CSE467: Computer Security

13. Control Flow Hijack

Seongil Wi

Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

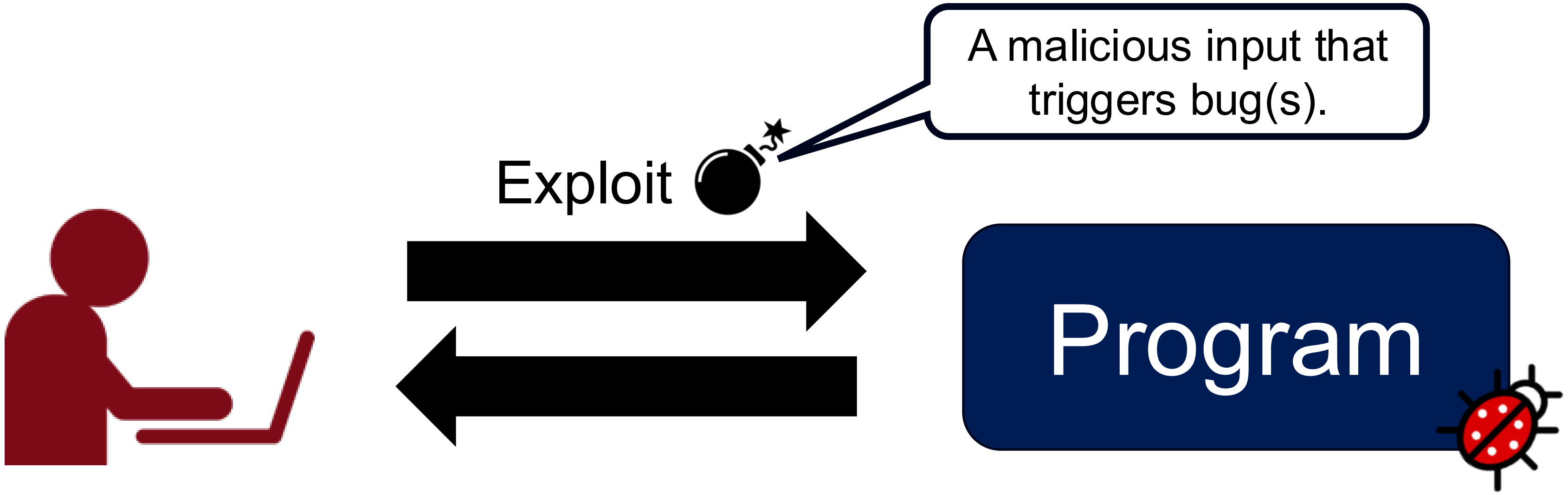
Software Bug

- Software bug is an ***error*** in a program

Q: If you only have time for fixing one bug out of hundred, which bug will you fix first?

Exploitable Bugs

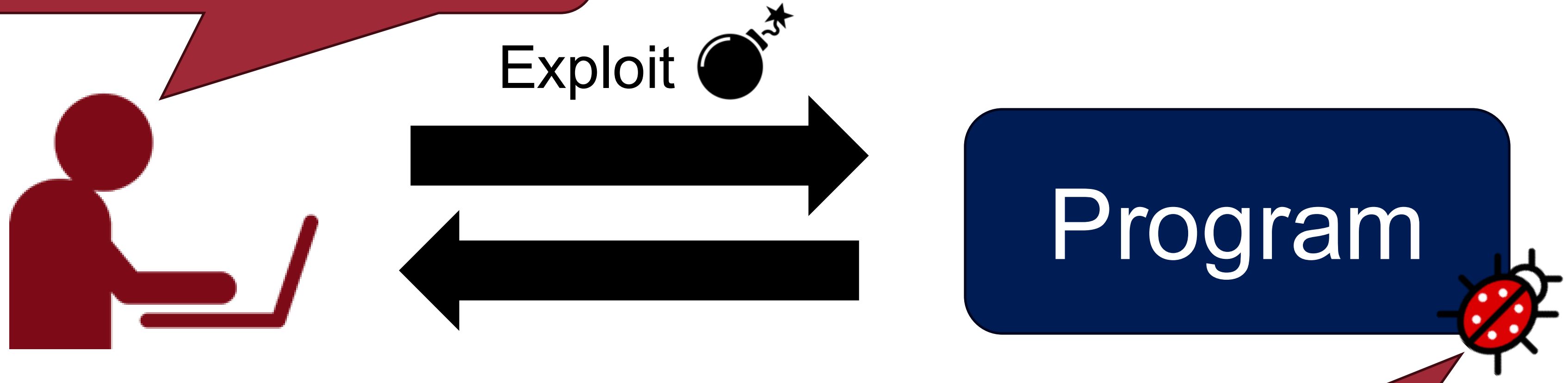
We often call an ***exploitable bug*** as a vulnerability



Exploitable Bugs

We often call an ***exploitable bug*** as a vulnerability

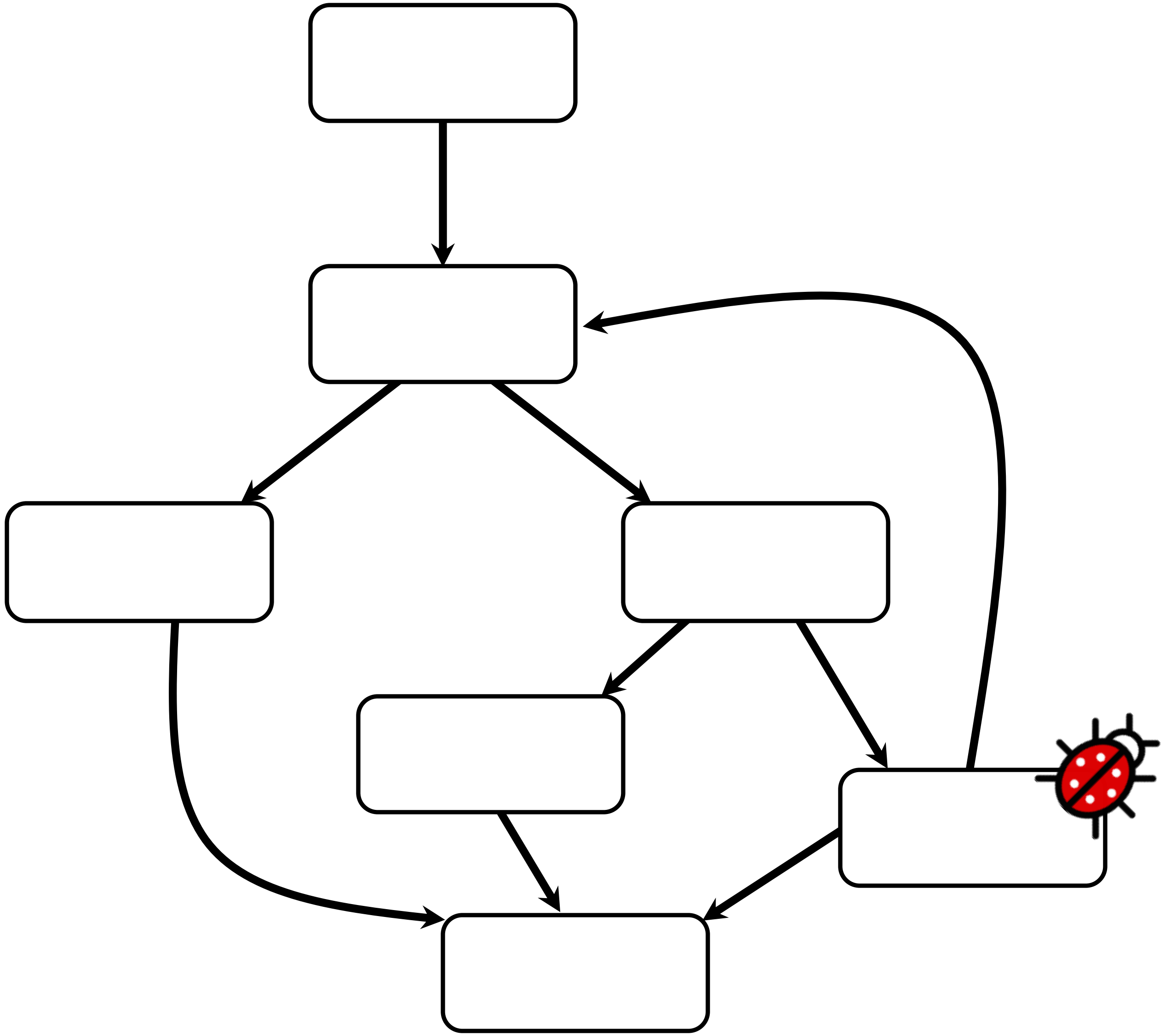
Exploitation is an act of taking advantage of a bug to cause unintended behavior of the target program



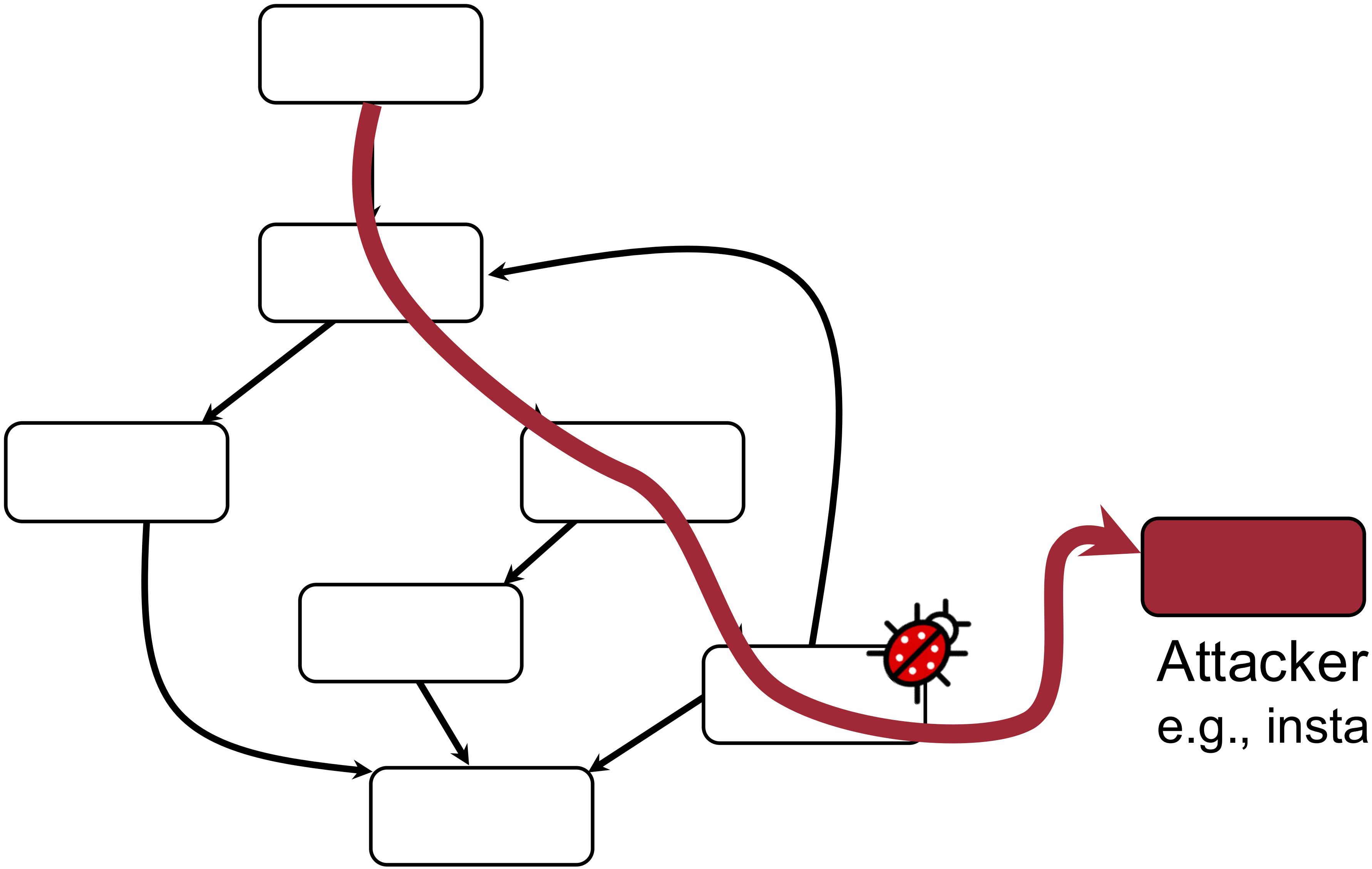
Some vulnerabilities allow an attacker to run any ***arbitrary code*** on victim's machines without their consent

Control Flow Hijack

Control Flow Hijack



Control Flow Hijack



Attacker’s own code/logic

e.g., install malicious software

The Classic Exploitation

- The first computer worm (called Morris Worm) was born

Robert Tappan Morris

- Creator of the worm
- Cornell graduate
- Professor at MIT now

Image from https://www.nndb.com/people/466/000027385/

Morris Worm

- Exploited a ***buffer overflow*** vulnerability

```
int main(int argc, char* argv[]) {
  char line[512];
  /* omitted ... */
  gets(line); /* Buffer Overflow! */
  /* omitted ... */
}
```

This simple line allowed the Morris Worm to infect 10% of the internet computers in 1988

Replicating Historic Exploitation

```
int main(int argc, char* argv[]) {
  char line[512];
  gets(line);
  return 0;
}
```

Compile this program with:

```
$ gcc -mpreferred-stack-boundary=2
-O0 -fno-stack-protector -fno-pic
-no-pie -z execstack -o morris
morris.c –m32
```

Compiler Warning (ignore this for now):
morris.c:(.text+0x11): warning: the `gets' function is dangerous and should not be used.

gets(char *s)

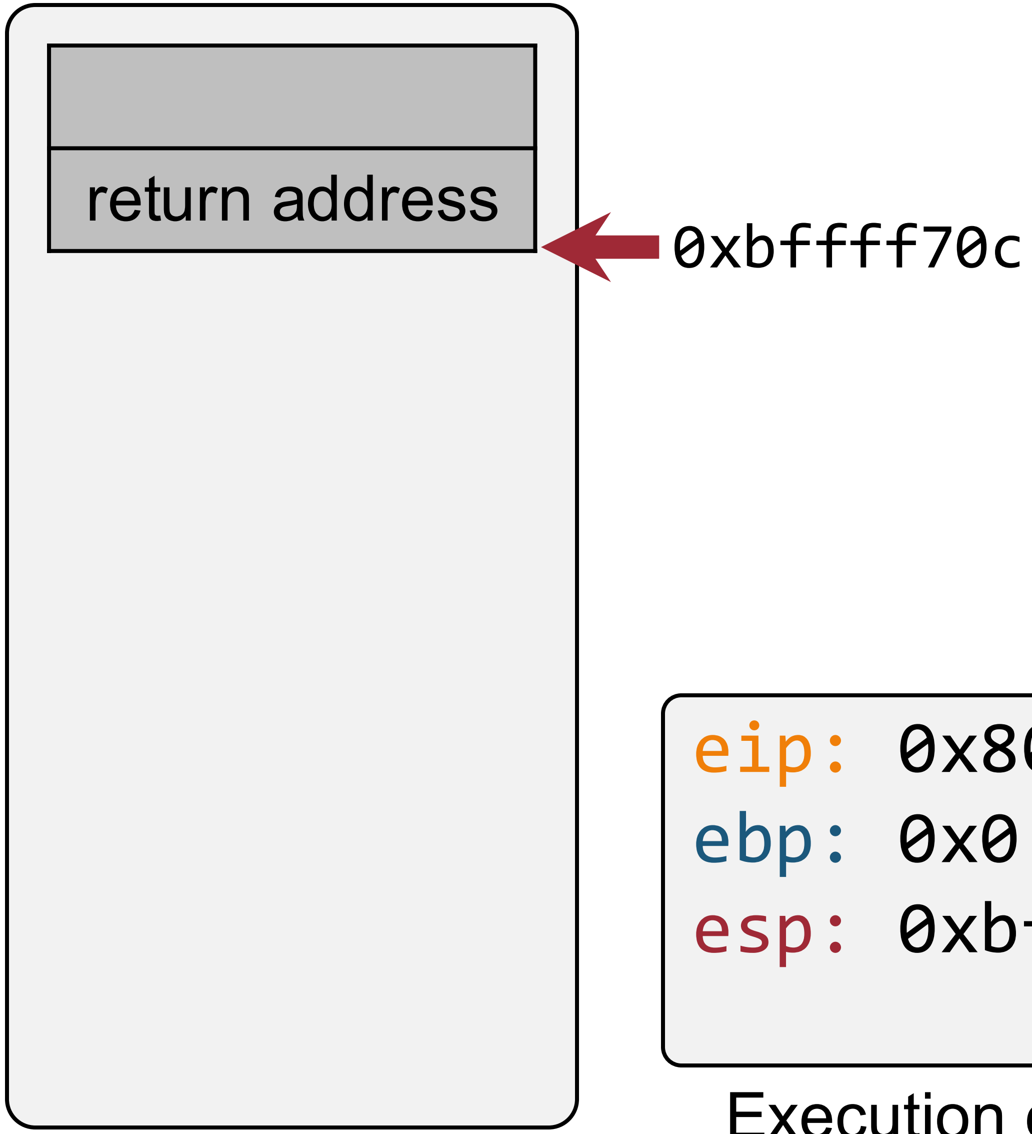
Reads a line from STDIN into the buffer pointed to by s until a terminating new line or EOF, which it replaces with a NULL byte (‘\0’)

Disassembled Code for the Morris Worm

```
$ objdump -M intel -d morris

08049162 <main>:
 8049162:  55                    push   ebp
 8049163:  89 e5                 mov    ebp,esp
 8049165:  81 ec 00 02 00 00     sub    esp,0x200
 804916b:  8d 85 00 fe ff ff     lea    eax,[ebp-0x200]
 8049171:  50                    push   eax
 8049172:  e8 b9 fe ff ff        call   8049030 <gets@plt>
 8049177:  83 c4 04              add    esp,0x4
 804917a:  b8 00 00 00 00        mov    eax,0x0
 804917f:  c9                    leave
 8049180:  c3                    ret
```

Analyzing the Vulnerability



return address

0xbffff70c

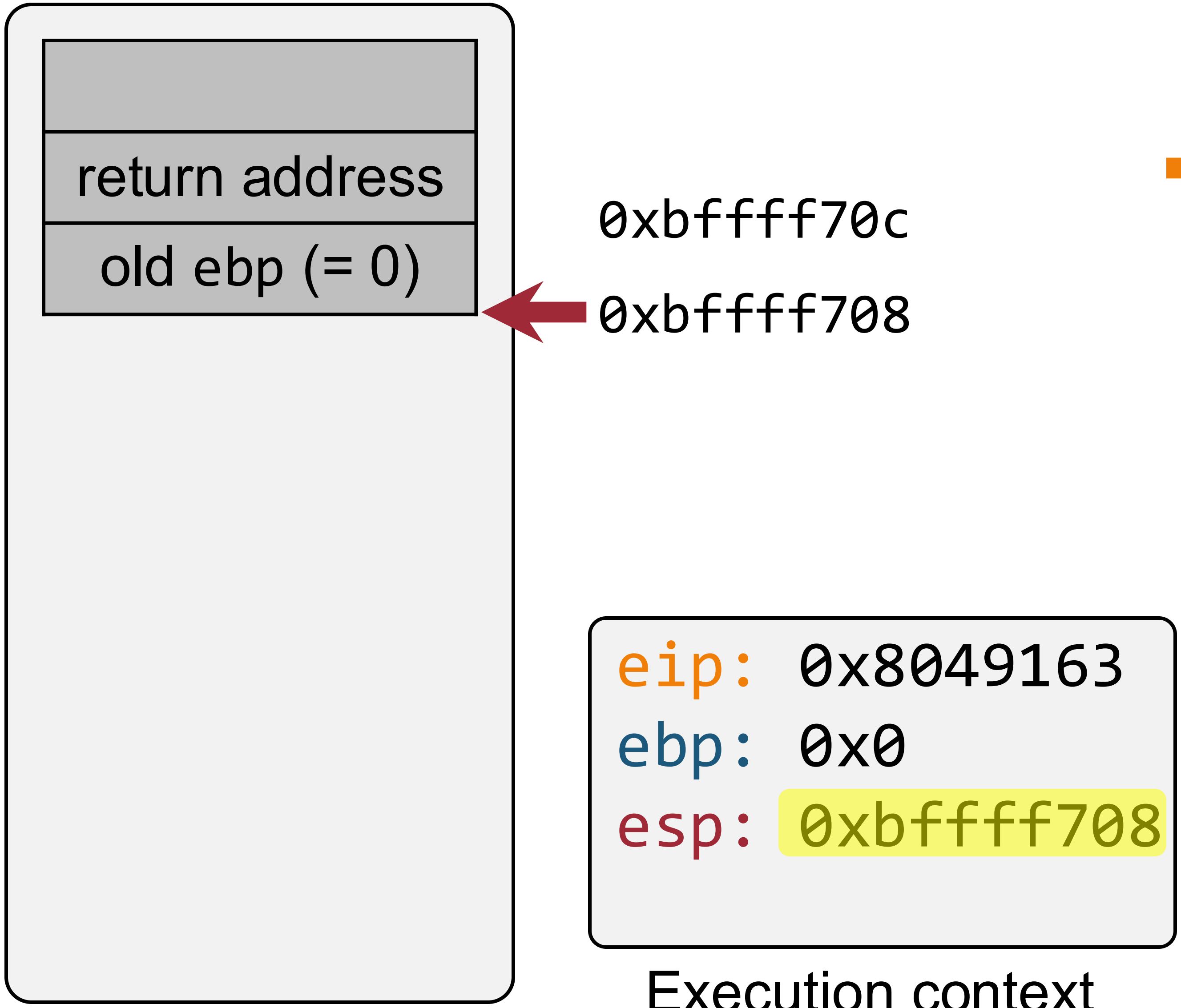
Virtual memory

```
eip: 0x8049162
ebp: 0x0
esp: 0xbffff70c
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability



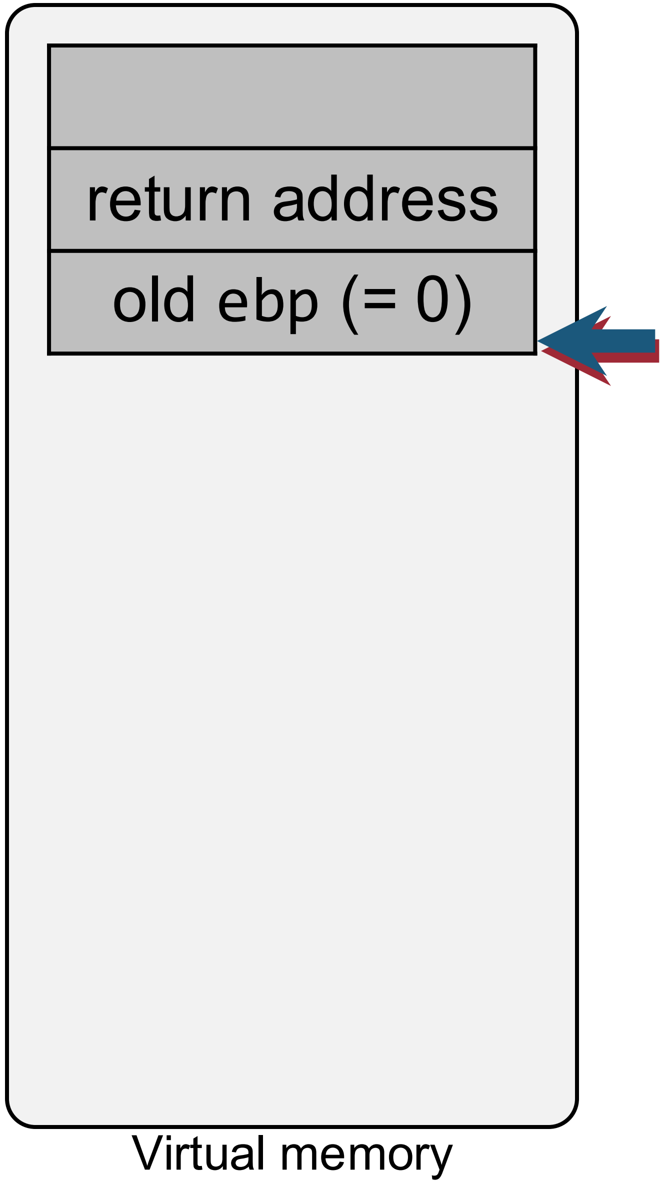
Virtual memory

```
eip: 0x8049163
ebp: 0x0
esp: 0xbffff708
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability

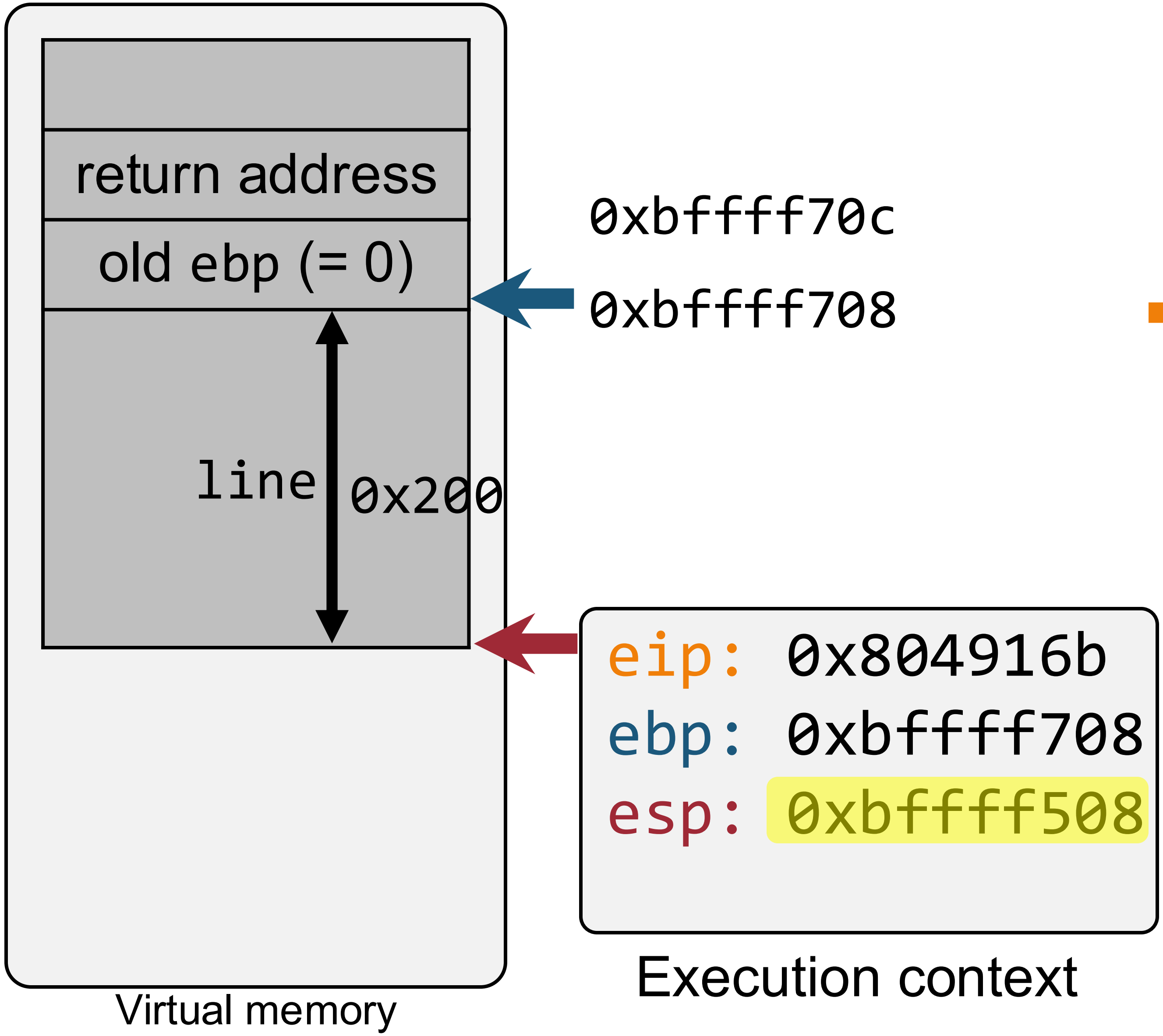


```
eip:  0x8049165
ebp:  0xbffff708
esp:  0xbffff708
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

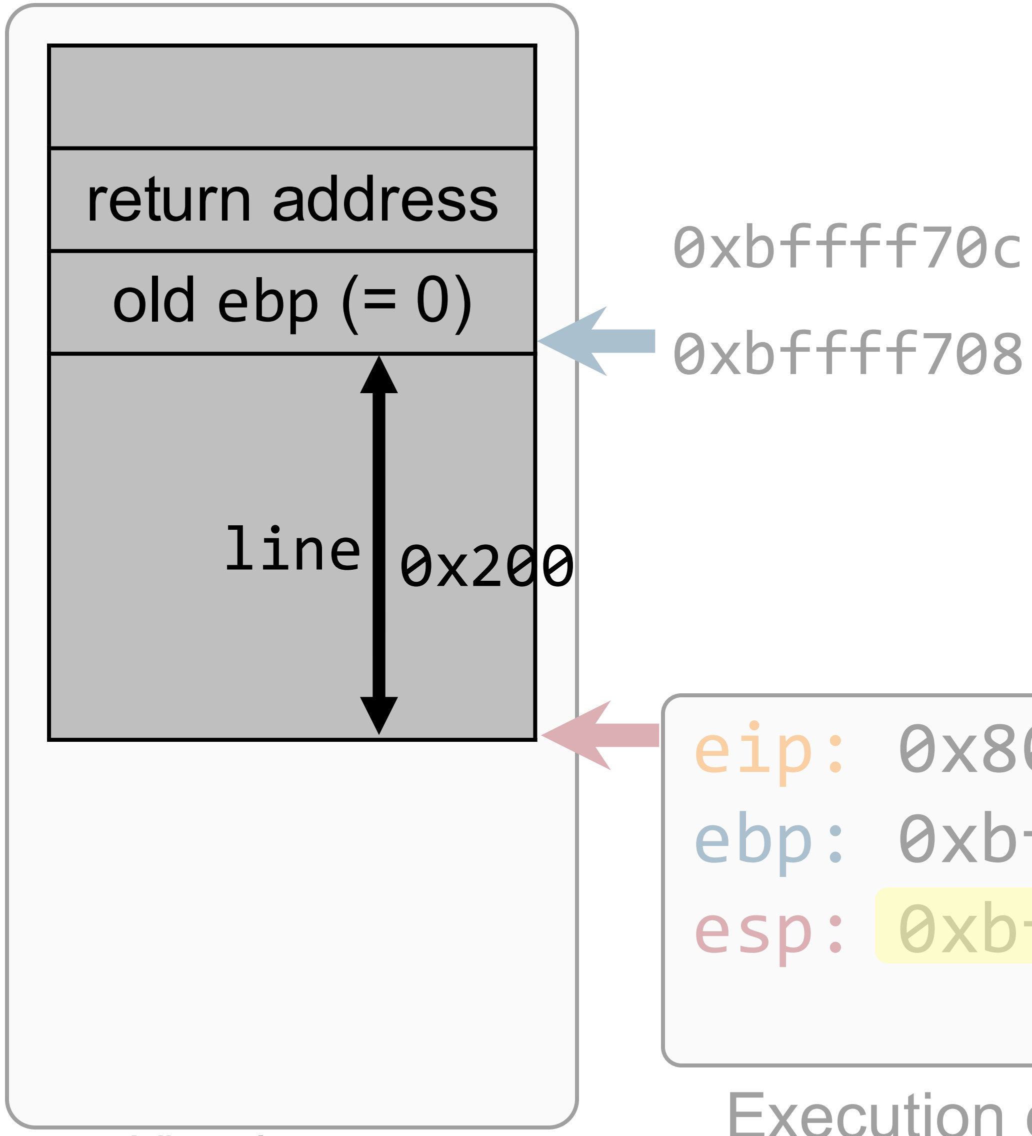
Analyzing the Vulnerability



eip: 0x804916b
ebp: 0xbffff708
esp: 0xbffff508

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability



Virtual memory

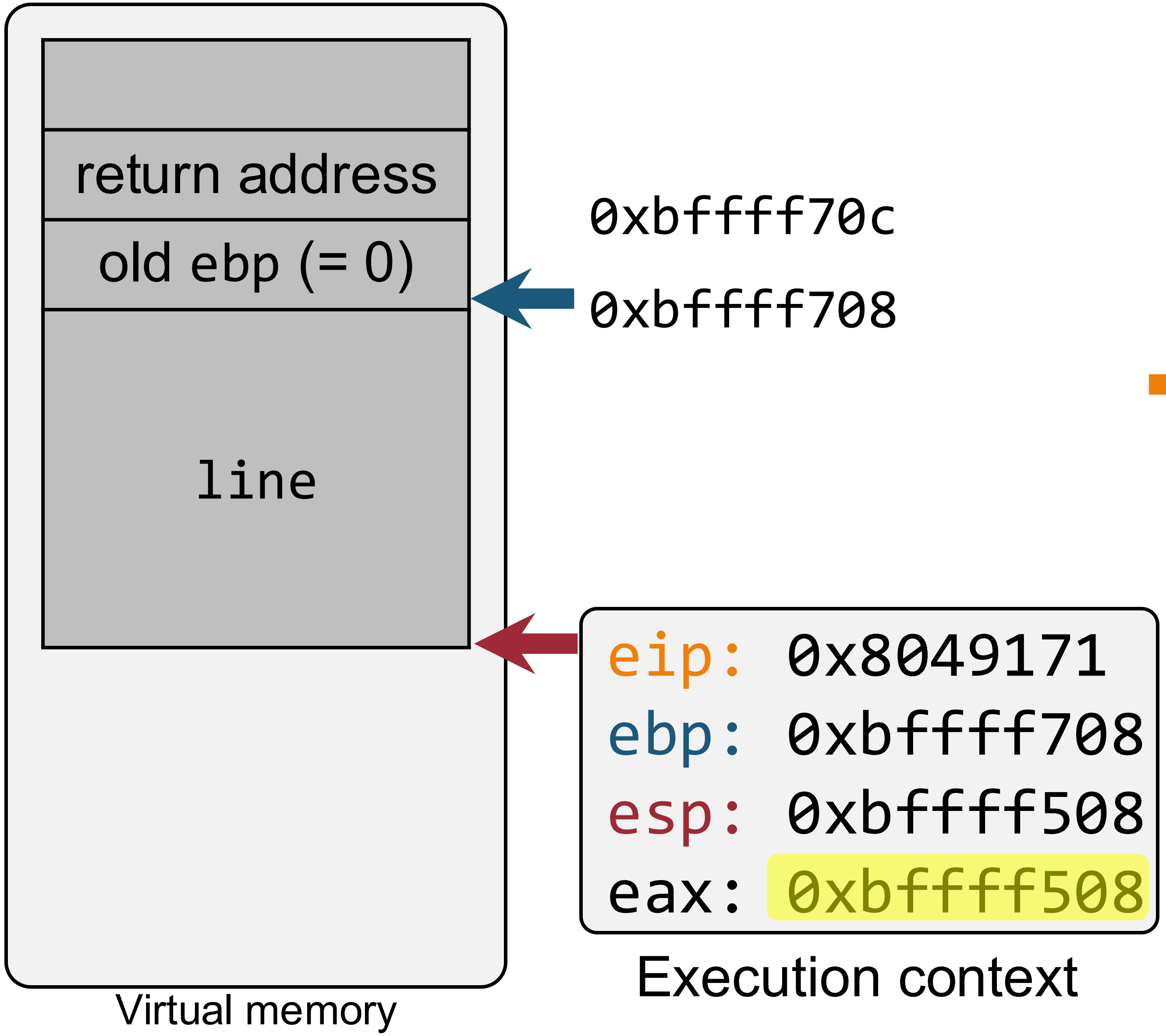
```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
```

```
eip: 0x80
ebp: 0xbf
esp: 0xbf
```

Execution c

```
int main(int argc, char* argv[]) {
  char line[512];
  gets(line);
  return 0;
}
```

Analyzing the Vulnerability

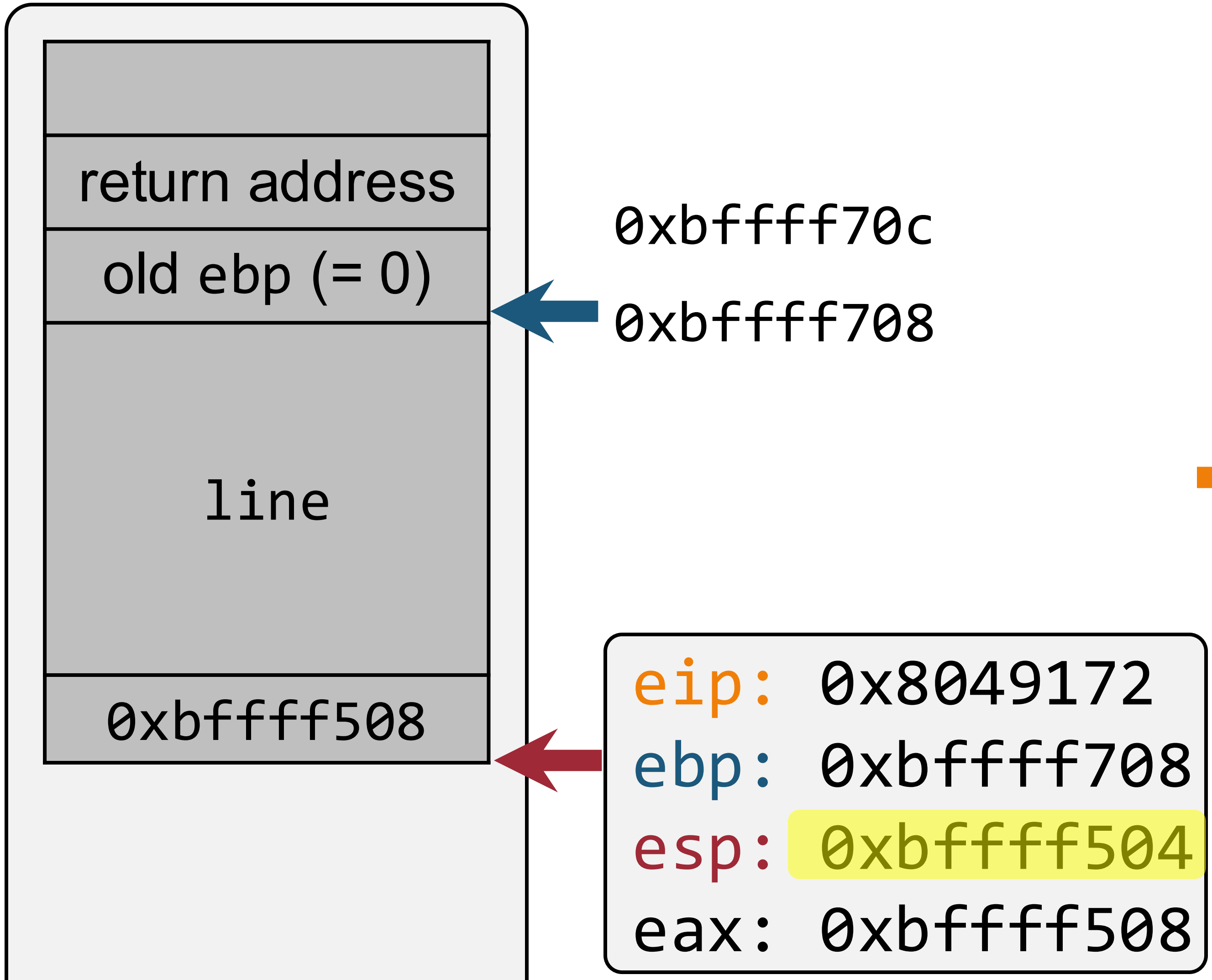


```
eip: 0x8049171
ebp: 0xbffff708
esp: 0xbffff508
eax: 0xbffff508
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability



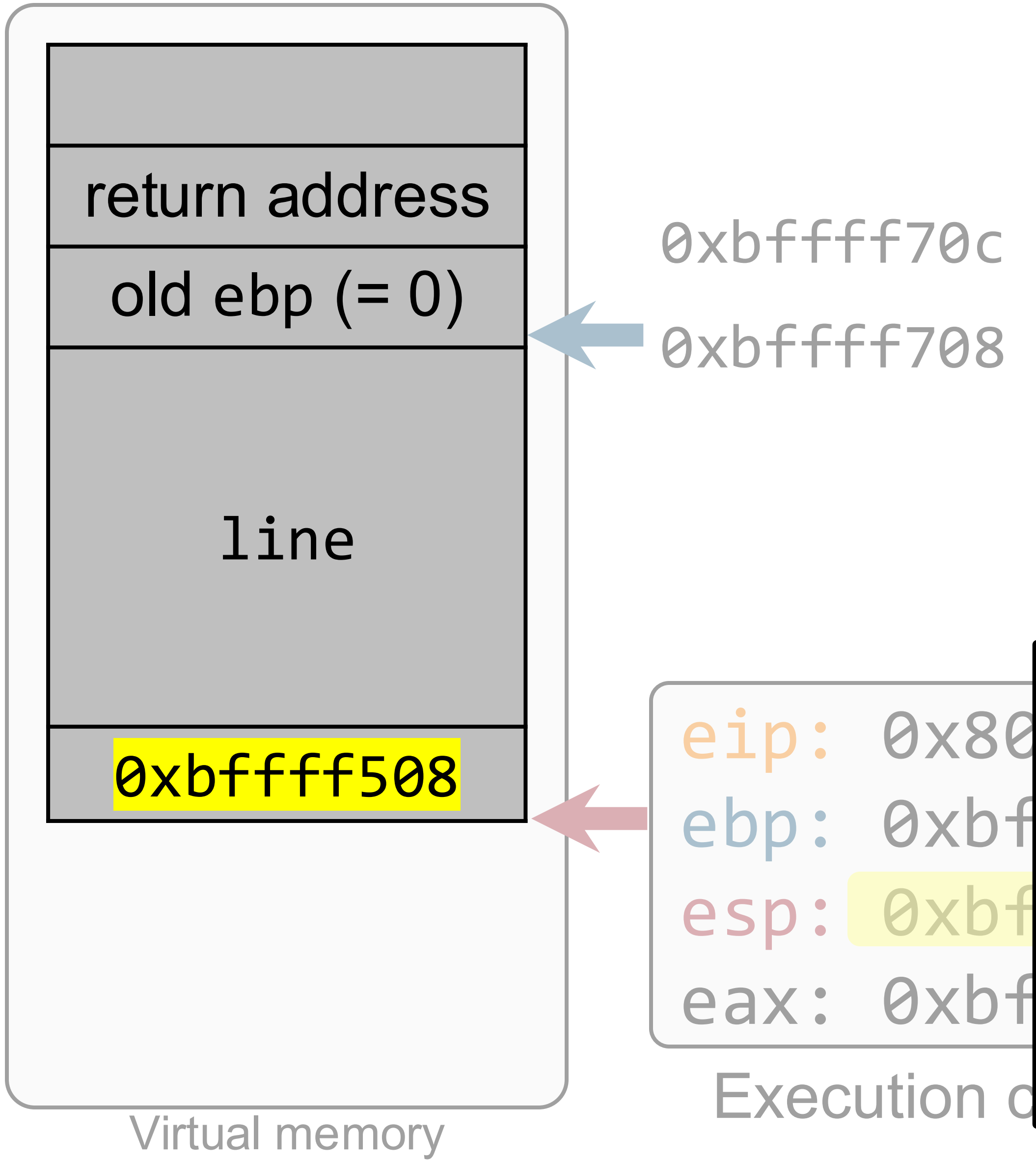
```
eip:  0x8049172
ebp:  0xbffff708
esp:  0xbffff504
eax:  0xbffff508
```

Virtual memory

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability

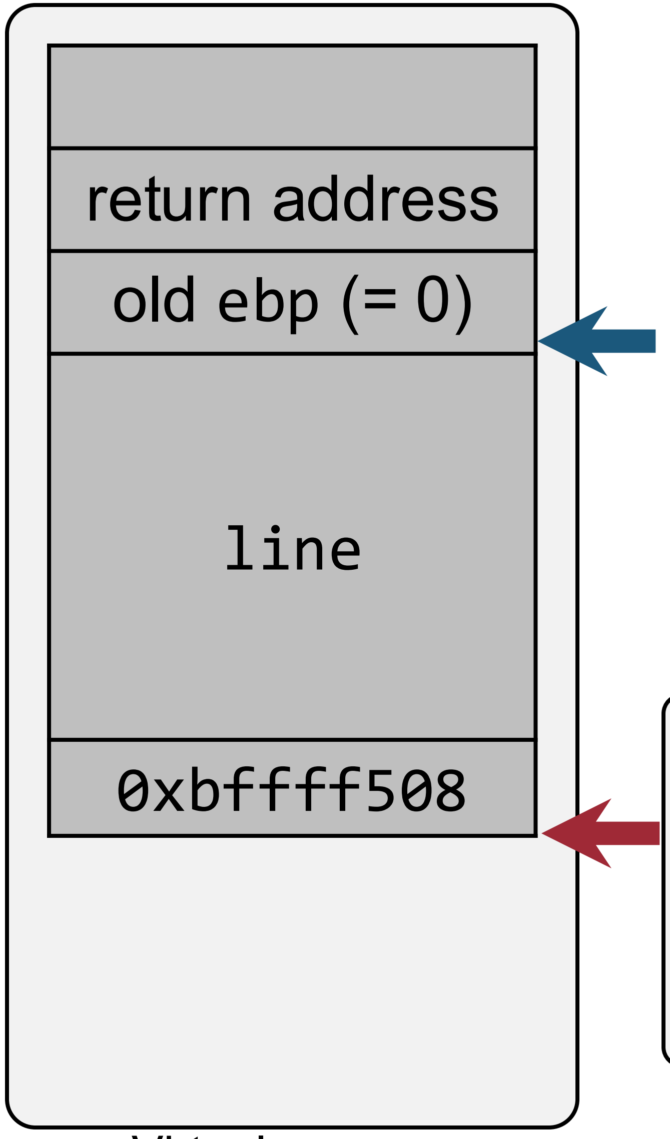


```
08049162 <main>:
8049162:  push    ebp
8049163:  mov     ebp,esp
8049165:  sub     esp,0x200
804916b:  lea     eax,[ebp-0x200]
8049171:  push    eax
8049172:  call    8049030 ; gets
8049177:  add     esp,0x4
```

```
int main(int argc, char* argv[]) {
  char line[512];
  gets(line);
  return 0;
}
```

Address of the variable `line`

Analyzing the Vulnerability



Virtual memory

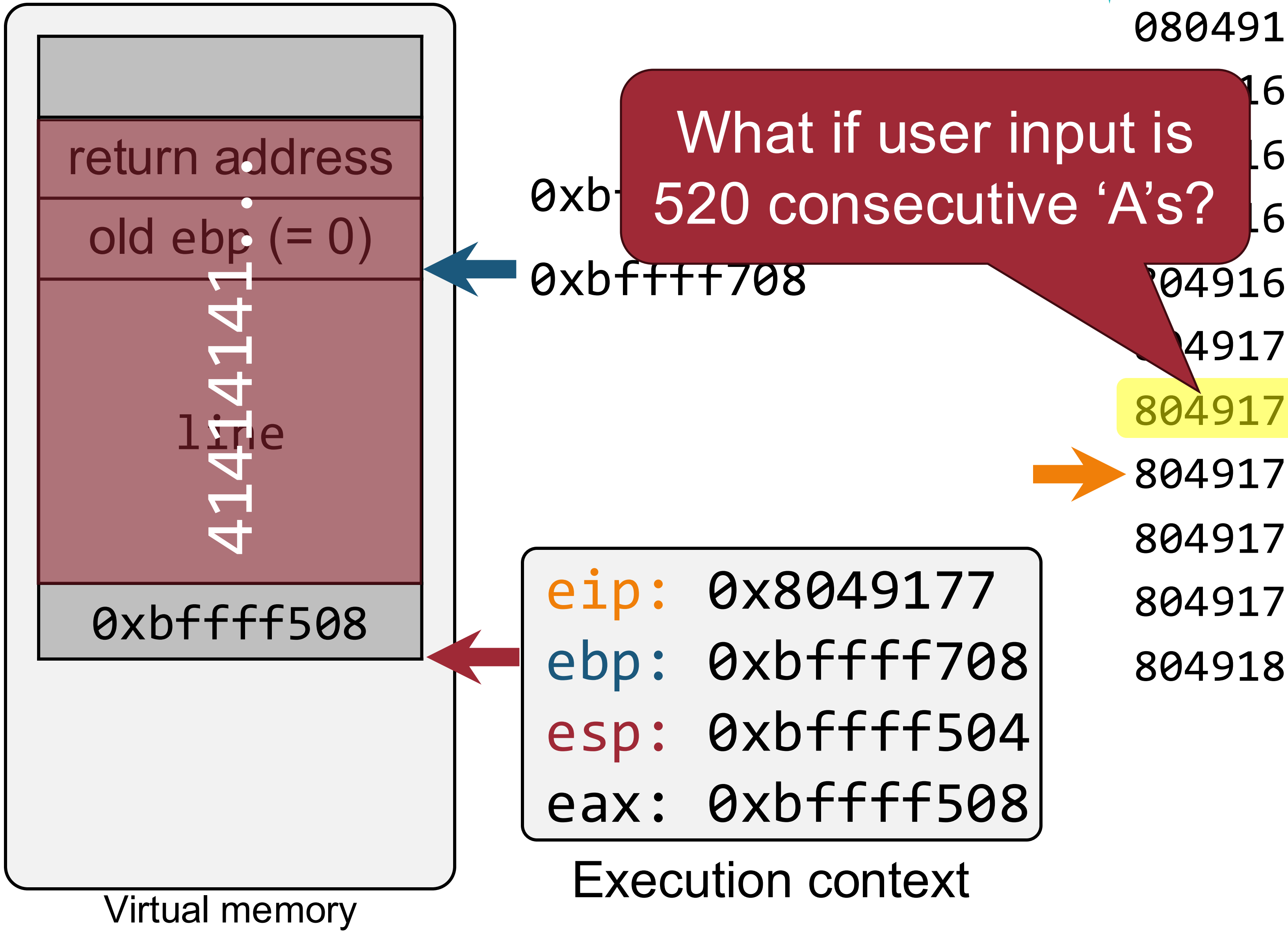
What if user input is 520 consecutive ‘A’s?

```
eip:  0x8049177
ebp:  0xbffff708
esp:  0xbffff504
eax:  0xbffff508
```

Execution context

```
08049162 <main>:
     162:   push    ebp
     163:   mov     ebp,esp
     165:   sub     esp,0x200
 04916b:   lea     eax,[ebp-0x200]
  49171:   push    eax
 8049172:   call    8049030  ; gets
 8049177:   add     esp,0x4
 804917a:   mov     eax,0x0
 804917f:   leave
 8049180:   ret
```

Analyzing the Vulnerability



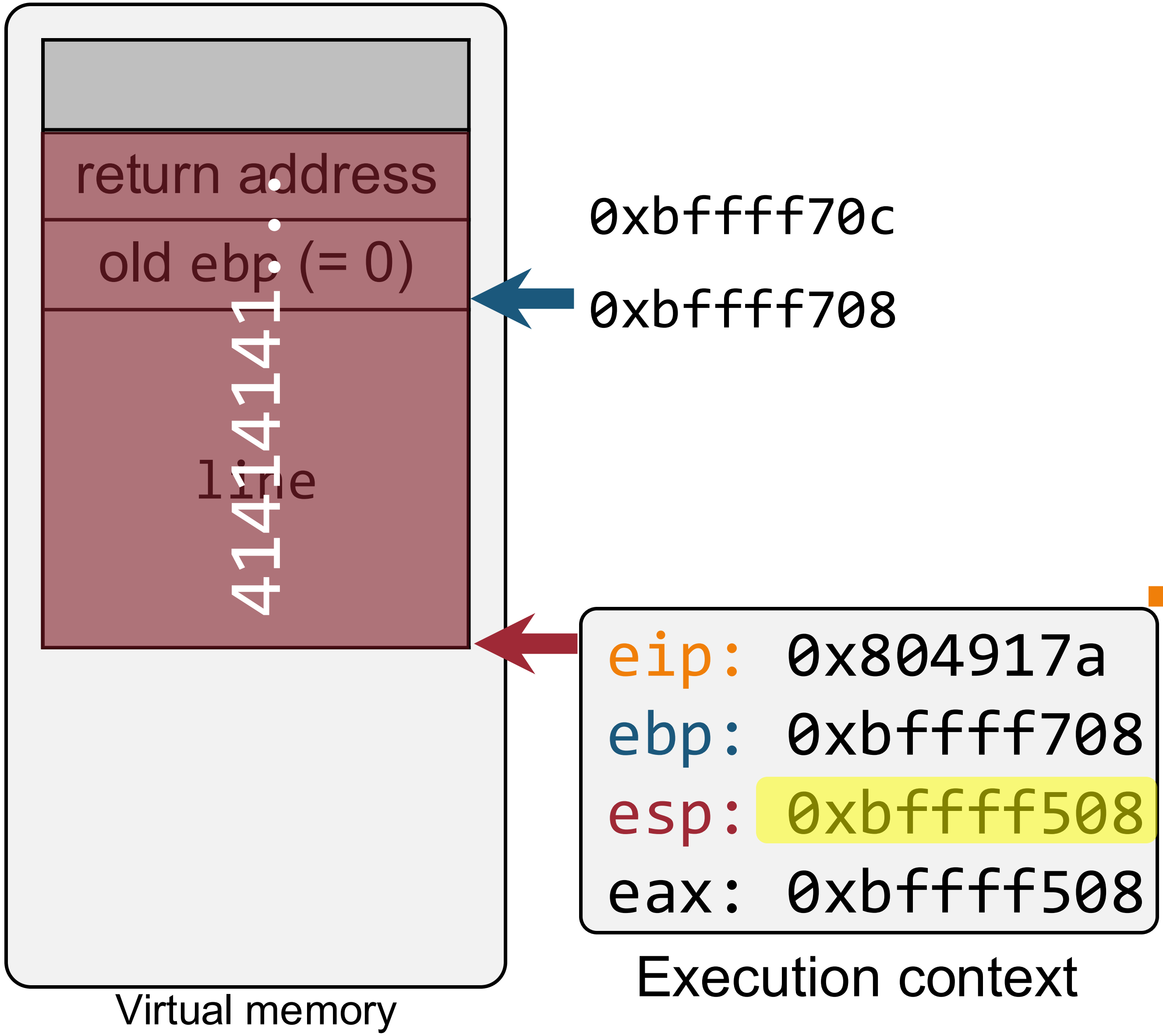
What if user input is 520 consecutive ‘A’s?

```
eip: 0x8049177
ebp: 0xbffff708
esp: 0xbffff504
eax: 0xbffff508
```

Execution context

```
08049162 <main>:
     162:   push    ebp
     163:   mov     ebp,esp
     165:   sub     esp,0x200
 04916b:    lea     eax,[ebp-0x200]
  49171:    push    eax
8049172:    call    8049030 ; gets
8049177:    add     esp,0x4
804917a:    mov     eax,0x0
804917f:    leave
8049180:    ret
```

Analyzing the Vulnerability

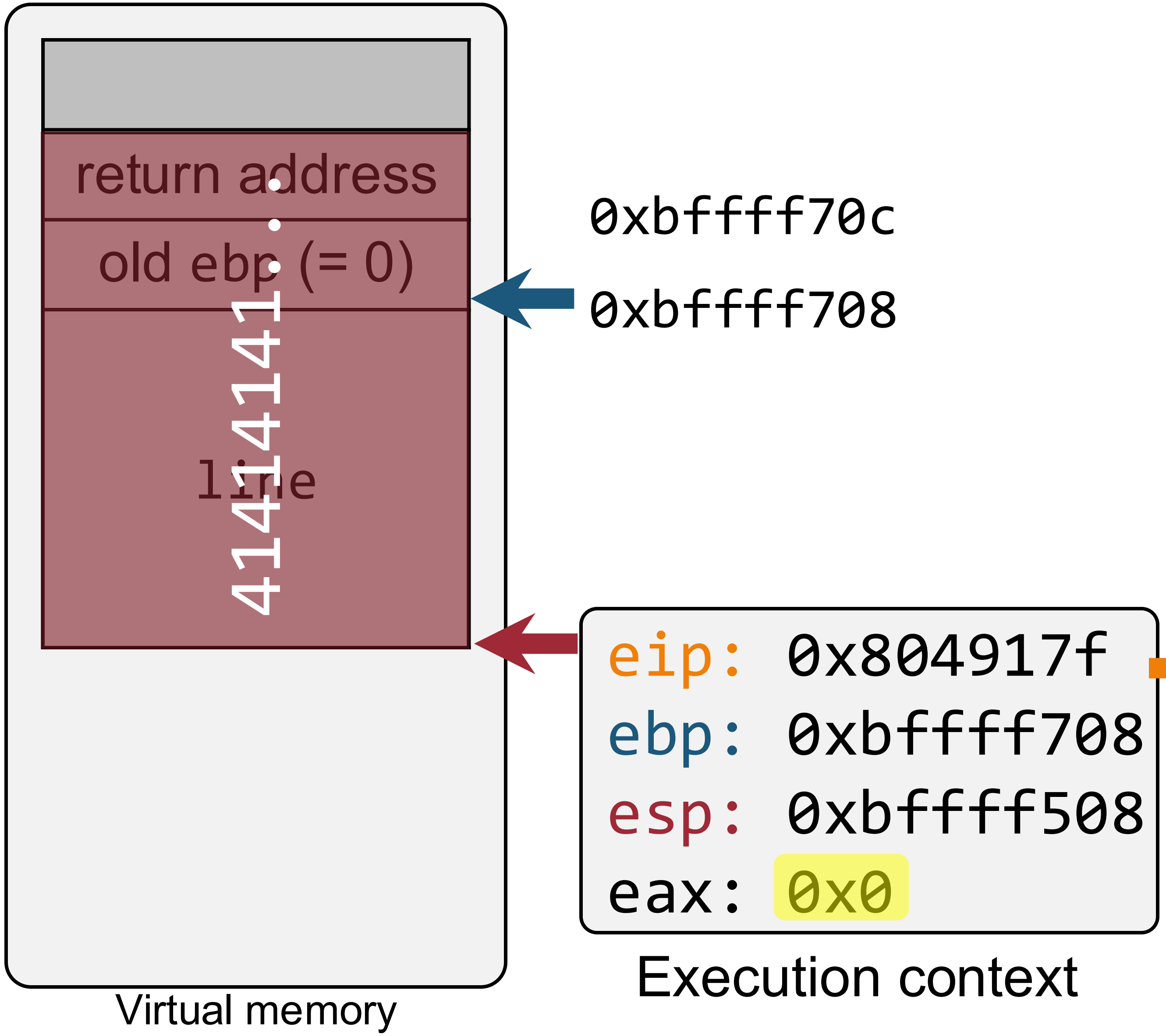


```
eip: 0x804917a
ebp: 0xbffff708
esp: 0xbffff508
eax: 0xbffff508
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability



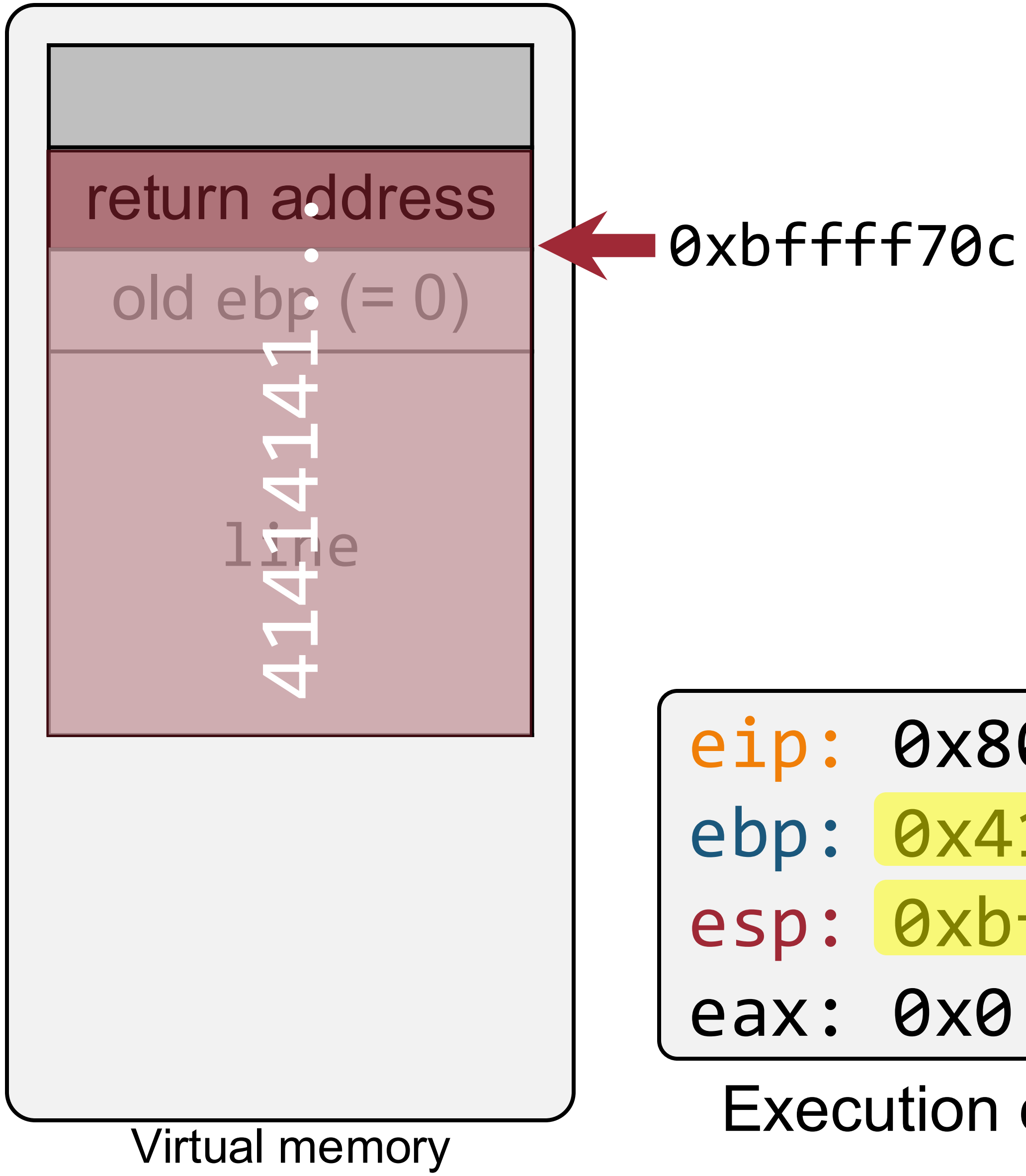
```
eip: 0x804917f
ebp: 0xbffff708
esp: 0xbffff508
eax: 0x0
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

```
mov esp, ebp
pop ebp
```

Analyzing the Vulnerability



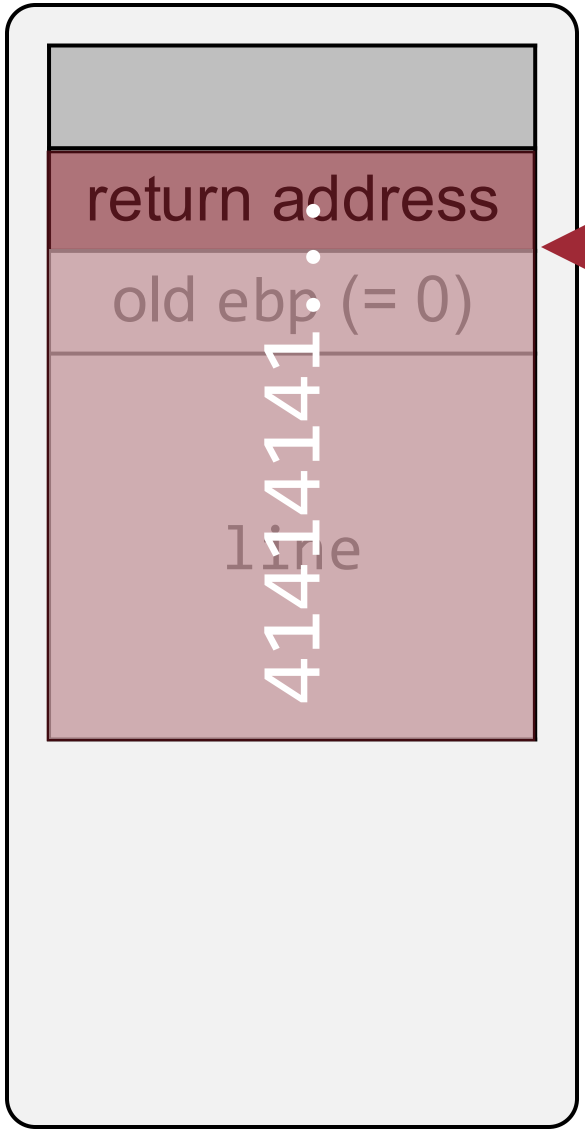
Virtual memory

```
eip:  0x8049180
ebp:  0x41414141
esp:  0xbffff70c
eax:  0x0
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

Analyzing the Vulnerability



0xbffff70c

Virtual memory

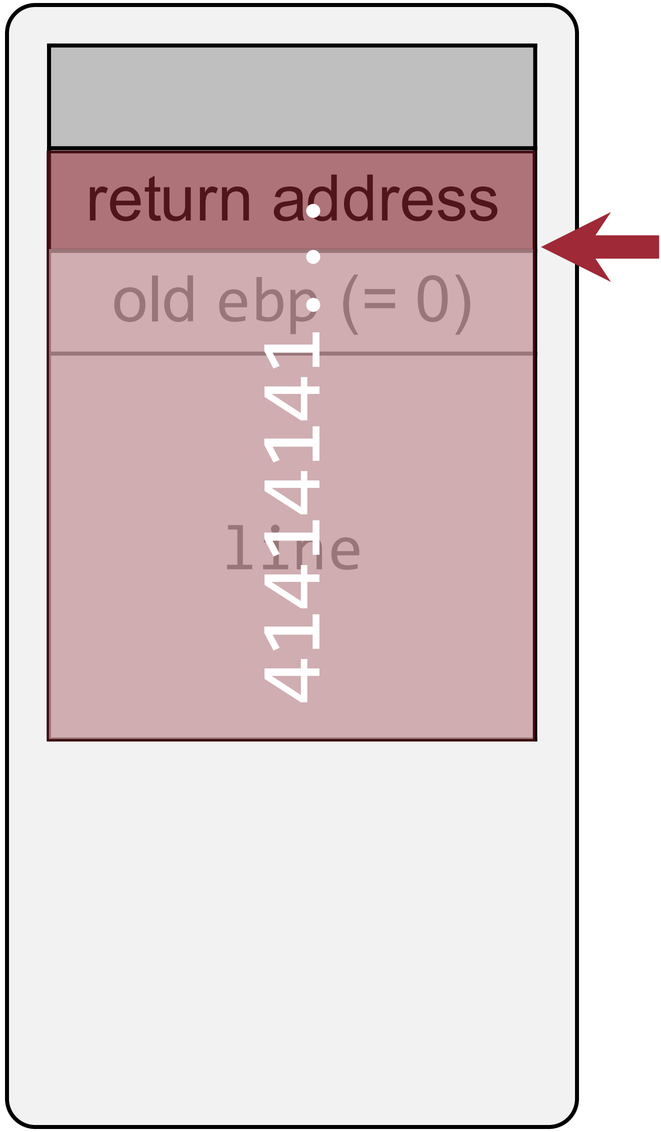
```
eip: 0x8049180
ebp: 0x41414141
esp: 0xbffff70c
eax: 0x0
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

pop eip

Analyzing the Vulnerability



Virtual memory

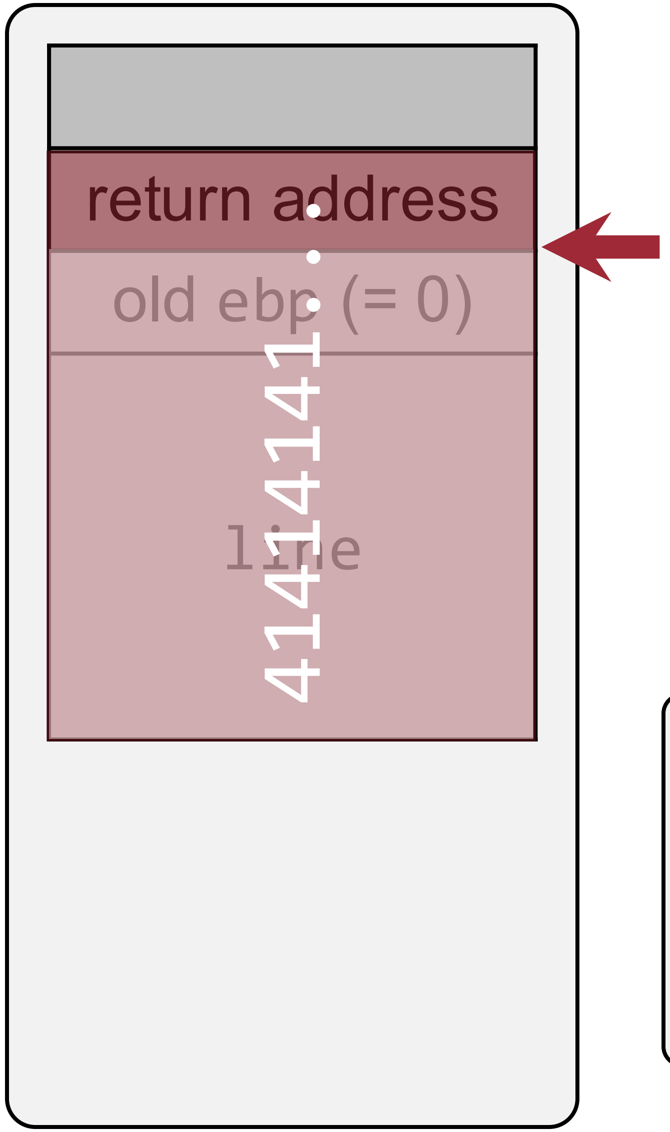
```
eip: 0x41414141
ebp: 0x41414141
esp: 0xbffff70c
eax: 0x0
```

Execution context

```
08049162 <main>:
8049162:   push    ebp
8049163:   mov     ebp,esp
8049165:   sub     esp,0x200
804916b:   lea     eax,[ebp-0x200]
8049171:   push    eax
8049172:   call    8049030 ; gets
8049177:   add     esp,0x4
804917a:   mov     eax,0x0
804917f:   leave
8049180:   ret
```

pop eip

Analyzing the Vulnerability



Virtual memory

Control flow hijacked!

```
eip: 0x41414141
ebp: 0x41414141
esp: 0xbffff70c
eax: 0x0
```

Execution context

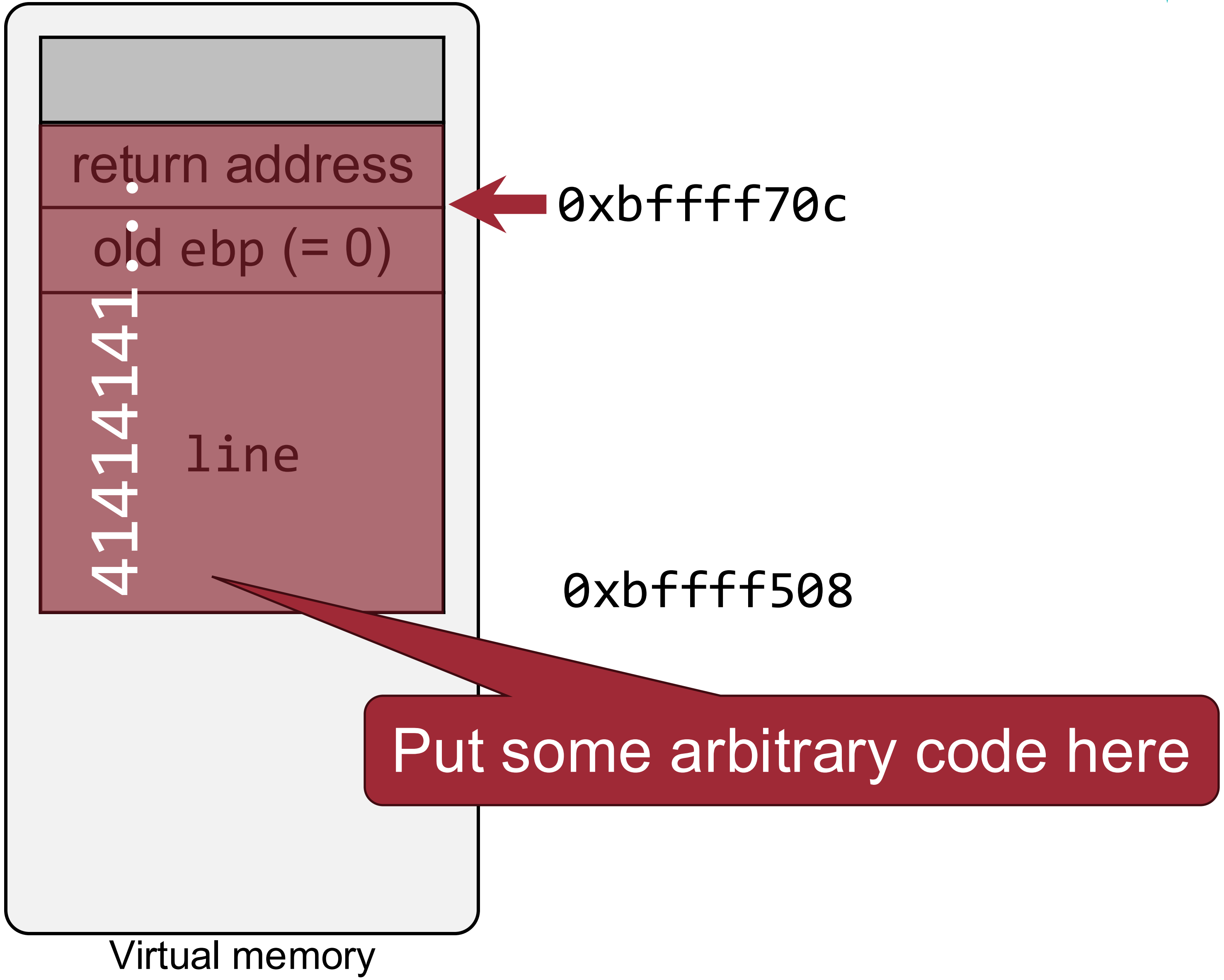
```
08049162 <main>:
8049162:  push   ebp
8049163:  mov    ebp,esp
8049165:  sub    esp,0x200
 4916b:   lea    eax,[ebp-0x200]
 49171:   push   eax
 49172:   call   8049030 ; gets
 049177:  add    esp,0x4
804917a:  mov    eax,0x0
804917f:  leave
8049180:  ret
```

pop eip

So Far …

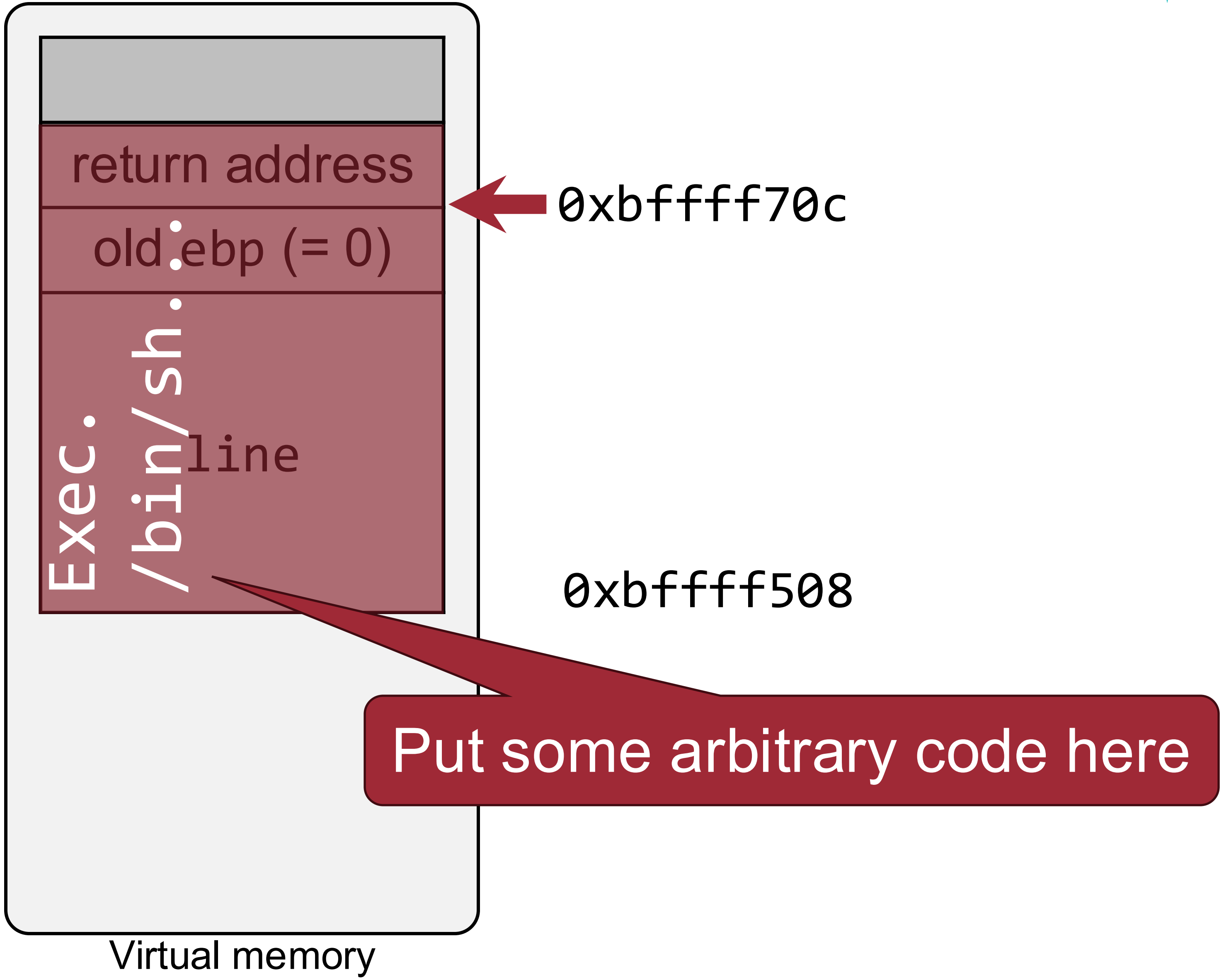
- We hijacked the control flow of the program, i.e., we can jump to any where!
- But, where do we jump to?
- We want to inject some ***arbitrary code*** to run!

Return-to-Stack Exploit



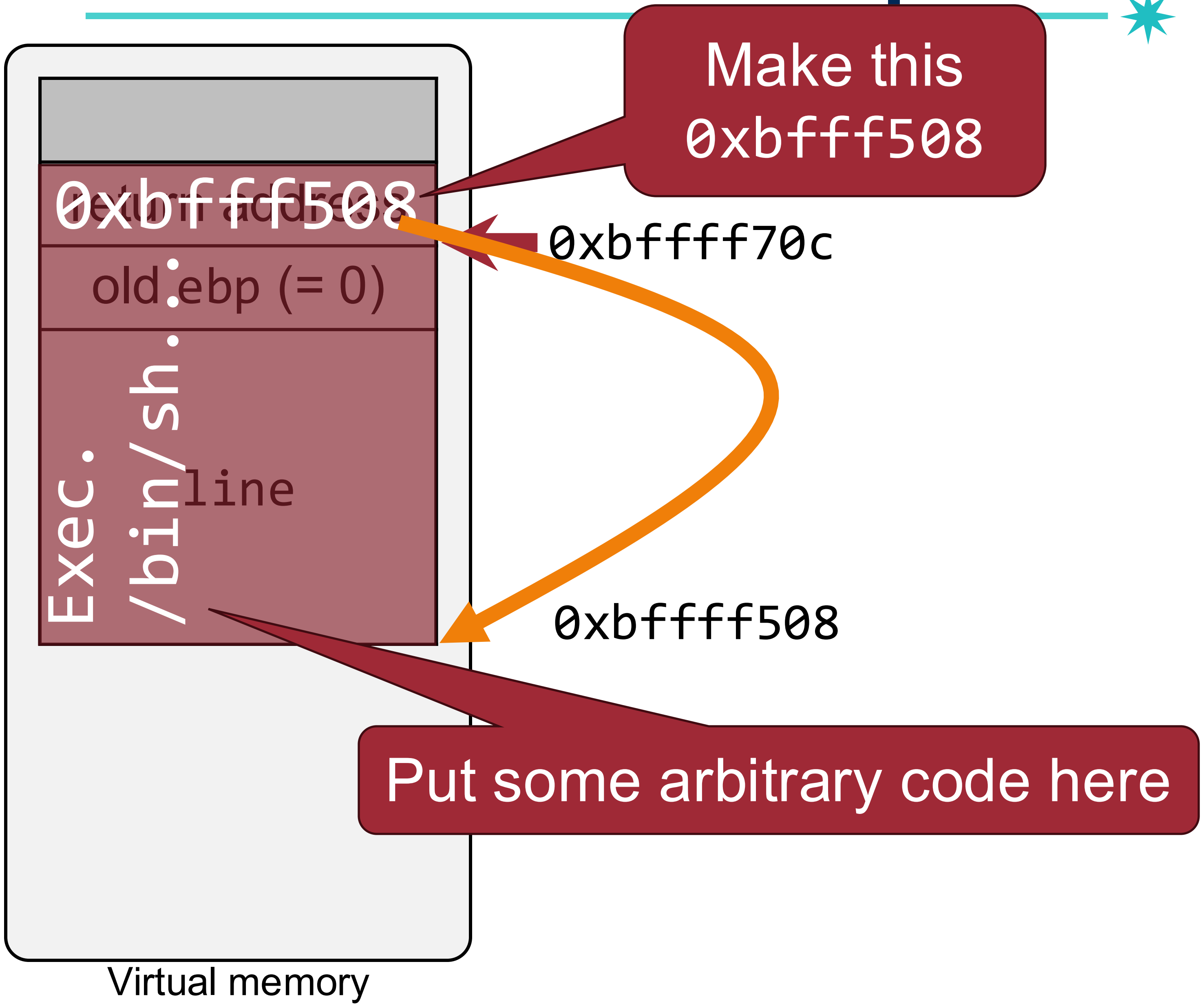
Put some arbitrary code here

Return-to-Stack Exploit



Put some arbitrary code here

Return-to-Stack Exploit



Make this 0xbfff508

Put some arbitrary code here

Executing *Shellcode*

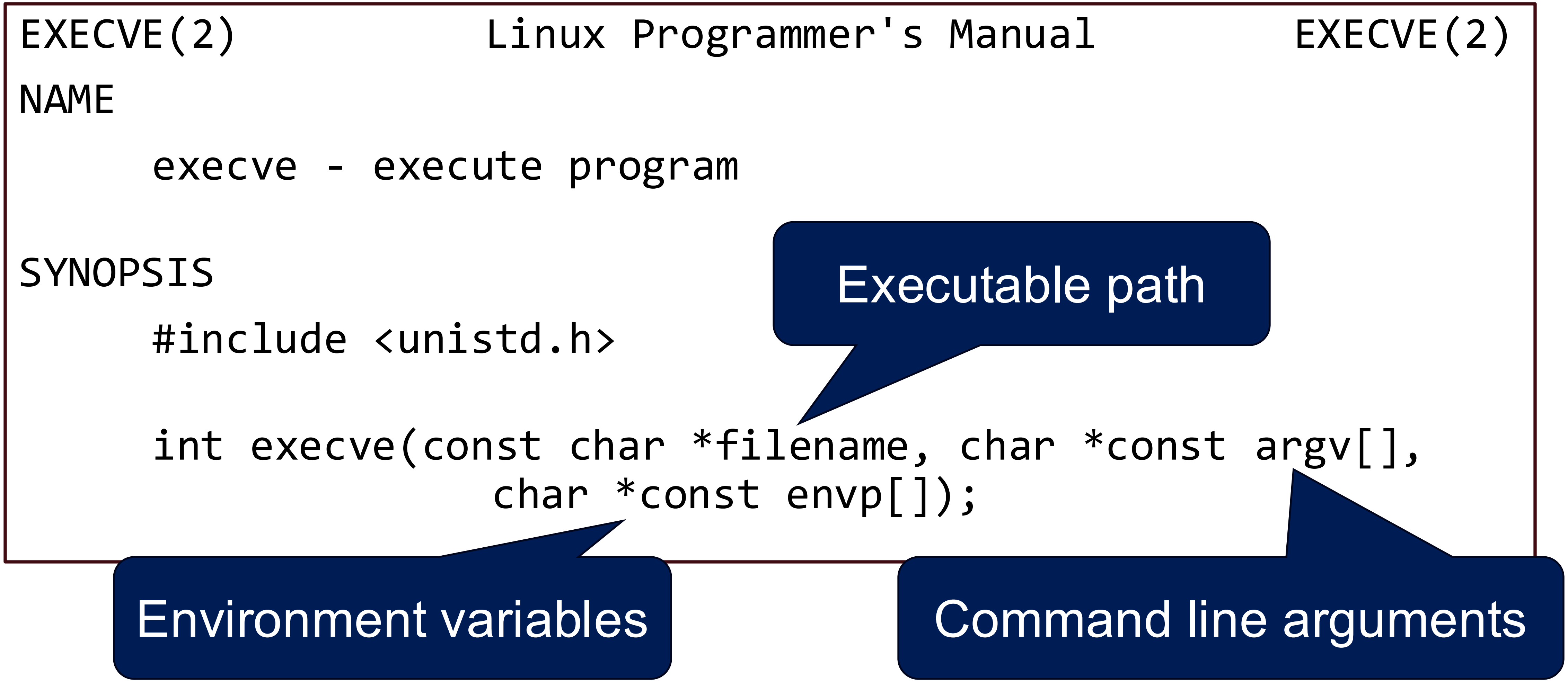
- Shellcode can run any arbitrary logic
  - Download /etc/passwd
  - Install malicious software (malware)
  - …
- Typically, executing `/bin/sh` is enough
  - This is the most powerful attack: we can run arbitrary commands
  - You can also achieve this with relatively ***small piece of code***
  - This is the reason why we call it as ***shellcode*** (code that typically runs shell)

Shellcoding

How to write code that executes `/bin/sh`?

execve() Function in C Library

```
$ man execve
```



```
EXECVE(2)             Linux Programmer's Manual          EXECVE(2)
NAME
     execve - execute program

SYNOPSIS
     #include <unistd.h>

     int execve(const char *filename, char *const argv[],
                             char *const envp[]);
```

Executable path

Environment variables

Command line arguments

Shellcode in C

```
/*
     int execve(const char *filename, char *const argv[],
          char *const envp[]);
*/
```

```
#include <stdio.h>
void main(void) {
  char* argv[] = { "/bin/sh", NULL };
  execve("/bin/sh", argv, NULL);
}
```

Compile with (-static) Option

```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08    mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[ebp-0x8]
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

Is it possible to use this assembly for our exploitation?

Challenge #1: Null Bytes

```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08    mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[ebp-0x8]
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

gets(char *s)

Reads a line from STDIN into the buffer pointed to by s until a terminating new line or EOF, which it replaces with a NULL byte (‘\0’)

Challenge #1: Null Bytes

```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08 mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00 mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[ebp-0x8]
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

Solution:

Write your own assembly code (shellcode) that does not contain any zero (NULL) byte

Challenge #2: String Pointer

```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08 mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00 mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[ebp-0x8]
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

Pointer to “/bin/sh”

Solution:

Push “/bin/sh” string to stack and get the pointer from esp

Challenge #3: External Call (execve)

```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08 mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00 mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[eb
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

Just a wrapper function in the C library (`libc`)

Solution:

We can just inline this function

Challenge #3: External Call (execve)

```
0806c4b0 <__execve>:
 806c4b0:   53                 push   ebx
 806c4b1:   8b 54 24 10        mov    edx,DWORD PTR [esp+0x10]
 806c4b5:   8b 4c 24 0c        mov    ecx,DWORD PTR [esp+0xc]
 806c4b9:   8b 5c 24 08        mov    ebx,DWORD PTR [esp+0x8]
 806c4bd:   b8 0b 00 00 00     mov    eax,0xb
 806c4c2:   cd 80              int    0x80
```

System Call!

Solution:

We can just inline this function

System Calls

allow a program to interface with OS

```
0806c4b0 <__execve>:
 806c4b0:    53                   push    ebx
 806c4b1:    8b 54 24 10          mov     edx,DWORD PTR [esp+0x10]
 806c4b5:    8b 4c 24 0c          mov     ecx,DWORD PTR [esp+0xc]
 806c4b9:    8b 5c 24 08          mov     ebx,DWORD PTR [esp+0x8]
 806c4bd:    b8 0b 00 00 00       mov     eax,0xb
 806c4c2:    cd 80                int     0x80
```

Register	Meaning
eax	System call number
ebx	1st argument
ecx	2nd argument
edx	3rd argument

Register	Meaning
esi	4th argument
edi	5th argument
ebp	6th argument
eax	Return value

List of System Calls for x86

- See: /usr/include/i386-linux-gnu/asm/unistd_32.h

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12 ...
```

0xb

Writing a Shellcode

- Shellcode should run regardless of the address it is loaded. In other words, it should be ***position independent***.

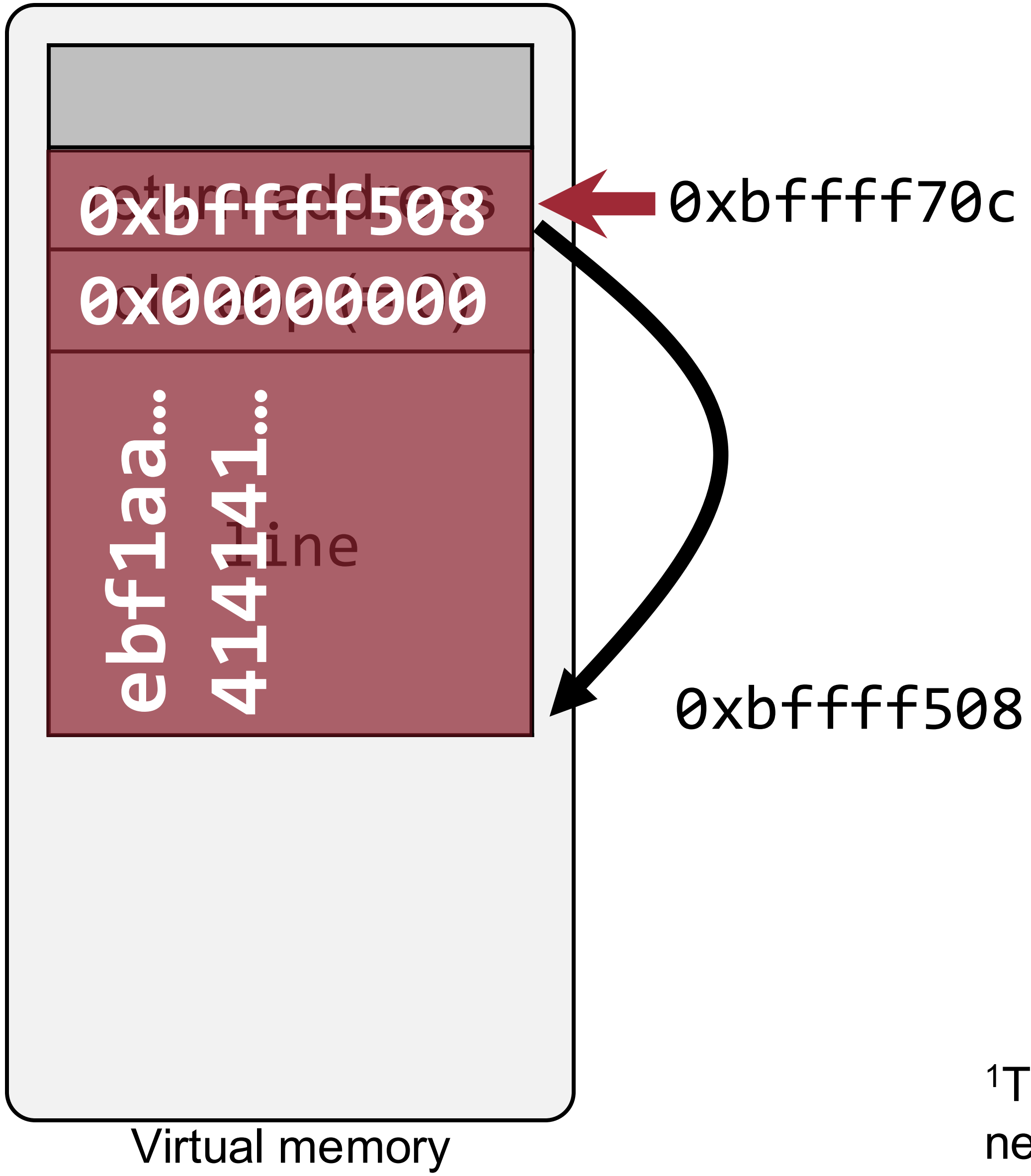
Writing a Shellcode

```
.intel_syntax noprefix

; This is a comment
```

Assemble this code
to see the binary

Final Exploitation



- Fill the buffer with our shellcode (Let’s assume that it is 31 bytes)
- The rest of the buffer (481 bytes = 512-31) can be filled with any characters
- The old ebp can be filled with any characters (4 bytes)
- The return address should point to the shellcode (`0xbffff508`)[1]

[1] The buffer address should differ from machine to machine. Thus, it is necessary to obtain the right address from a debugger (e.g., GDB)

Caveat

We assume that we know the exact address of the buffer

This is very difficult even without modern defenses such as ASLR

Using GDB

- GDB reference: http://www.yolinux.com/TUTORIALS/GDB-Commands.html
- It is recommended to always turn on the Intel syntax by using this command:

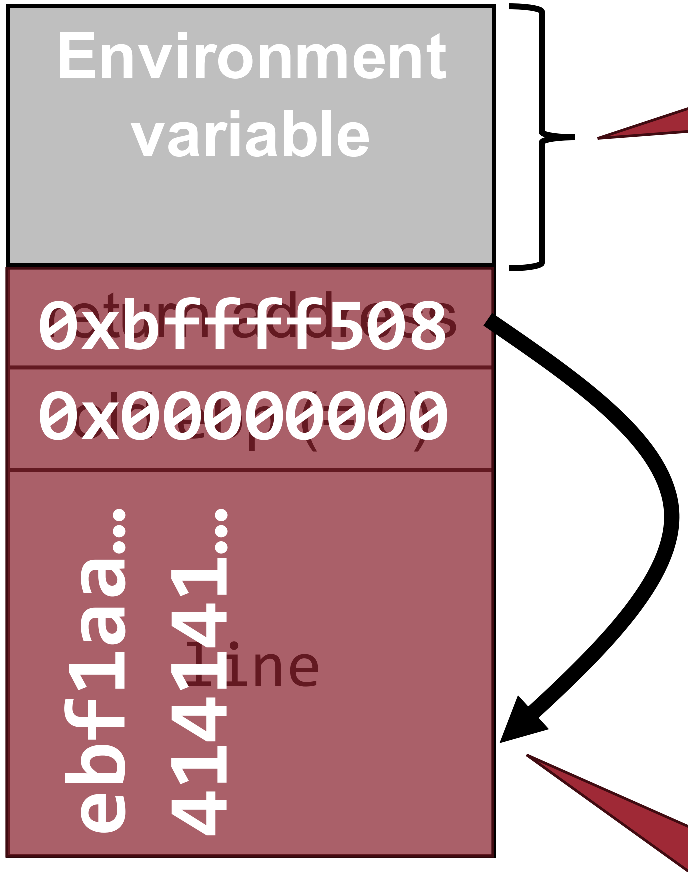
```
$ echo "set disassembly-flavor intel" > ~/.gdbinit
```

Exploit w/ or w/o GDB

- The buffer address identified through GDB is not the same as it without GDB

- Thus, our exploit doesn’t work outside GDB!

Why Different?



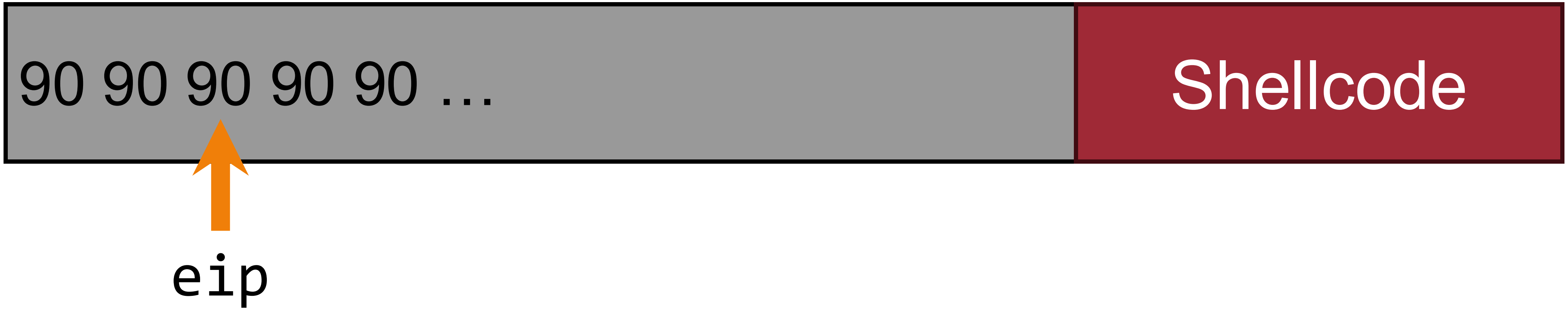
The key problem is
Environment Variables

- GDB puts extra environment variables
- Each machine has different environment variables

Not the start address of the `line`

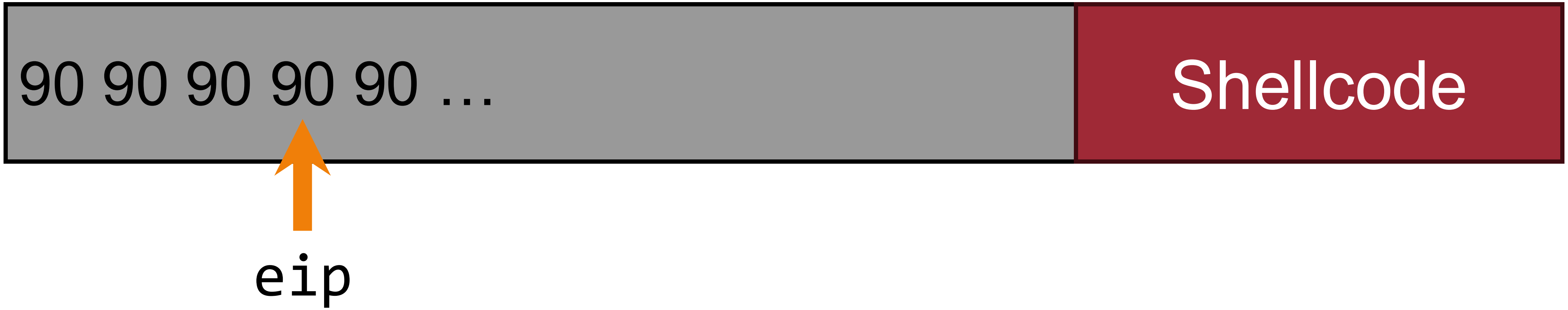
Making Exploit Robust: NOP Sled

- NOP sled (a.k.a., NOP slide) is used to make the exploit robust against different buffer addresses
  - One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`
  - 0x90 represents the NOP instruction



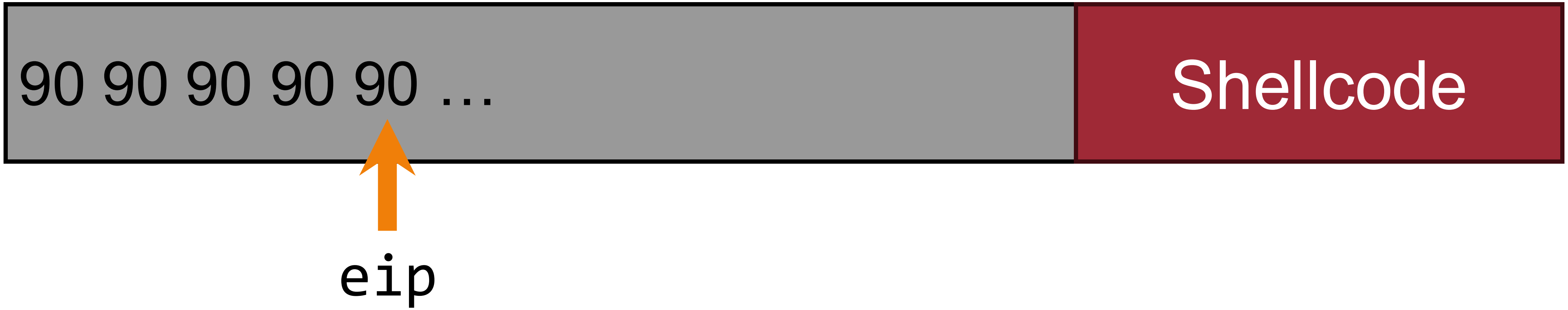
Making Exploit Robust: NOP Sled

- NOP sled (a.k.a., NOP slide) is used to make the exploit robust against different buffer addresses
  - One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`
  - 0x90 represents the NOP instruction



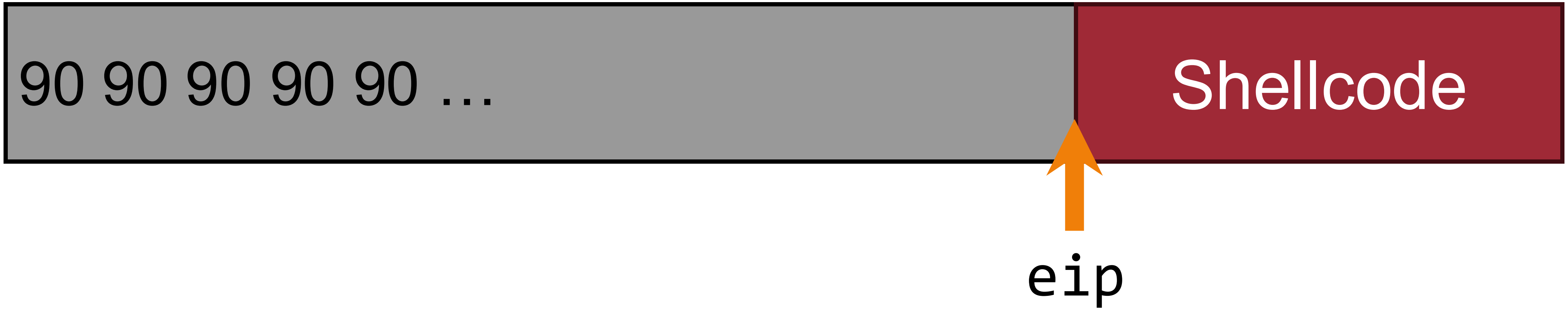
Making Exploit Robust: NOP Sled

- NOP sled (a.k.a., NOP slide) is used to make the exploit robust against different buffer addresses
  - One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`
  - 0x90 represents the NOP instruction



Making Exploit Robust: NOP Sled

- NOP sled (a.k.a., NOP slide) is used to make the exploit robust against different buffer addresses
  - One-byte NO-OP (NOP) instruction is equivalent to `xchg eax, eax`
  - 0x90 represents the NOP instruction



Off-by-One Error

Subtle Error

```
#include <stdio.h>
#include <string.h>
#define BUFSIZE (512)

void printer(char* str) {
  char buf[BUFSIZE];
  strcpy(buf, str);
  printf("%s\n", buf);
}

int main(int argc, char* argv[]) {
  if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
  printer(argv[1]);
  return 0;
}
```

Subtle Error

```
#include <stdio.h>
#include <string.h>
#define BUFSIZE (512)

void printer(char* str) {
  char buf[BUFSIZE];
  strcpy(buf, str);
  printf("%s\n", buf);
}

int main(int argc, char* argv[]) {
  if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
  printer(argv[1]);
  return 0;
}
```

We can just overwrite 1 byte NULL beyond the size of the buffer (buf)

But, some off-by-one bugs are exploitable!

Subtle Error

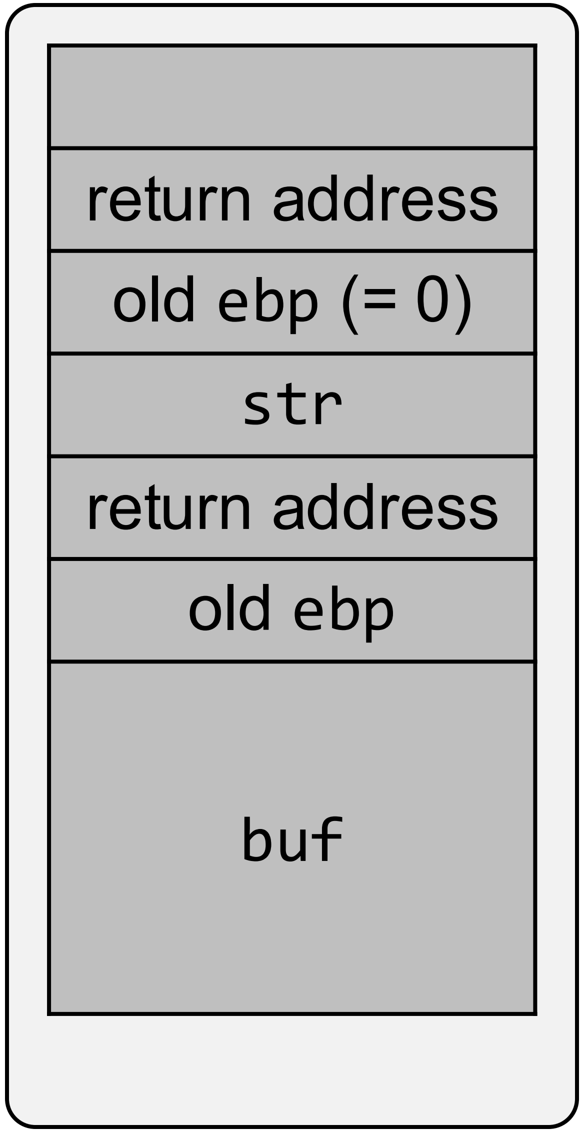
```c
#include <stdio.h>
#include <string.h>
#define BUFSIZE (512)

void printer(char* str) {
  char buf[BUFSIZE];
  strcpy(buf, str);
  printf("%s\n", buf);
}

int main(int argc, char* argv[]) {
  if ( argc < 2 || strlen(argv[1]) > BUFSIZE ) return -1;
  printer(argv[1]);
  return 0;
}
```

Exercise: Can you draw the stack diagram?

Off-by-one Bugs Can be Exploitable



return address

old ebp (= 0)

str

return address

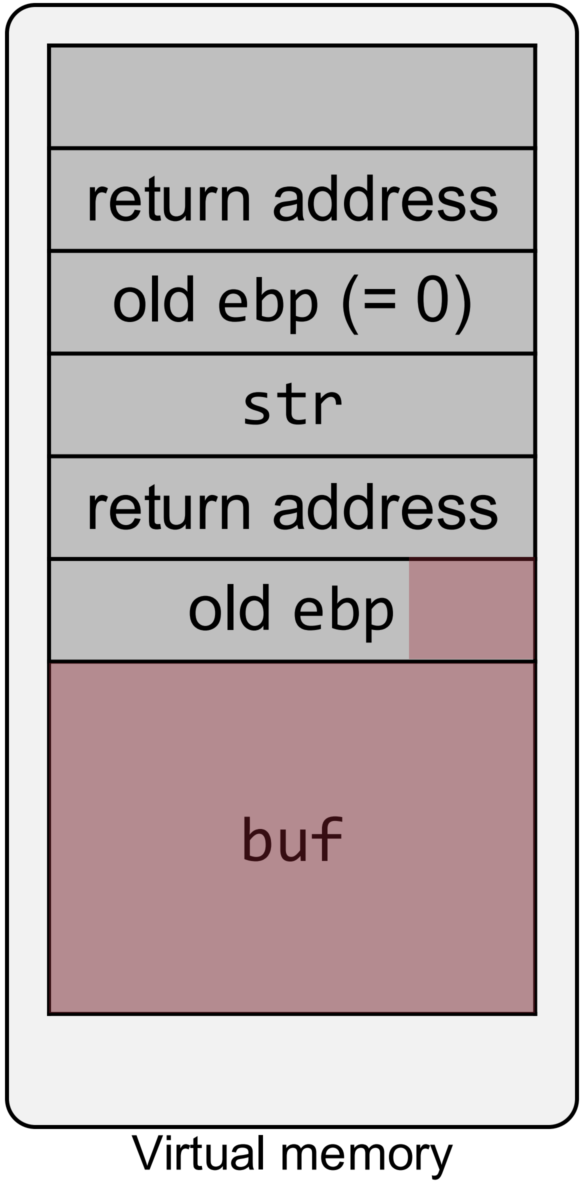
old ebp

buf

Virtual memory

```
leave = mov esp, ebp;
        pop ebp
```

Off-by-one Bugs Can be Exploitable



```
leave = mov esp, ebp;
        pop ebp
```

GDB Usage

- Start: `$ gdb <your binary>`
- Disassemble:

```
(gdb) disass <func name>
```

- Breakpoint setting:

```
(gdb) b *<address>
```

- Run:

```
(gdb) r
```

- Step:

```
(gdb) step # Go to next instruction, diving into function
(gdb) next # Go to next instruction but don’t dive into function
(gdb) continue # Continue normal execution
```

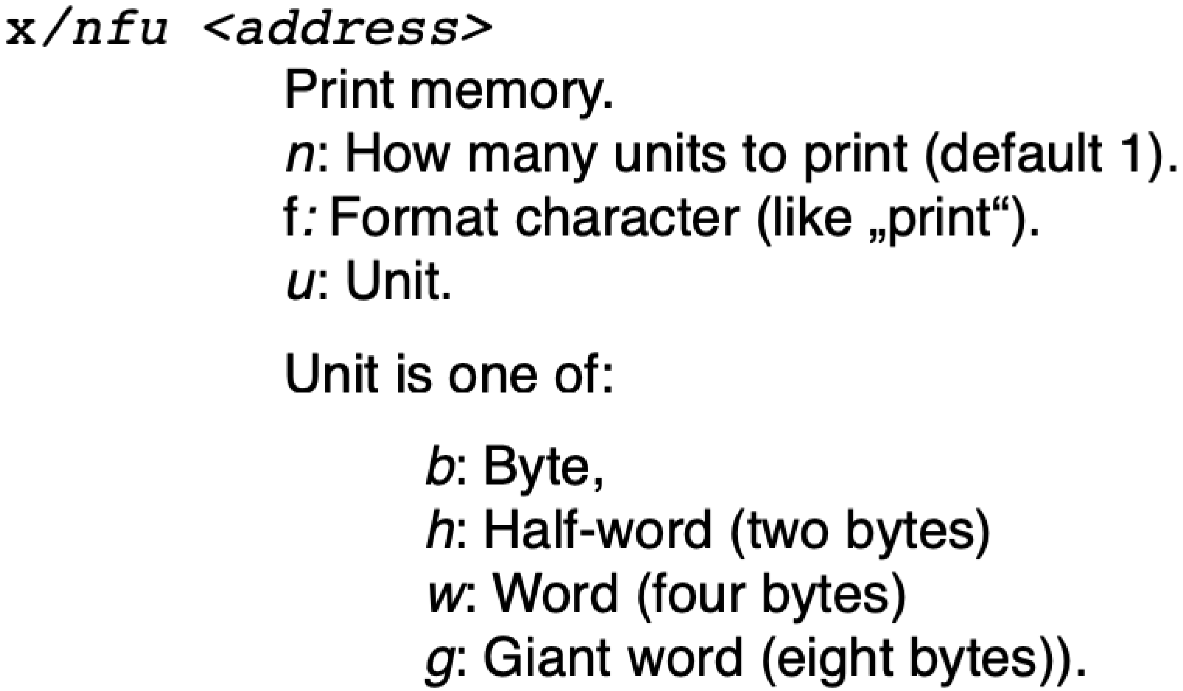
GDB Usage

- Register information:

```
(gdb) i r <register_name>
(gdb) info register
```

- Memory information:

```
(gdb) x/16w <address>
(gdb) x/4w  <register_name>
…
```



`x/nfu <address>`

Print memory.
n: How many units to print (default 1).
f*:* Format character (like „print“).
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

Recommended Reading

GDB Cheatsheet

- https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Smashing the Stack for Fun and Profit, Phrack 1996, by Alphe One

- http://phrack.org/issues/49/14.html

Summary

- Only some bugs are exploitable
- Some exploits allow an attacker to hijack the control flow of the program and to run any arbitrary code
- Return-to-stack exploit puts a shellcode into a stack buffer and jumps to it by overwriting the return address
- We can make return-to-stack exploit robust by using NOP sleds
- Off-by-one errors can sometimes be exploitable

Question?