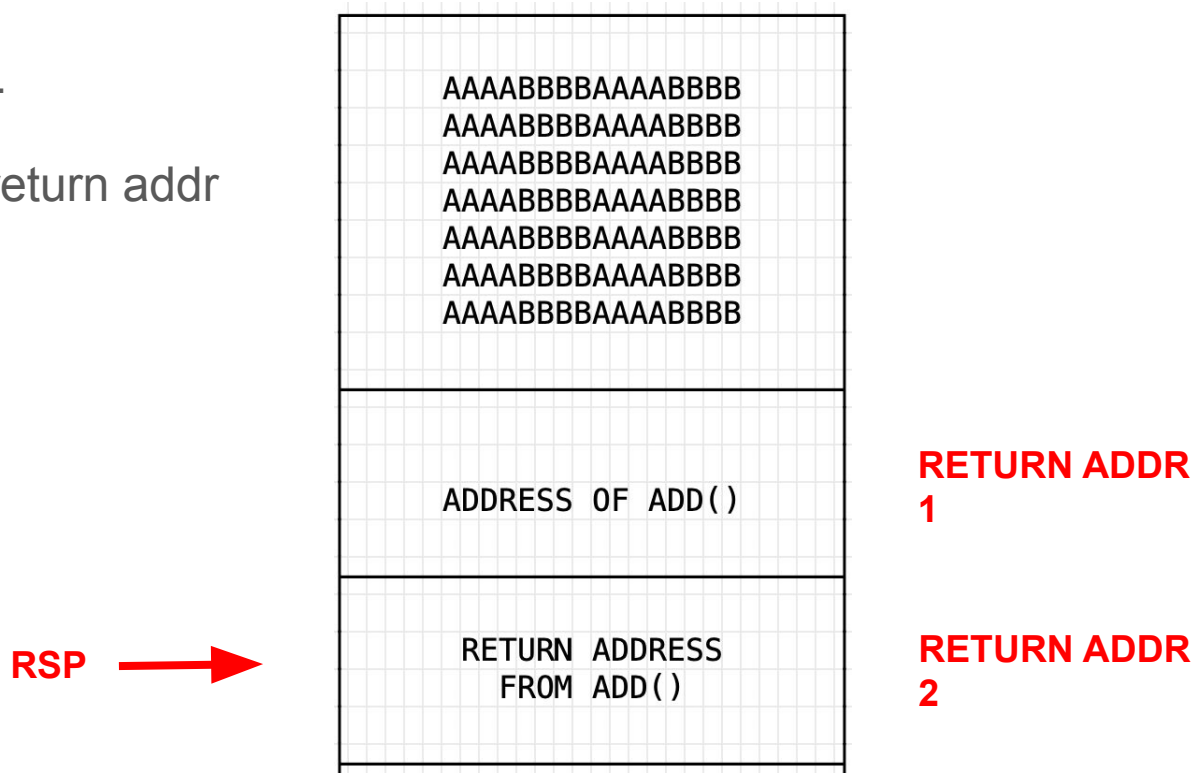
**How do we pass the return addr from add()?**



# Return Function

Chaining functions is easy.

Functions just expect the return addr to be next on the stack



# Args?

Remember: Args are passed in via registers.

We need a way to move arguments into registers **before** calling the function

We will come back to this. Keep this in mind

# What if I wanted to call the same function twice

With different arguments

add(01010101, 02020202)

add(03030303, 04040404)

How would this look?

# This is the problem ROP solves… Let's come back to this

> 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"


- There are a limited number of regions where we can actually execute code
- ASLR means we don't really know where our shellcode is in the HEAP/STACK
- NX means we can't even execute it if we did
- So where else can we redirect execution?

> TEXT section | Static Libraries | LIBC ==> **ROP**

# What is a function?

- A function is a block of organized, reusable code that is used to perform a single, related action
- A function is a block of reusable code ending with a **return statement**

This is technically a function...

```
sub_4c2:
mov     ebx, dword [esp {__return_addr}]
retn     {__return_addr}
```

# What do these all have in common

```
0x00000000004010f7: nop dword ptr cs:[rax + rax]; endbr64; ret;
0x00000000004010f6: nop word ptr cs:[rax + rax]; endbr64; ret;
0x000000000040119d: pop rbp; ret;
0x000000000040118d: push rbp; mov rbp, rsp; call 0x1110; mov byte ptr [rip + 0x2ebb], 1; pop rbp; ret;
0x00000000004011ba: push rbp; mov rbp, rsp; ret;
0x00000000004011dd: ret 0x90c3;
0x00000000004011d1: ret 0xc301;
0x000000000040105a: ret 0xffff;
0x0000000000401011: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x00000000004011c7: sub ecx, 4; ret;
0x00000000004012a5: sub esp, 8; add rsp, 8; ret;
0x00000000004011c6: sub rcx, 4; ret;
0x00000000004012a4: sub rsp, 8; add rsp, 8; ret;
0x00000000004010fa: test byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; endbr64; ret;
0x0000000000401010: test eax, eax; je 0x1016; call rax; add rsp, 8; ret;
0x000000000040100f: test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
0x00000000004011c0: xor edx, edx; ret;
0x00000000004011bf: xor rdx, rdx; ret;
0x00000000004011b9: cli; push rbp; mov rbp, rsp; ret;
0x00000000004012a3: cli; sub rsp, 8; add rsp, 8; ret;
0x0000000000401103: cli; ret;
0x00000000004011b6: endbr64; push rbp; mov rbp, rsp; ret;
0x00000000004012a0: endbr64; sub rsp, 8; add rsp, 8; ret;
0x0000000000401100: endbr64; ret;
0x00000000004010f5: hlt; nop word ptr cs:[rax + rax]; endbr64; ret;
0x0000000000401214: leave; ret;
```

# Gadgets

- “a **small** mechanical or electronic device or tool, especially an ingenious or **novel** one.” ~ dictionary.com

In ROP terminology

- A gadget is a **small set of instructions**, that together performs a certain task
- Most importantly, a gadget **ends in either a ret** or a jmp/call instruction
- We use these gadgets to construct a rop **chain**
- What does **RET** do?
    - It looks at where the **current Stack pointer** is looking, takes the value there, and **jumps** to that position
- We can point our execution towards these small gadgets, one after another...

# Look at x86 instructions

- An Instruction is a base building block in x86
- In x86 instructions can be between 1 and 15 bytes long
  - Dynamically sized based on how often they're used
  - 90 => NOP
  - F2 F0 36 66 67 81 84 24 12 34 56 78 12 34 56 78 => xaquire lock add [ss:esp*1+0x12345678], 0x12345678
- Instructions often overlap
  - 66 **90** => xchg ax, ax
  - **90** => nop
- Often can find instructions that aren't supposed to be there
  - But due to alignment not being an issue in x86, different instructions can be found in larger instructions

# Rop Chains

We can construct a **chain** of these small **functions/gadgets**

They all do small things.

- mov rax, rbx; ret
- syscall
- pop eax; ret

But if you chain tiny instructions together… you are pretty much writing shellcode

## Back to our previous example

We want to call a function, with two arguments**. Remember calling conventions**

**Arg1 goes into RCX**

**Arg2 goes into RDX**

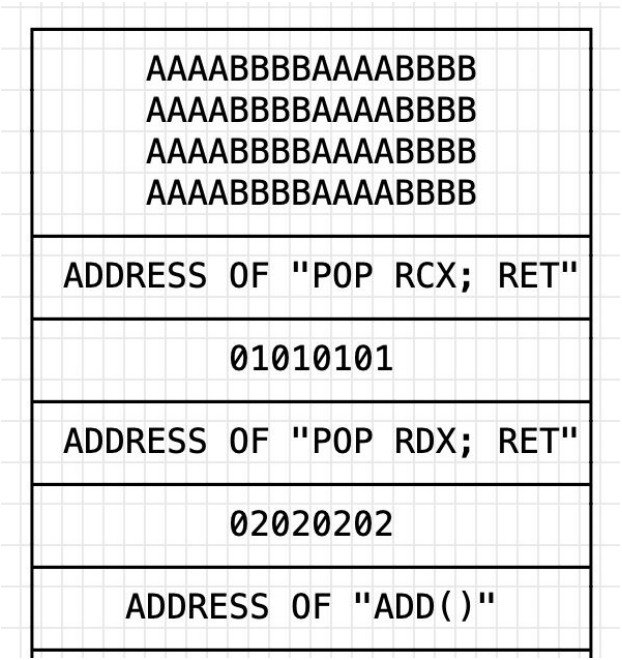
**Arg3 goes into R8**

**etc**

**pop**

**ret**

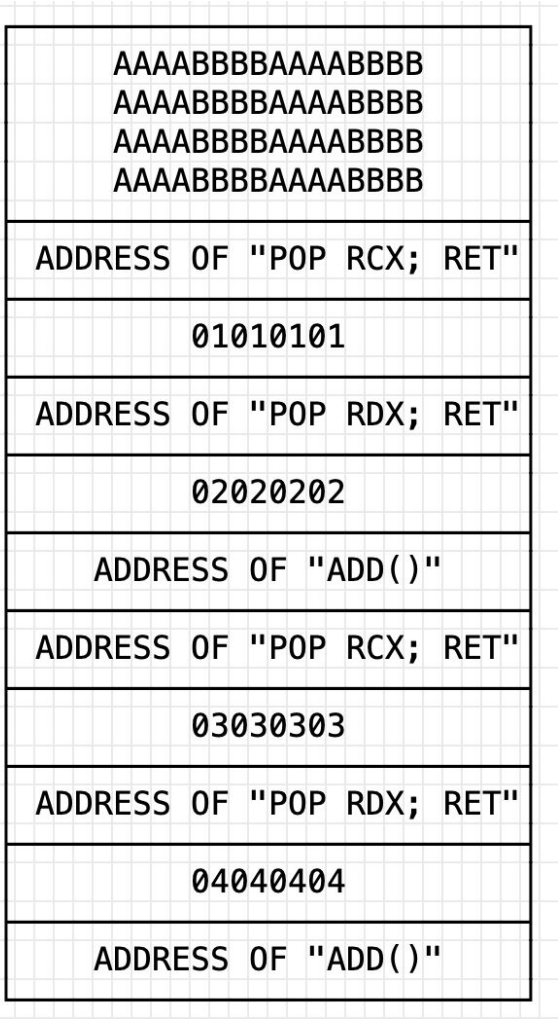
## call add(01010101,02020202)




Using our found gadget:

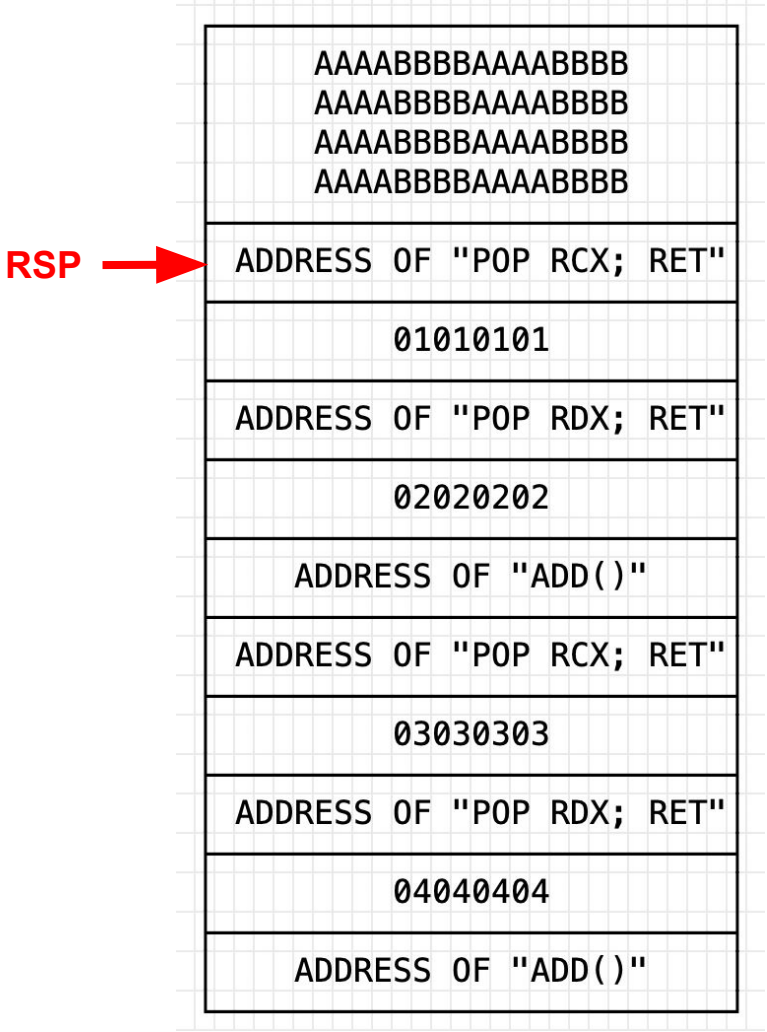
We can

- Set each Argument to our wanted value
- Call the function
- Rinse and repeat if you want to call more functions

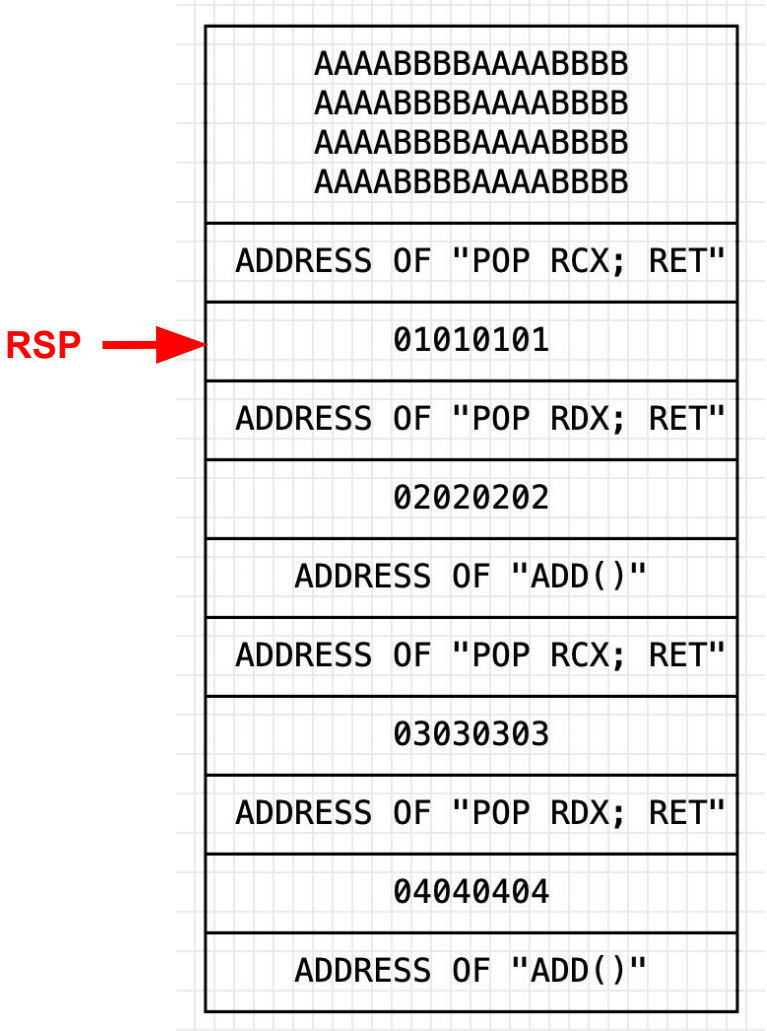



Rinse and repeat if you want to call more functions

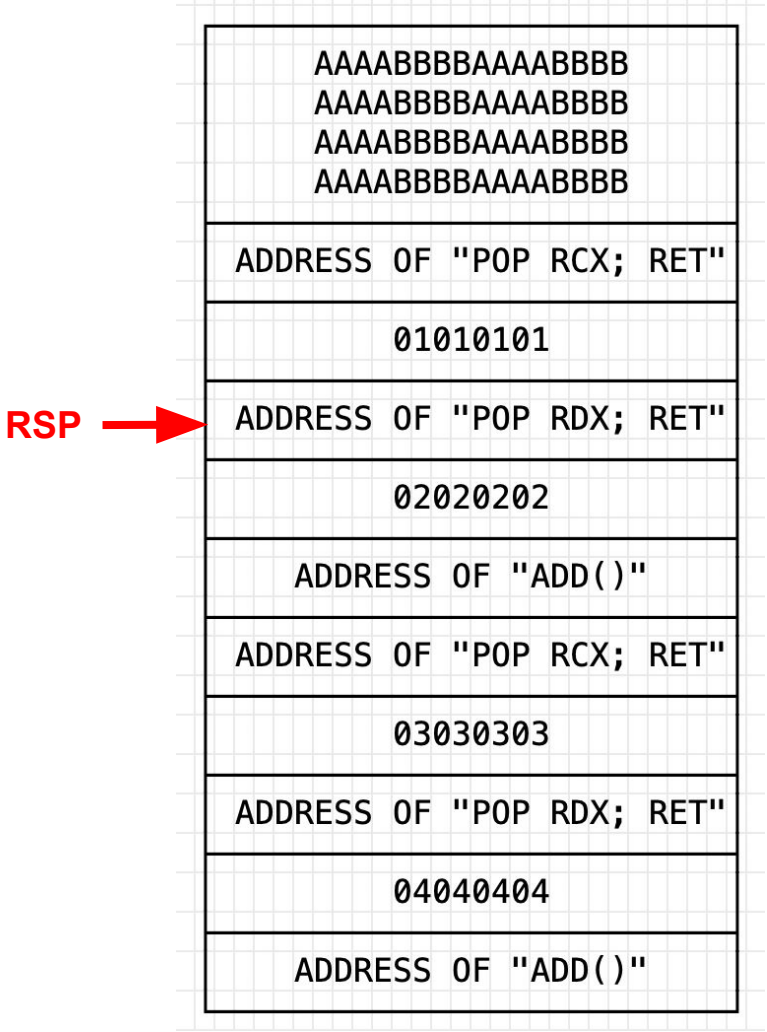
# Lets see how this works



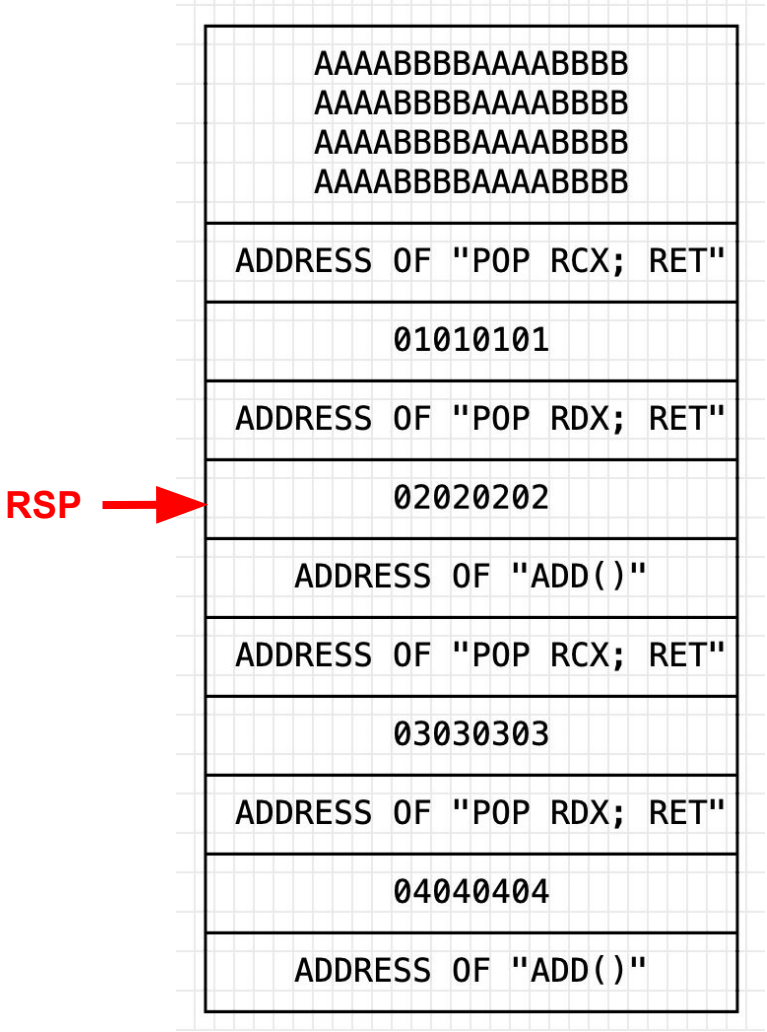

- Return from overflow function



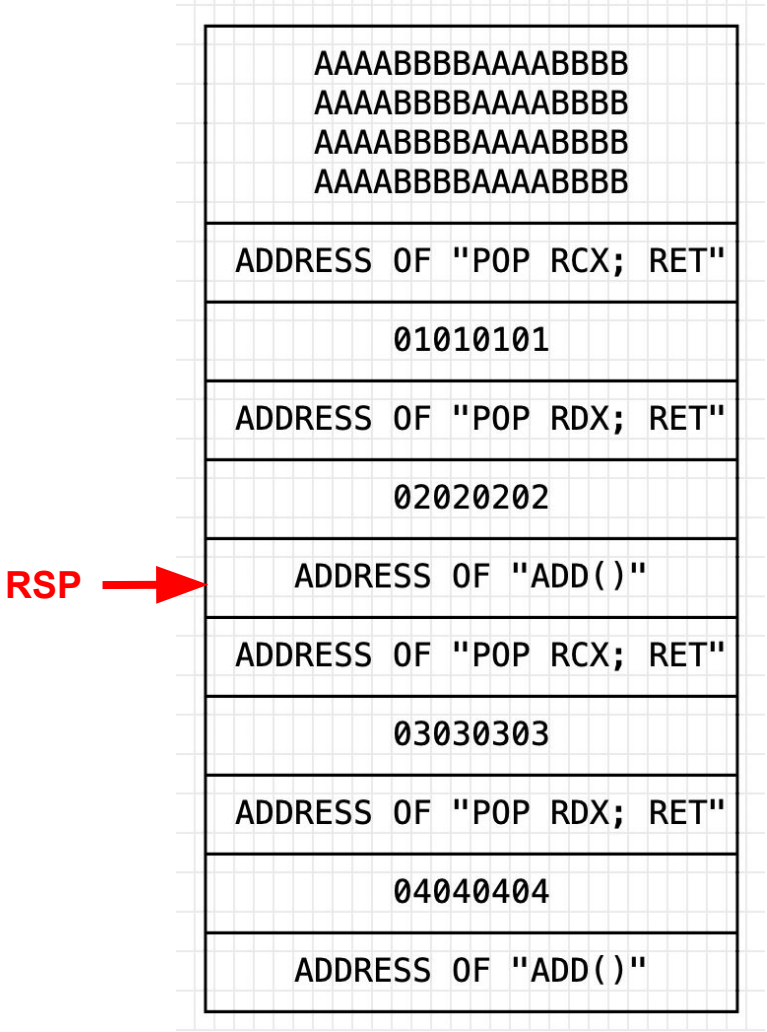

- Return from overflow function
  - POP top of stack (01010101) INTO RCX



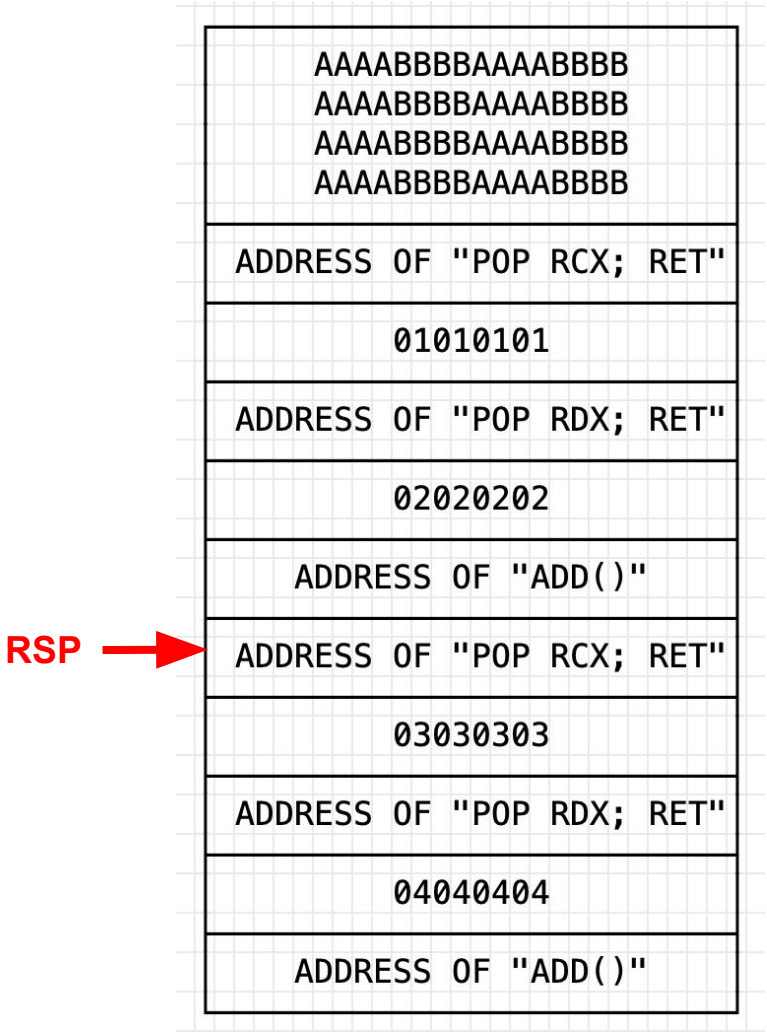

- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget



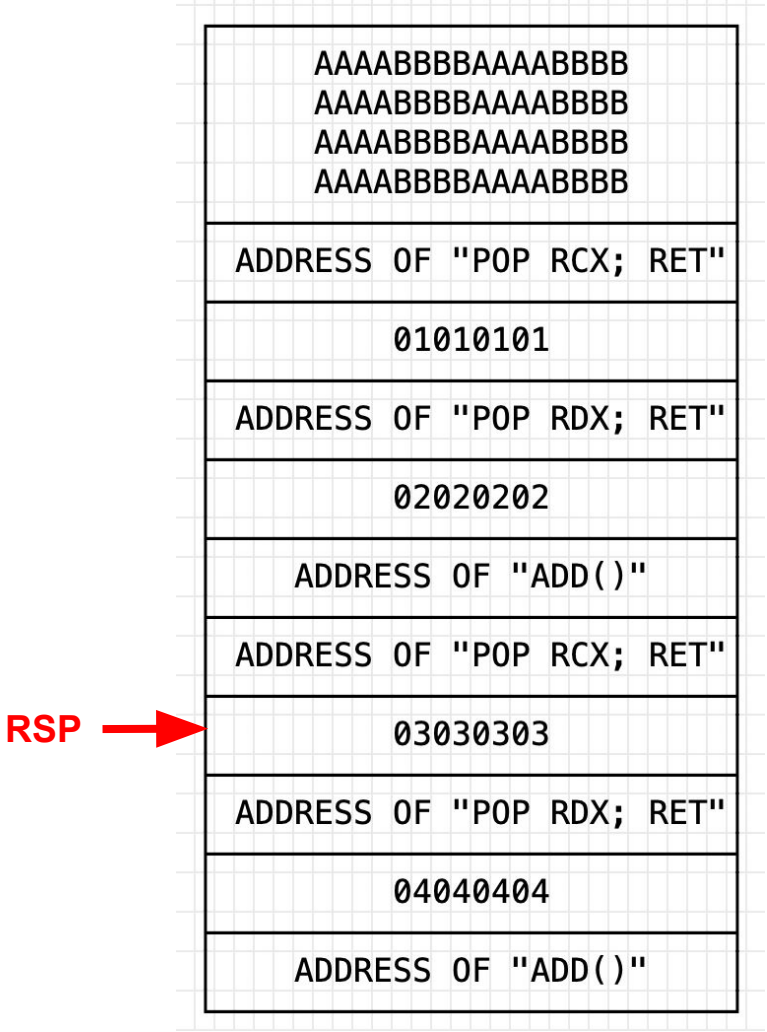

- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RCX



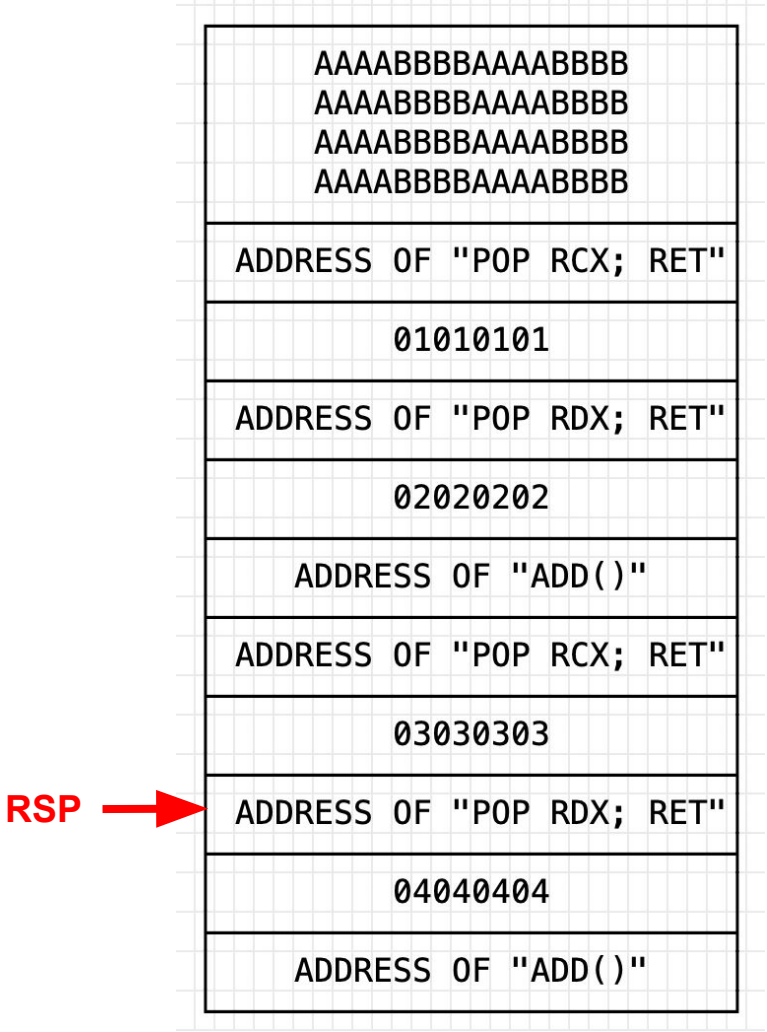

- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RCX
- Returns into Add()



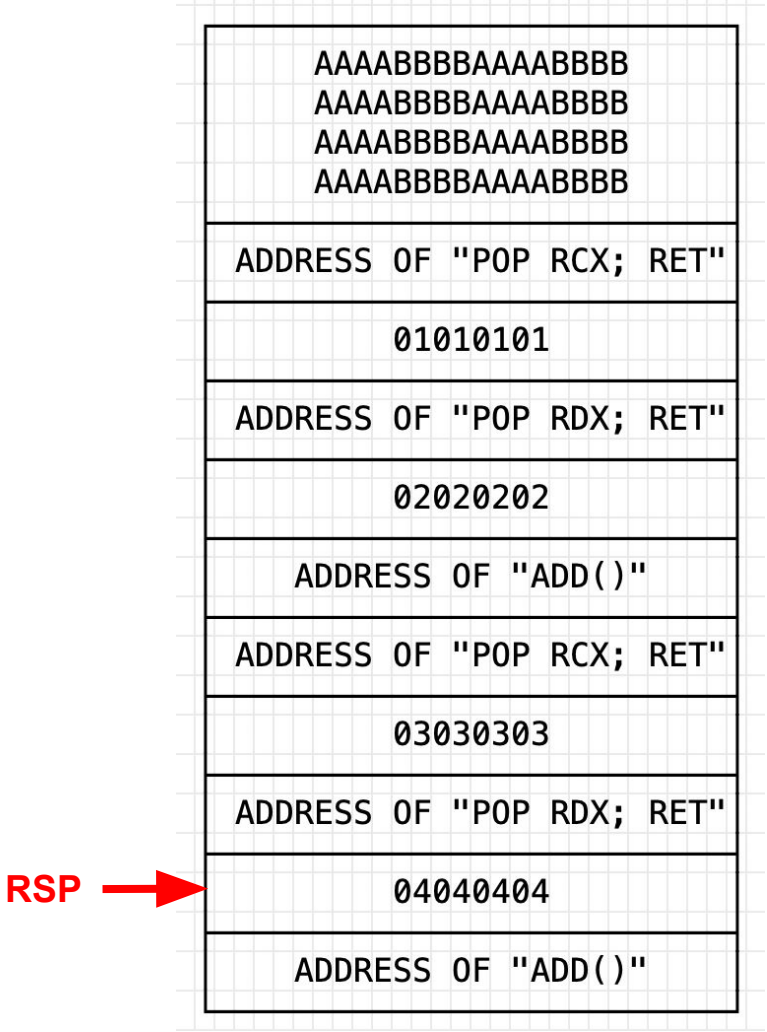

- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RCX
- Returns into Add()
- Returns into third gadget



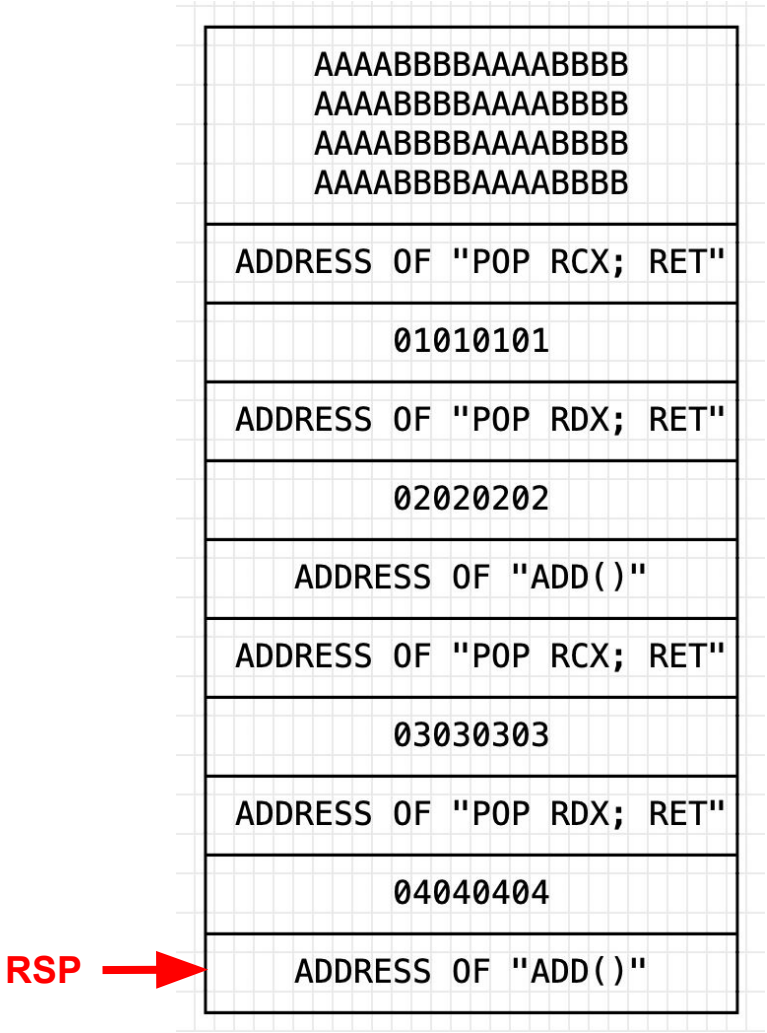

- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RCX
- Returns into Add()
- Returns into third gadget
  - POP top of stack (03030303) INTO RCX




- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RCX
- Returns into Add()
- Returns into third gadget
  - POP top of stack (03030303) INTO RCX
- Returns into fourth gadget




- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RDX
- Returns into Add()
- Returns into third gadget
  - POP top of stack (03030303) INTO RCX
- Returns into fourth gadget
  - POP top of stack (04040404) INTO RDX




- Return from overflow function
  - POP top of stack (01010101) INTO RCX
- Returns from first gadget into second gadget
  - POP top of stack (02020202) INTO RDX
- Returns into Add()
- Returns into third gadget
  - POP top of stack (03030303) INTO RCX
- Returns into fourth gadget
  - POP top of stack (04040404) INTO RDX
- Returns into Add()

# This technique is commonly called

ret2**code** -> Calling functions defined in the program itself

ret2**libc** -> Calling functions defined in libc (system/fread/fopen/etc)

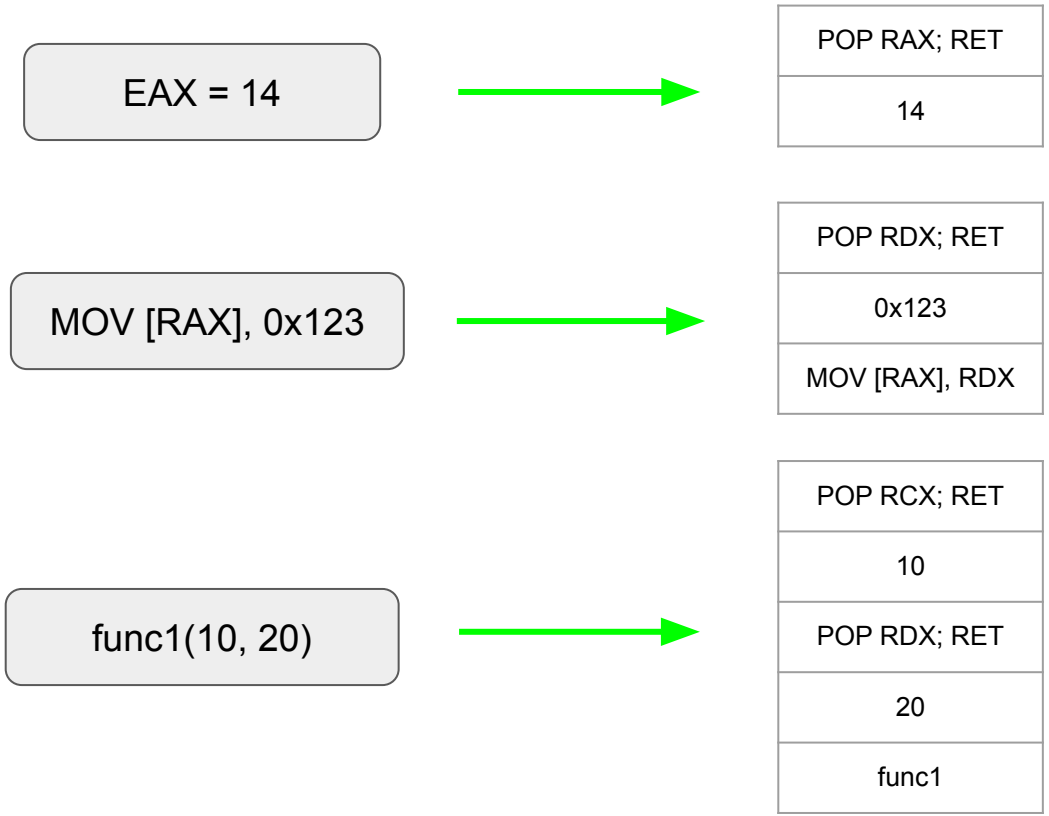
ret2**dl** -> Calling functions defined in libc via the dynamic linker

ret2**XXX** -> Calling functions defined in some part of the program

**ROP** -> Calling functions you create yourself with gadgets...


rop is like a
ransom note

# To be a ROP chain you must think like a ROP chain



# Steps to successfully ropping

1. Work out what you want to execute
2. Find gadgets that you can chain together
3. ???
4. PROFIT

# What does a typical rop chain look like?

you leak the libc addr for **system()**:

- 0x7ffff7c58740

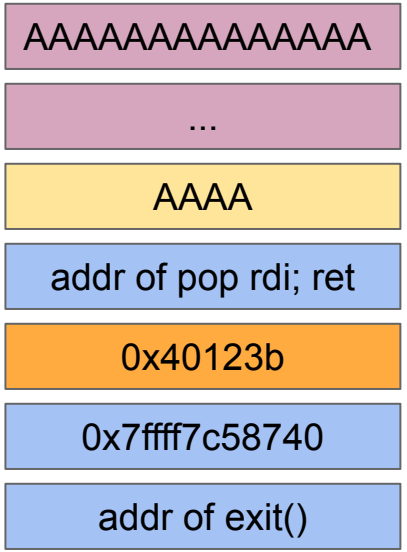
you find the address for the string **“/bin/sh”**

- 0x40123b

provide **system()** RET address

or **exit()** for clean exit (**Optional**)

This is called **ret2libc**



# More on ROP

- Ret2libc/Ret2code require you to have either
  - Binaries with useful functions
    - or
  - libc linked in (and leakable)
- This isn’t always true
  - In IOT devices, programs are usually small, and do not include any standard libraries
- If we can’t call functions, we can call **syscalls**
- What we require
  - Same as before +
  - A **syscall** gadget

# How do we find gadgets?

Pwntools

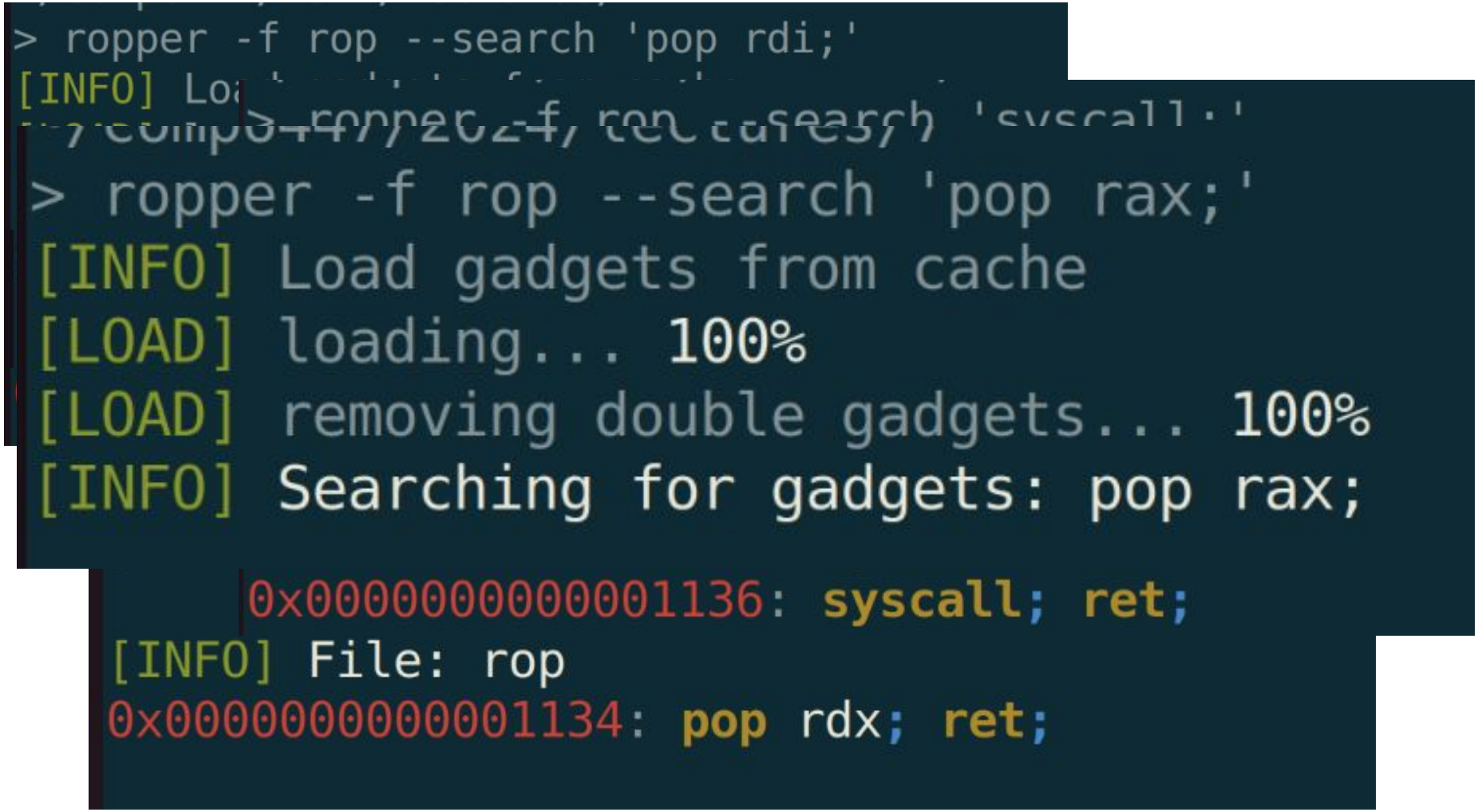
```
code = ELF("./ropme")
gadget = lambda x: next(code.search(asm(x, os='linux', arch=code.arch)))
```

```
p.elf.search('/bin/sh').next()
```

Ropper

```
static honeypot% ropper -f static --search 'pop eax; ret'
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop eax; ret

[INFO] File: static
0x080a8cb6: pop eax; ret;
```


> ropper -f rop --search 'pop rdi;'
> ropper -f rop --search 'syscall;'
> ropper -f rop --search 'pop rax;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rax;
0x0000000000001136: syscall; ret;
[INFO] File: rop
0x0000000000001134: pop rdx; ret;

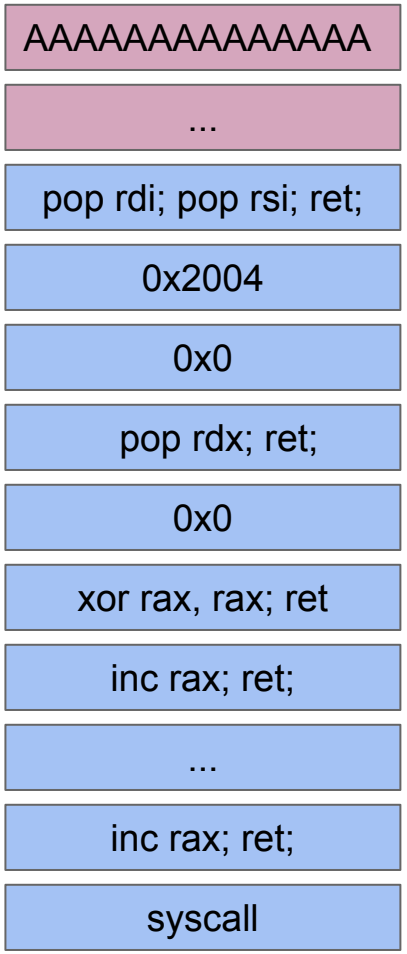
# So… do we give up?

```
> ropper -f rop --search '??? rax'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: ??? rax

[INFO] File: rop
0x000000000000107e: cmp rax, rdi; je 0x1098; mov rax, qword ptr [rip + 0x2f56]; test rax, rax; je 0x1098; jmp rax;
0x000000000000113d: inc rax; ret;
0x000000000000108f: jmp rax;
0x00000000000010c4: mov rax, qword ptr [rip + 0x2f25]; test rax, rax; je 0x10d8; jmp rax;
0x0000000000001083: mov rax, qword ptr [rip + 0x2f56]; test rax, rax; je 0x1098; jmp rax;
0x0000000000001008: mov rax, qword ptr [rip + 0x2fd9]; test rax, rax; je 0x1016; call rax;
0x0000000000001139: xor rax, rax; ret;
```

# Construct your chain

chain =



# That was fun...

Some tools like ropper & pwntools have functionality to **automagically** generate these rop chains.

**You are not allowed to use ROP chain generators in this course (incl exams).**

**They are not good. They are obvious when marking.**

# What about bad bytes

In shellcode sometimes we had to avoid NULL bytes.

Similarly in ROP chains, what are some bad bytes?

- 0x0a?
- 0x0b?
- 0x00?

Must make sure we don’t use addresses that have these bytes

# What about Stack Alignment

The x86-64 Calling Convention on linux **requires** the stack be 16-byte aligned. Some library functions will **crash** if this assumption is false.

When doing ROP, keep this in mind. If your stack is not aligned (Segfault on some random system function), try adding an extra `ret` gadget.

# Some more on ret2libc

If there are not enough **gadgets** in your **binary**, we know that LIBC MUST be running somewhere on the target computer

**We can use gadgets found in LIBC just as we would from the target binary**

First we need 2 pieces of information

**Where is LIBC?**

**What version is LIBC?**

**WHERE is LIBC?**

> ASLR is enabled...

**GOT Table -> Leak LIBC Address!**

How can I leak an address?

# One example

puts@plt(puts@got)

We have learnt how to call **any** function with **any** argument. If you call puts(), with the argument being the GOT entry for puts…

It will just print out the address of puts?

be creative. There’s many ways to do this

# What version is the libc?

Tools exist that take offsets/addresses of functions like printf/gets/etc

And they will give you a candidate version for libc..

libc.rip is an example

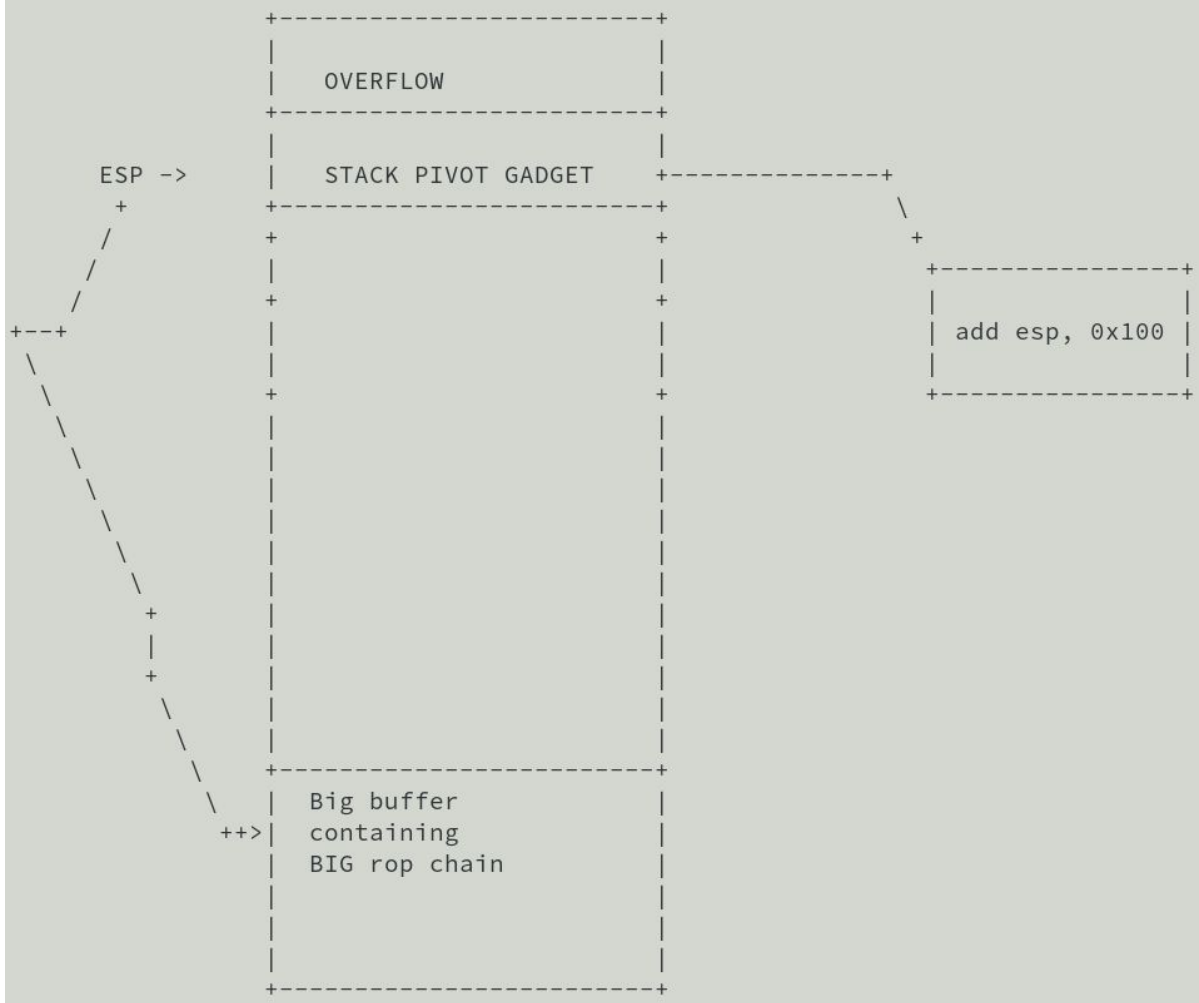
# Stack pivots

Sometimes ROP chains can get very large..

Sometimes your buffer might not be big enough to fit your entire ROP chain

A pivot is simply moving the stack pointer somewhere else..

1. Use a gadget like **sub rsp, 0x80**
2. Use a partial/complete overwrite of **RBP** on the stack


OVERFLOW
STACK PIVOT GADGET
ESP ->
add esp, 0x100
Big buffer
containing
BIG rop chain

# Sometimes your stack pivot might not be exact.

Can we use a NOPSLED???

No but we can use a **RETSLED.**

ret->ret->ret->ret->ret….

Just needs a single ret gadget.

# Automation

- ROP can be hard
- Sometimes you just can’t find the right gadgets
  - If you want to set RDI=4
    - You chain might look like
      - RAX = 1
      - RBX = RAX * 4
      - XOR RDI, RDI
      - ADD RDI, RBX
  - Finding good gadgets is hard
    - Especially in small programs

# Automation

- At the core of **automating** ROP chains is
  - **Symbolic Execution**
    - is a means of analyzing a program to determine what inputs cause each part of a program to execute
    - Used to **understand effect of gadgets**
  - **Constraint satisfaction problems**
    - are mathematical questions defined as a set of **objects** whose **state** must **satisfy a number of constraints or limitations**
    - Used to **generate chains**
  - **SAT solvers**
    - is something you give a boolean formula to, and it tells you whether it can find a value for the different variables such that the formula is true.
    - Used to solve above problems

ie: a lot of maths

```
0x403be4:    and    ebp,edi
0x403be6:    mov    QWORD PTR [rbx+0x90],rax
0x403bed:    xor    eax,eax
0x403bef:    add    rsp,0x10
0x403bf3:    pop    rbx
0x403bf4:    ret
```

- What is **angrop**?
  - **angrop** is a rop gadget finder and chain builder
  - It is built on top of angr's symbolic execution engine, and uses constraint solving for generating chains and understanding the effec
- Stores gadgets
  - Dependencies
  - Side effects

```
>>> print(rop.gadgets[0])
Gadget 0x403be4
Stack change: 0x20
Changed registers: set(['rbx', 'rax', 'rbp'])
Popped registers: set(['rbx'])
Register dependencies:
    rbp: [rdi, rbp]
Memory write:
    address (64 bits) depends on: ['rbx']
    data (64 bits) depends on: ['rax']
```

```
# angrop includes methods to create certain common chains

# setting registers
chain = rop.set_regs(rax=0x1337, rbx=0x56565656)

# writing to memory
# writes "/bin/sh\0" to address 0x61b100
chain = rop.write_to_mem(0x61b100, b"/bin/sh\0")

# calling functions
chain = rop.func_call("read", [0, 0x804f000, 0x100])

# adding values to memory
chain = rop.add_to_mem(0x804f124, 0x41414141)

# chains can be added together to chain operations
chain = rop.write_to_mem(0x61b100, b"/home/ctf/flag\x00") + rop.func_call("open", [0x61b100,os.O_RDONLY]) +

# chains can be printed for copy pasting into exploits
>>> chain.print_payload_code()
chain = b""
chain += p64(0x410b23)  # pop rax; ret
chain += p64(0x74632f656d6f682f)
chain += p64(0x404dc0)  # pop rbx; ret
chain += p64(0x61b0f8)
chain += p64(0x40ab63)  # mov qword ptr [rbx + 8], rax; add rsp, 0x10; pop rbx; ret
...
```

- These tools are cool
  - Take advantage of them in **CTFS or real world analysis**
  - **Don’t use them in this course**

# Securities

> 2002 Tyler Durden “Bypassing PAX ASLR protection”

> 2005 Sebastian Krahmer “Borrowed code chunks technique”

< ARMv8.3 - We’ll use crypto to sign pointers so you can’t call Gadgets!

> 2019 Adam T ”Adam started lecturing COMP6447”

> 2019 Google “Examining Pointer Authentication on the iPhone XS”

> 2022 Lachlan W ”Lachlan starts tutoring in COMP6447”

> 2022 MIT Researchers “PACMAN”

< ARMv8.5 - We’ll add new instructions to identify branch targets!

> 20XX COMP6447 Graduate “TBD”

# Wargame hint

This weeks reversing challenge is about reversing a **struct. You must submit the struct type as well as the code...**

**Assignment is partly due this week ;)**