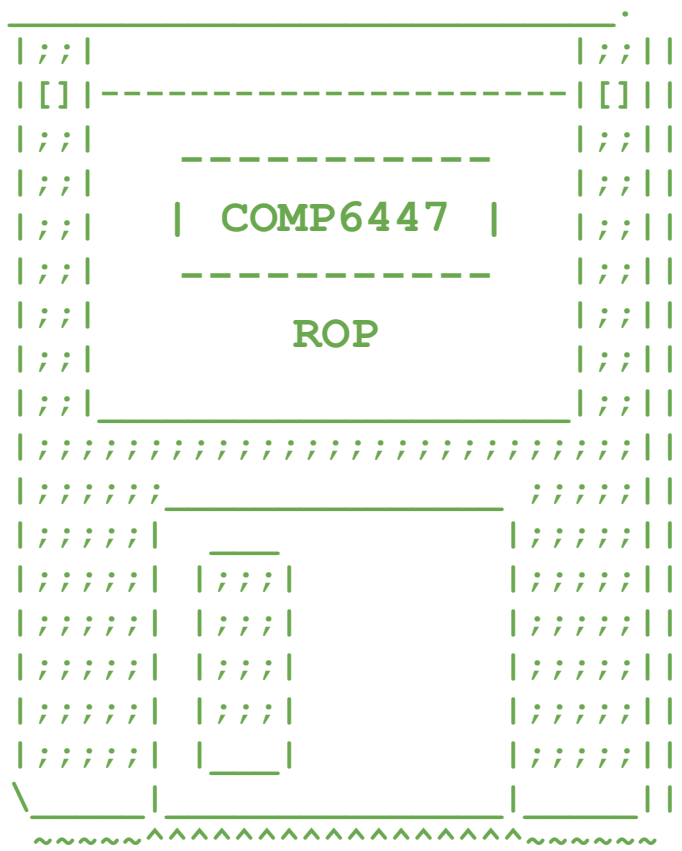

COMP6447
ROP

Assignment & Exam

- Good job on the exam (marks are out on webcms)
  - If you think you need help talk to your tutor sooner rather than later
- Good luck on assignment (talk to tutors if you need help)
  - Challenges were reuploaded late last week to fix a bug
  - Any questions / faq email cs6447@cse.unsw.edu.au or ping me on slack (spod@)

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

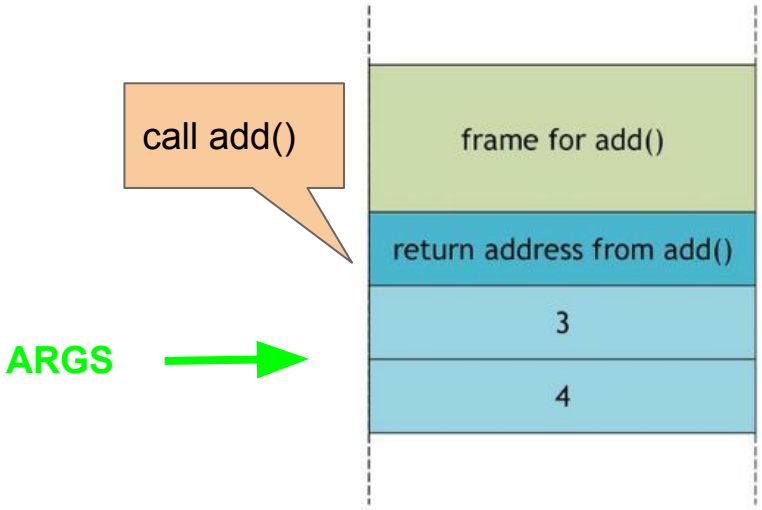
What is rop

- 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?

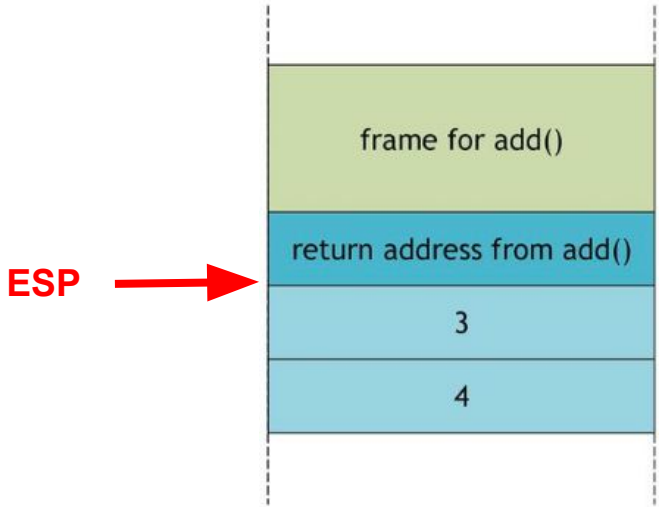
- Defeats NX / Code Signing protections

How do we call a function?




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

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

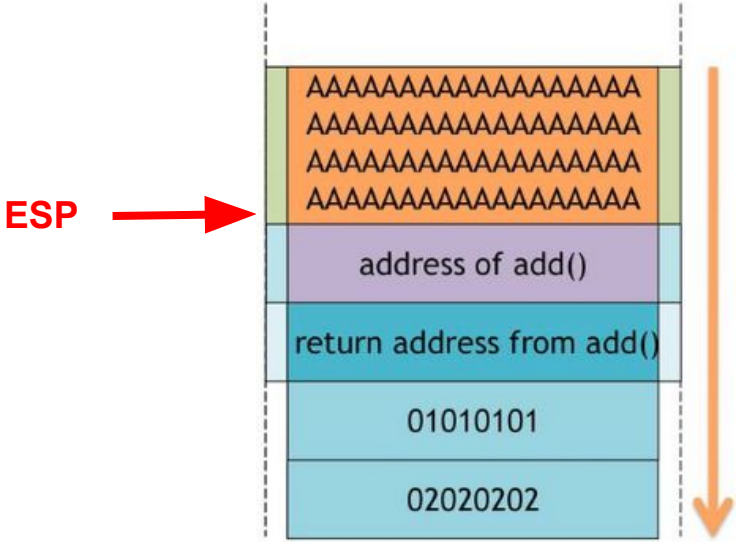
Finished executing. Now ESP is here



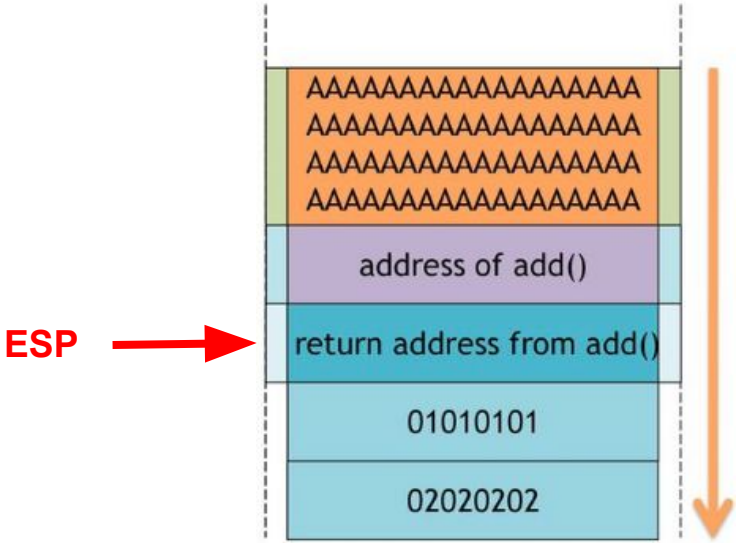

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}

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

Now with a buffer overflow

call add(01010101,02020202)



before add returns



What if I wanted to call the same function twice

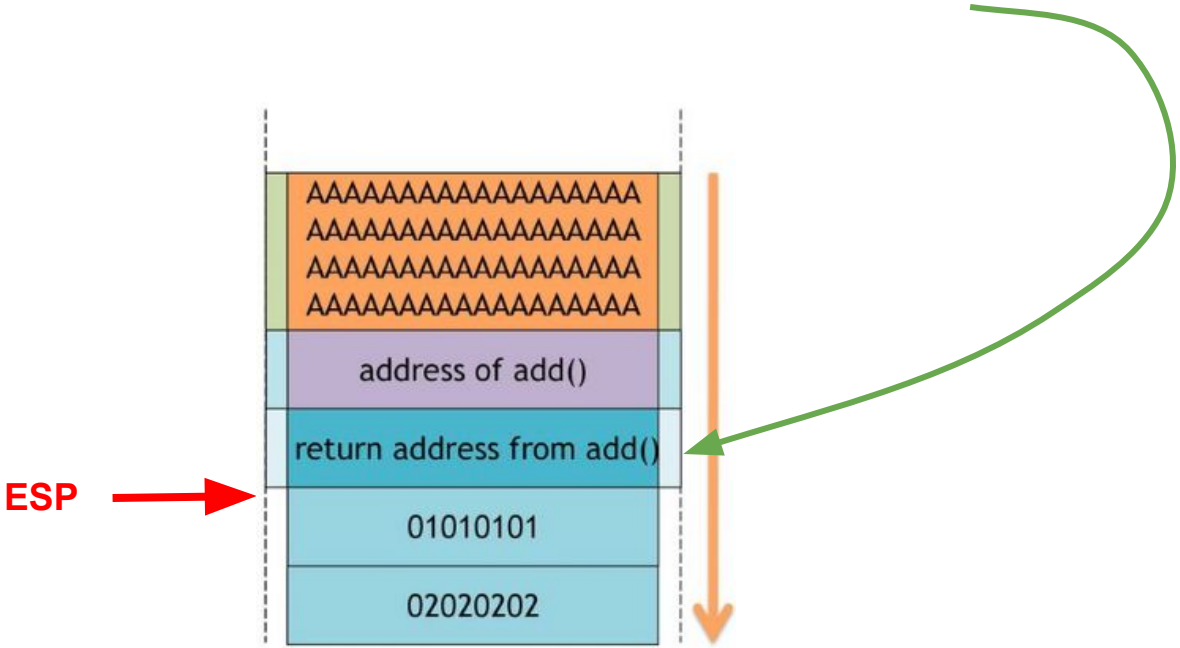
With different arguments

add(01010101, 02020202)

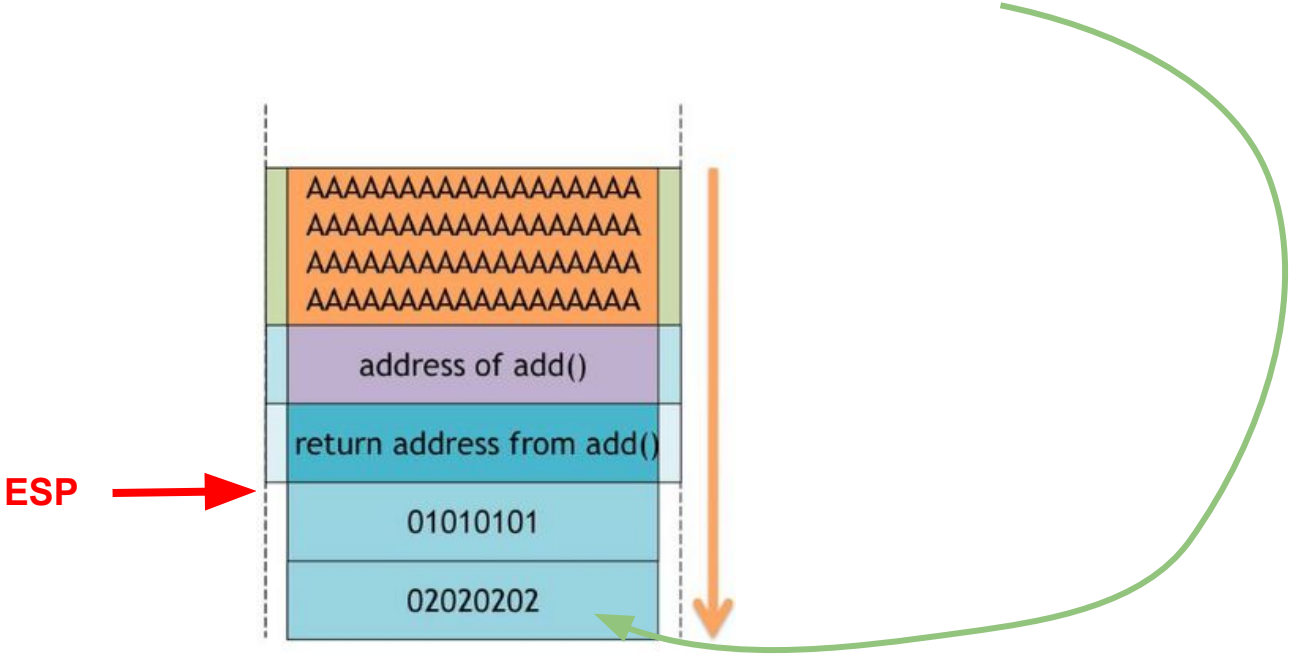
add(03030303, 04040404)

How would this look?

Can I just put another add here?



Where would the new arguments go?



So how do we do it

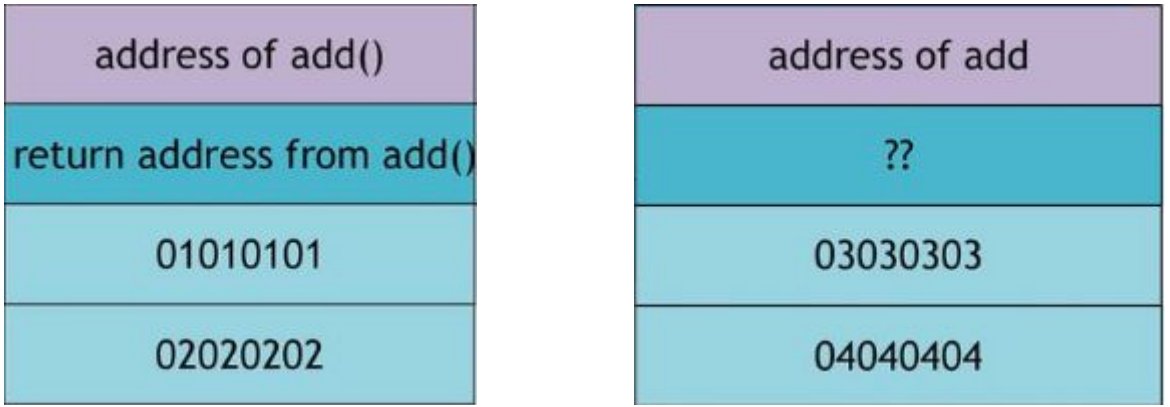
How do we call

add(01010101, 02020202)

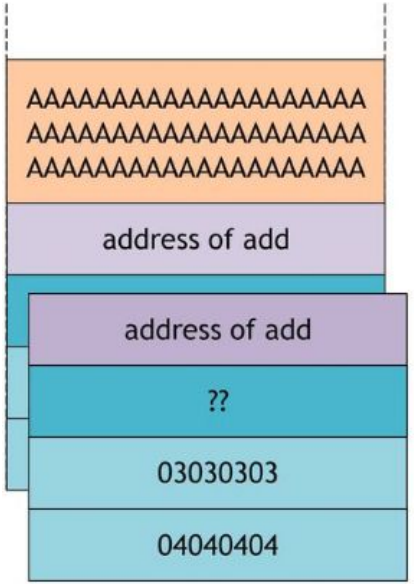
then

add(03030303, 04040404)

We have 2 stack frames



We need to find a place to put the second stack frame



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

```
0x00000858: push cs; adc al, 0x41; ret;
0x00000855: push cs; and byte ptr [edi + 0xe], al; adc al, 0x41; ret;
0x00000852: push cs; xor byte ptr [ebp + 0xe], cl; and byte ptr [edi + 0xe], al; adc al, 0x41; ret;
0x000004b6: push dword ptr [ebx + 0x30]; call 0x460; hlt; mov ebx, dword ptr [esp]; ret;
0x0000050b: push ecx; call eax; add esp, 0x10; leave; ret;
0x0000055c: push ecx; call edx; add esp, 0x10; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x000004b5: push esi; push dword ptr [ebx + 0x30]; call 0x460; hlt; mov ebx, dword ptr [esp]; ret;
0x000004c0: push esp; mov ebx, dword ptr [esp]; ret;
0x00000526: ret 0x1aa3;
0x000004e6: ret 0x1ae3;
0x0000055e: rol byte ptr [ebx + 0x5d8b10c4], cl; cld; leave; ret;
0x000004c3: sbb al, 0x24; ret;
0x00000850: sub al, 0x44; push cs; xor byte ptr [ebp + 0xe], cl; and byte ptr [edi + 0xe], al; adc al, 0x41; ret;
0x00000508: sub esp, 0x14; push ecx; call eax; add esp, 0x10; leave; ret;
0x00000810: test dword ptr [edx], eax; inc edx; or eax, 0x3834105; add al, byte ptr [edx - 0x3b]; ret;
0x000007ec: test dword ptr [edx], eax; inc edx; or eax, 0x3834705; add cl, byte ptr [eax - 0x3b]; ret;
0x00000411: test eax, eax; je 0x41a; call 0x480; add esp, 8; pop ebx; ret;
0x000004b8: xor byte ptr [eax], al; add byte ptr [eax], al; call 0x460; hlt; mov ebx, dword ptr [esp]; ret;
0x00000853: xor byte ptr [ebp + 0xe], cl; and byte ptr [edi + 0xe], al; adc al, 0x41; ret;
0x000005a7: xor eax, 0xc6ffffff; add dword ptr [eax], 0; add byte ptr [ecx], al; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x00000564: cld; leave; ret;
0x000004c1: hlt; mov ebx, dword ptr [esp]; ret;
0x00000511: leave; ret;
0x000004cf: nop; mov ebx, dword ptr [esp]; ret;
0x000004cd: nop; nop; mov ebx, dword ptr [esp]; ret;
0x000004cb: nop; nop; nop; mov ebx, dword ptr [esp]; ret;
0x000004c9: nop; nop; nop; nop; mov ebx, dword ptr [esp]; ret;
0x00000406: 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.

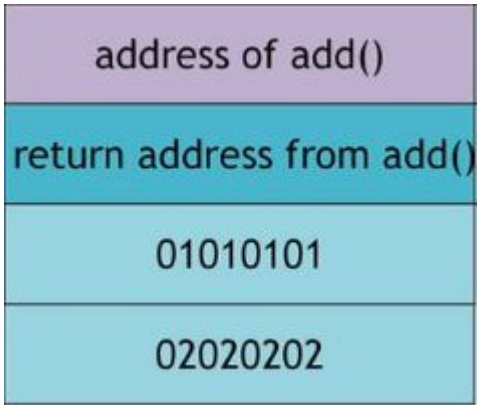
- mov eax, ebx
- int 0x80
- pop eax

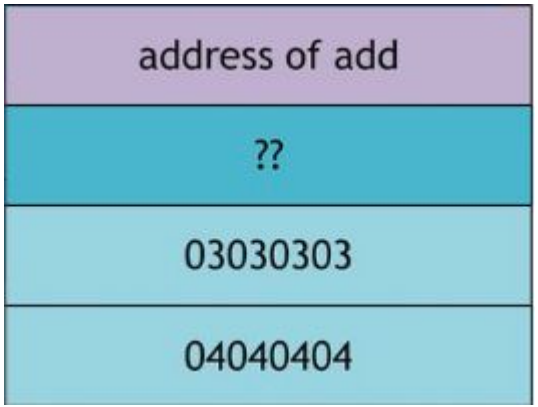
But if you chain tiny instructions together… you are pretty much writing shellcode

Back to our previous example

We have 2 stack frames to execute..

We need to somehow clean up the old arguments before calling the second add function...

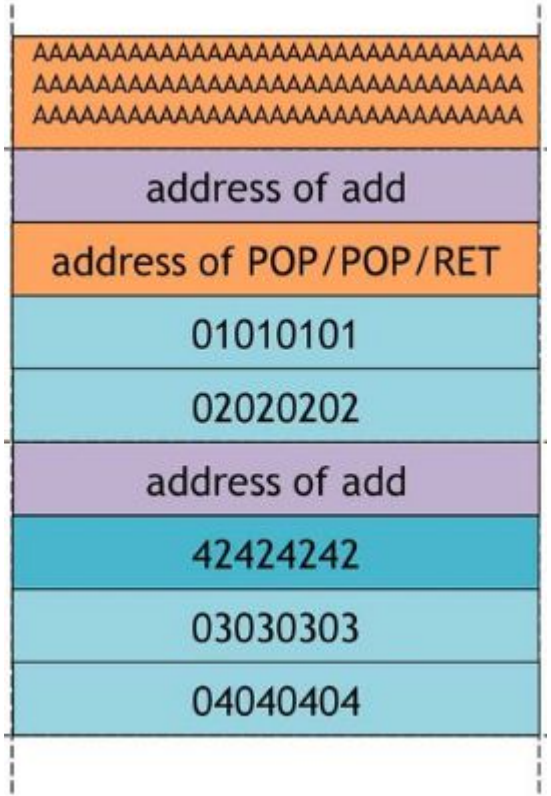




pop

pop

ret

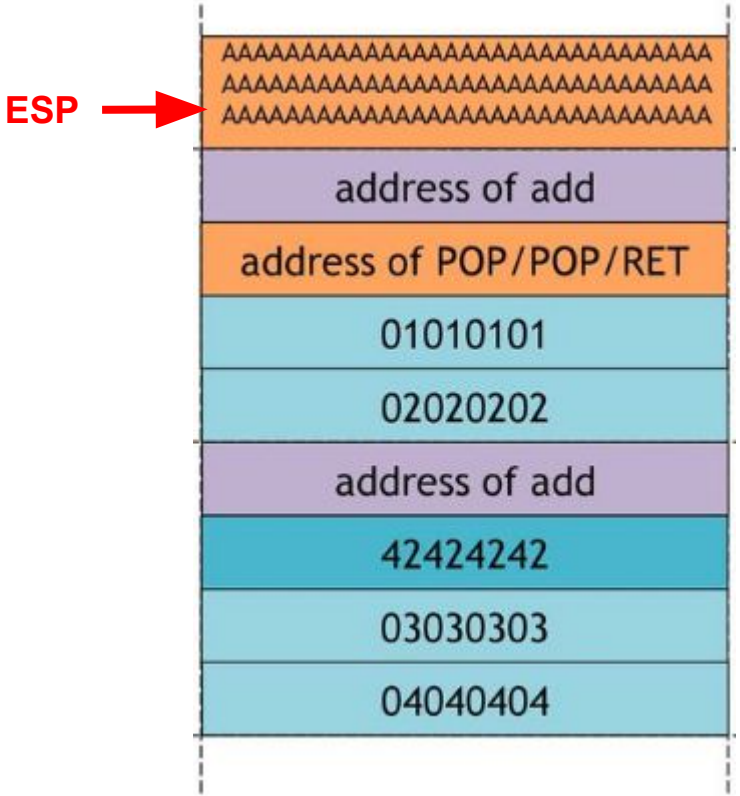


Using our found gadget:

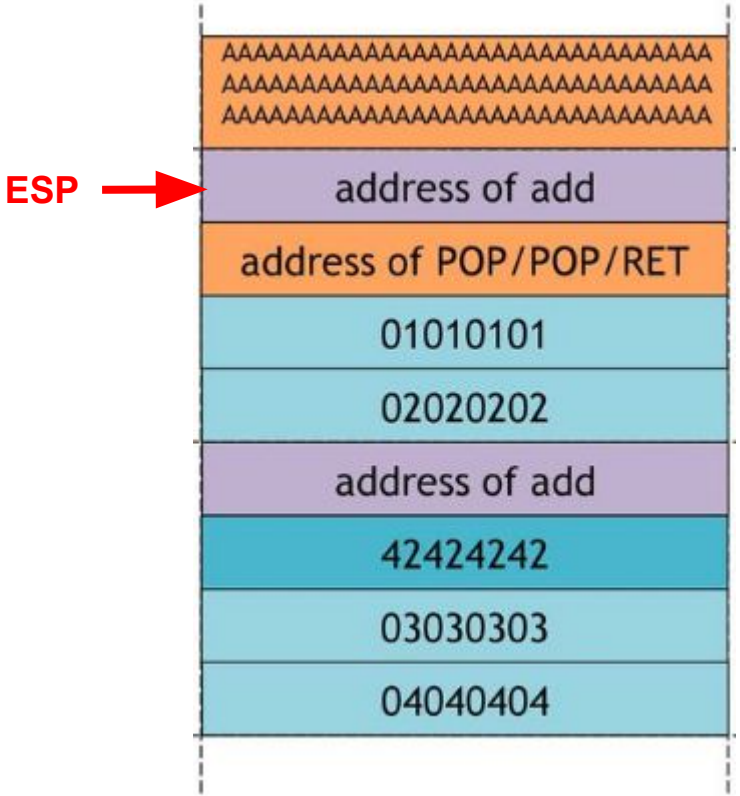
We can **POP, POP, RET,** our way over the first functions arguments

Then execute our second stack frame!

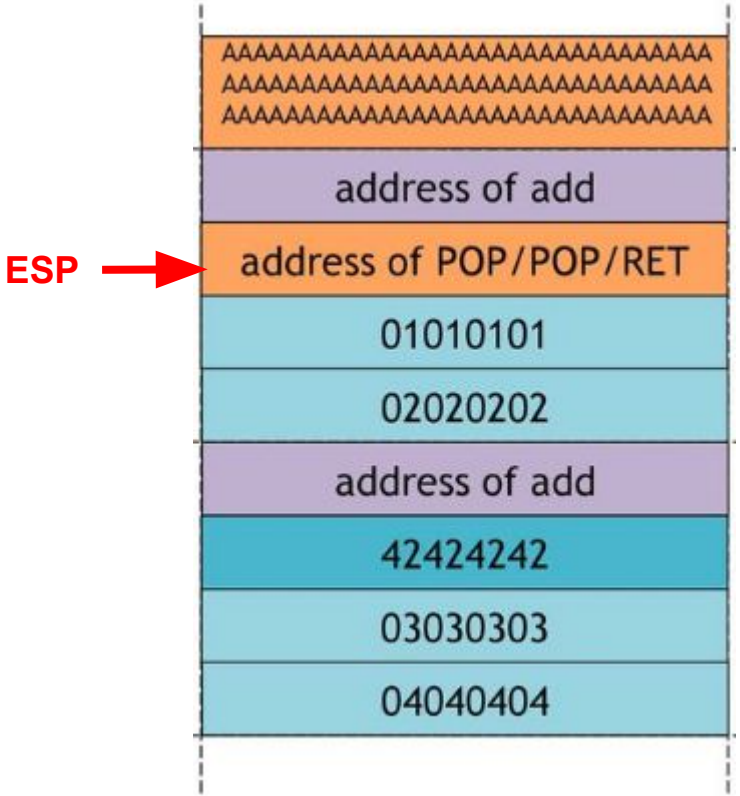
Lets see how this works



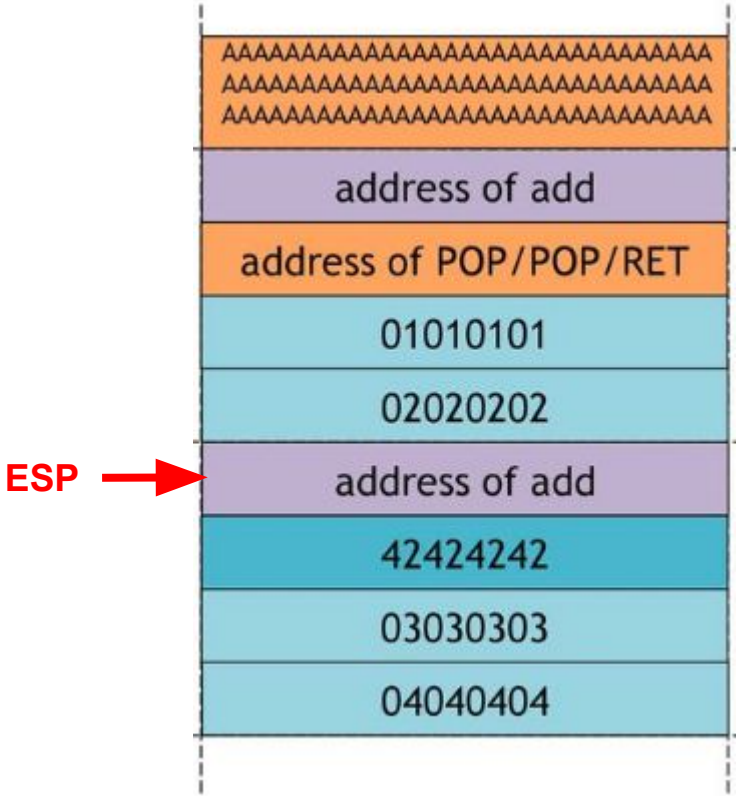
- Return from overflow function



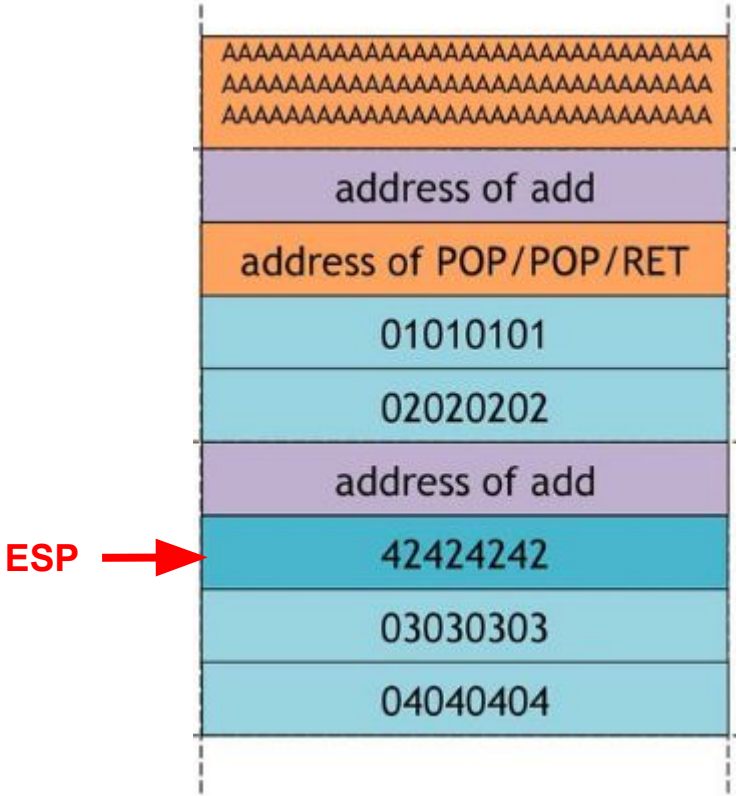
- Return from overflow function
- Return to add()



- Return from overflow function
- Return to add()
- Return to poppopret gadget



- Return from overflow function
- Return to add()
- Return to poppopret gadget
- POP
- POP
- RET to add



- Return from overflow function
- Return to add(01010101...)
- Return to poppopret gadget
- POP
- POP
- RET to add(03030303,...)
- Return to 0x42424242

This technique is commonly called

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

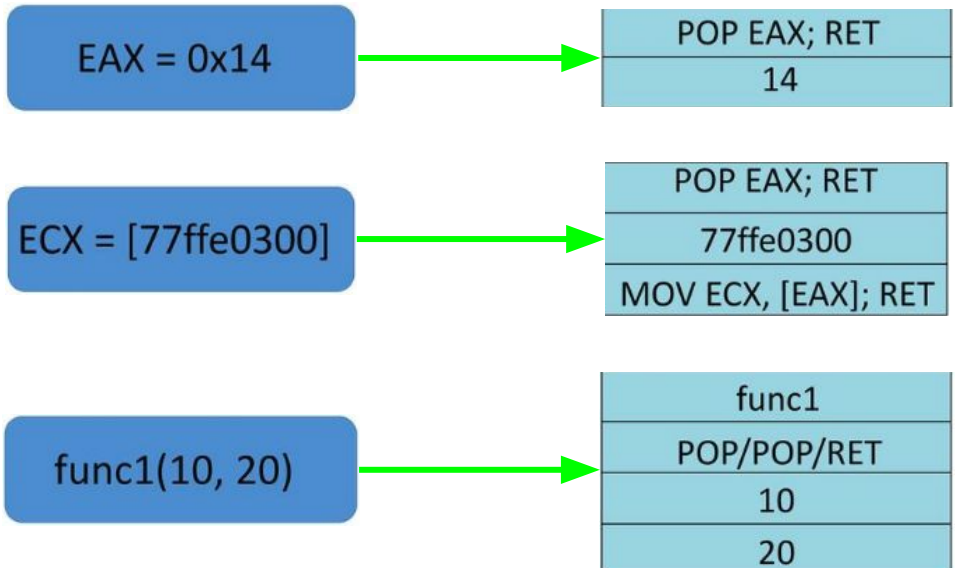
ret2**libc** -> Calling functions defined in libc (system/fread/fopen/etc)

ret2**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 succesfully 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()**:

- 0x7fcc6704

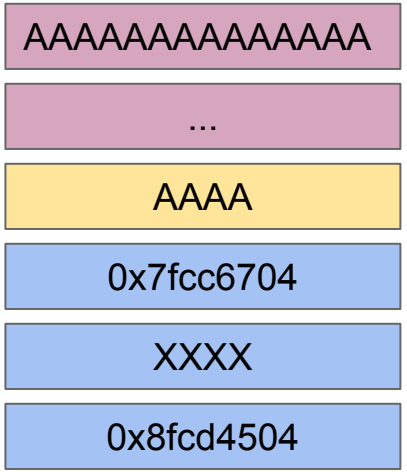
you find the address for the string **“/bin/sh”**

- 0x8fcd4504

provide **system()** RET address

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

This is **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;
```

```
tmp.wKN5ucSfYU honeypot% ropper -f a.out --search 'pop esi;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
```

```
tmp.wKN5ucSfYU honeypot% ropper -f a.out --search 'pop eax;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop eax;
```

```
tmp.wKN5ucSfYU honeypot% ropper -f a.out --search 'int 0x80;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: int 0x80;

[INFO] File: a.out
0x0804916c: int 0x80;
0x0804916c: int 0x80; ret;
```

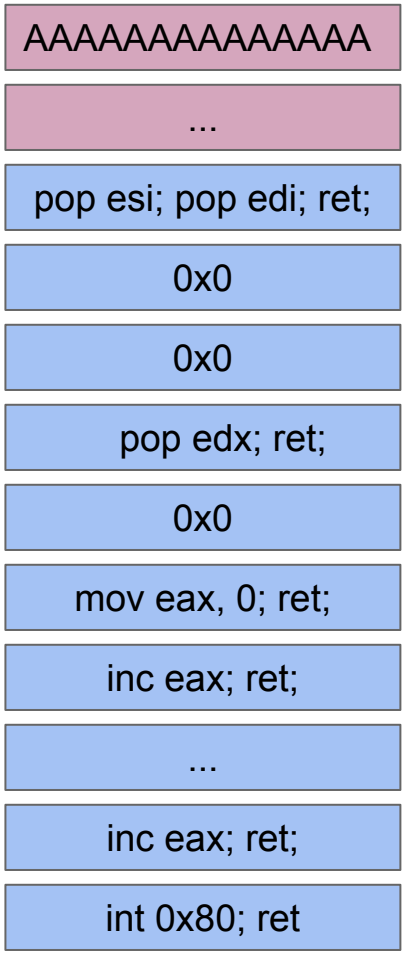
So… do we give uP?

no

```
tmp.wKN5ucSfYU honeypot% ropper -f a.out --search '??? eax'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: ??? eax

[INFO] File: a.out
0x08049149: add eax, 0x804c014; add ecx, ecx; ret;
0x08049177: inc eax; ret;
0x08049179: mov eax, 0; pop ebp; ret;
0x08049171: mov eax, 0; 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.

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(*puts)

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 canditate version for libc..

https://libc.nullbyte.cat/ is an example

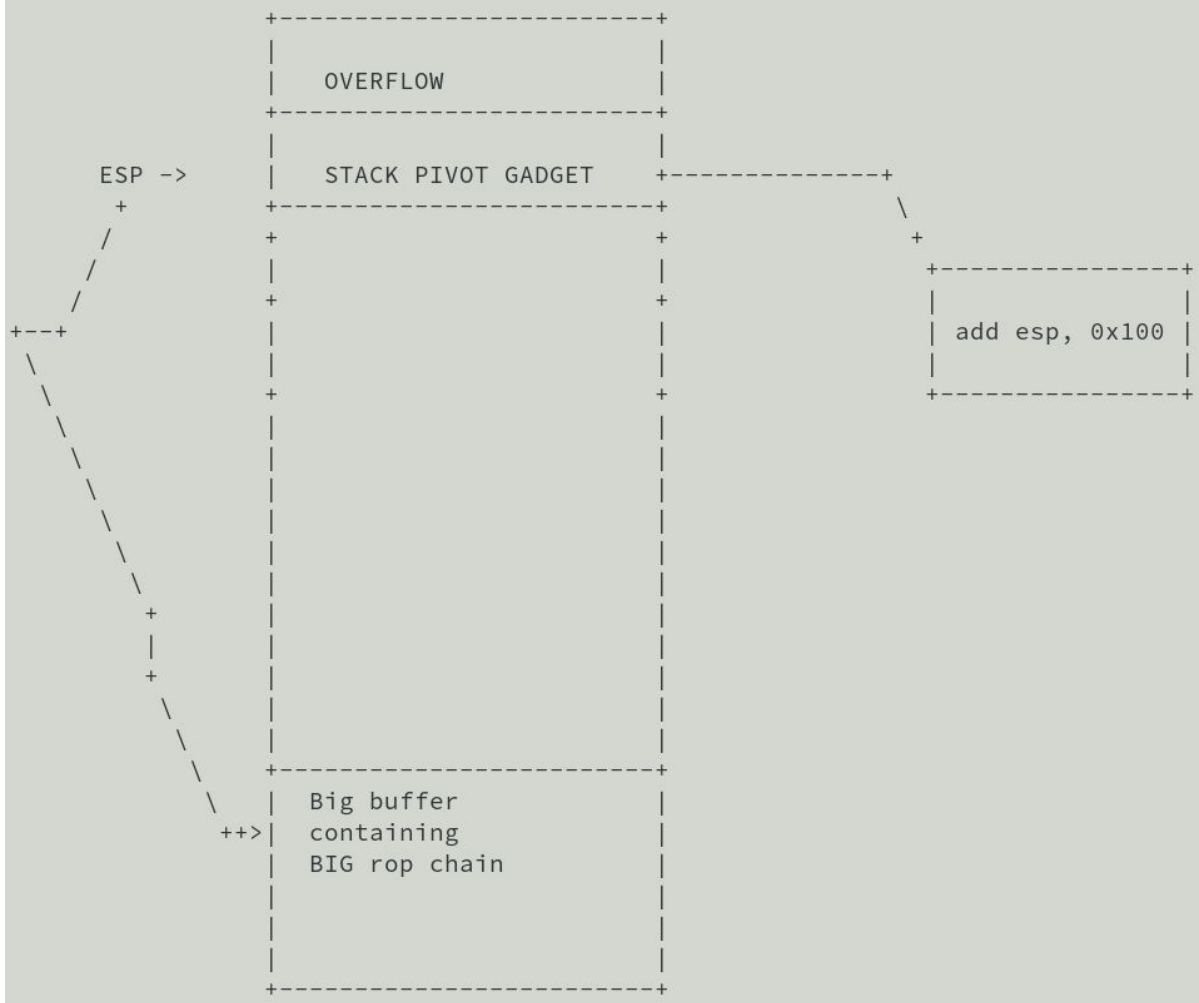
Stack pivots

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 esp, 0x80**
2. Use a partial/complete overwrite of **EBP** 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**

Wargame hint

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

Assignment stage 1 is due this week!

final exam format

3 sections

1. Binary exploitation
    a. Similar to midsem exam
2. Source code Auditing
    a. Similar to wargames
3. Reverse engineering
    a. Similar to wargames

Start preparing

- Biggest issue in midsem exam was timing.
    - To get faster at exploiting you need to exploit more
    - Should be able to go back and do most of wargames from previous weeks as practice