CSE251: System Programming

7. Assembly (4)

Seongil Wi

Department of Computer Science and Engineering

X86-64 Instructions Summary

- We learned how to move around data
  - `mov, lea, push, pop, etc.`
- We learned how to perform arithmetic and logical operations
  - `add, sub, and, or, etc.`
- We also learned how to control program flows
  - `cmp, jmp, ja, jz, etc.`
- We also leaned about calling conventions

Software Bug

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

Where there is engineering, there is a ***security problem***

Software Bug

- Software bug is an *error/fault/mistake* in the code that produces **an unexpected result**

Thinking like an adversary is important for security

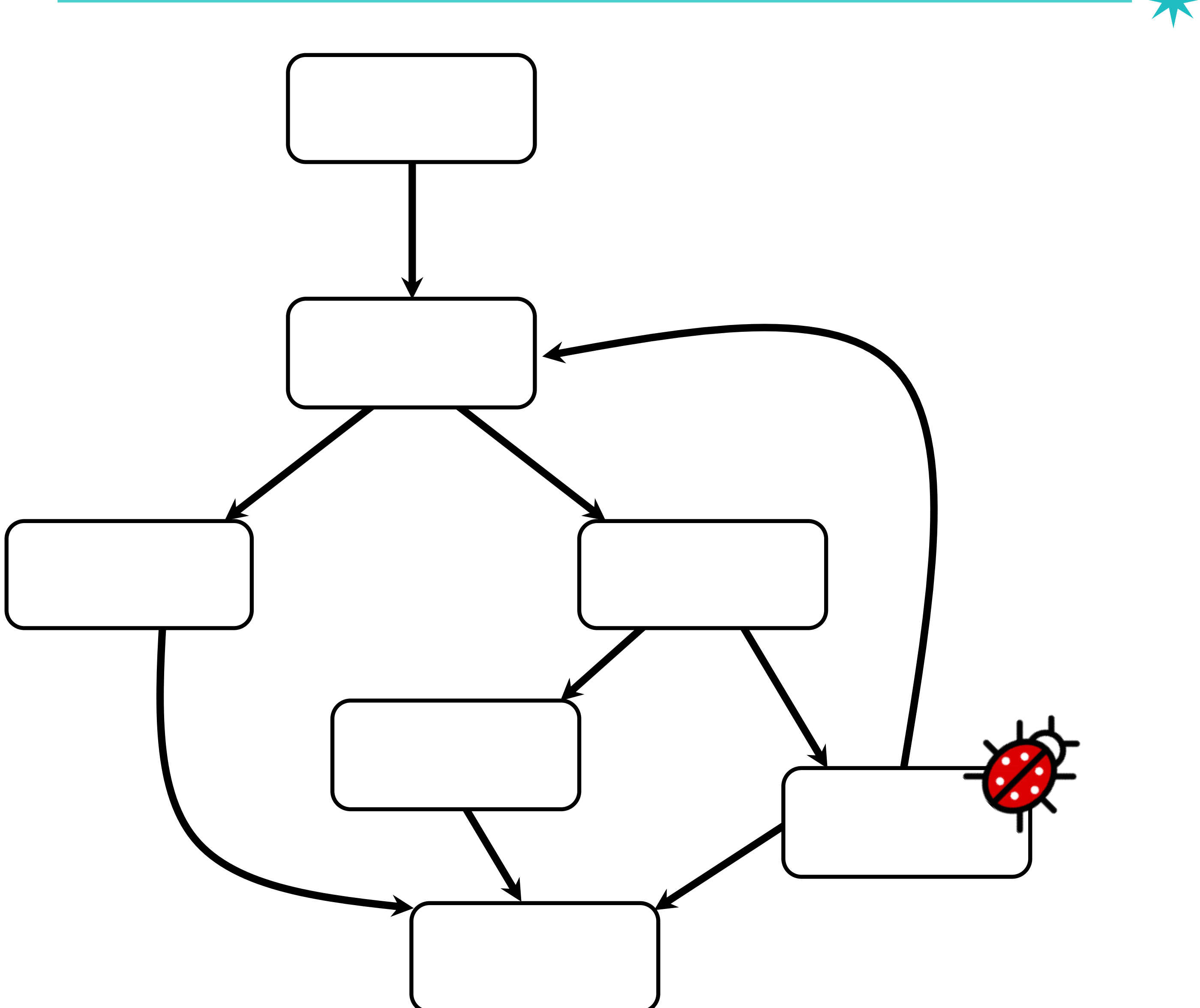
Our Goal

- Find out whether a program is secure or not

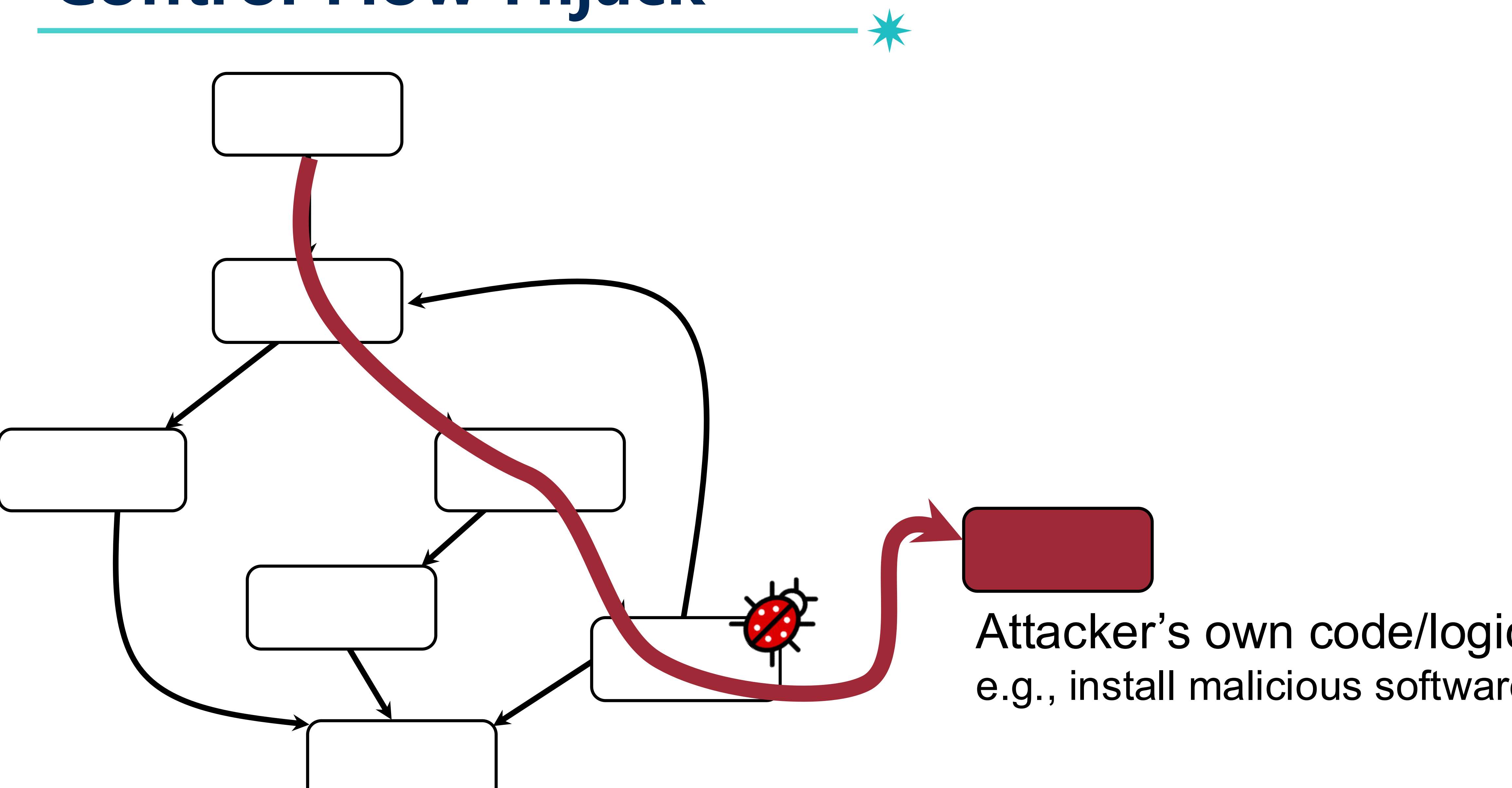
- To do so, we need to see how the ***binary code*** (i.e., assembly) executes on a computer system!

Hacking:
Control Flow Hijack

Control Flow Hijack



Control Flow Hijack



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() {
  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

```c
int main(void) {
   char line[512];
   gets(line);
   return 0;
}
```

Compile this program with:

```
$ gcc -mpreferred-stack-boundary=4
-O0 -fno-stack-protector -fno-pic
-no-pie -fcf-protection=none -z
execstack -o morris morris.c
```

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

401126 <main>:
  401126:  55                    push   rbp
  401127:  48 89 e5              mov    rbp,rsp
  40112a:  48 81 ec 00 02 00 00  sub    rsp,0x200
  401131:  48 8d 85 00 fe ff ff  lea    rax,[rbp-0x200]
  401138:  48 89 c7              mov    rdi,rax
  40113b:  b8 00 00 00 00        mov    eax,0x0
  401140:  e8 eb fe ff ff        call   401030 <gets@plt>
  401145:  b8 00 00 00 00        mov    eax,0x0
  40114a:  c9                    leave
  40114b:  c3                    ret
```

Analyzing the Vulnerability


```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability

return address
old ebp (= 0)

0xbffefff8

0xbffefff0

Virtual memory

```
rip: 0x401127
rbp: 0x0
rsp: 0xbffefff0
```

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability

return address
old ebp (= 0)

0xbffefff8

0xbffefff0

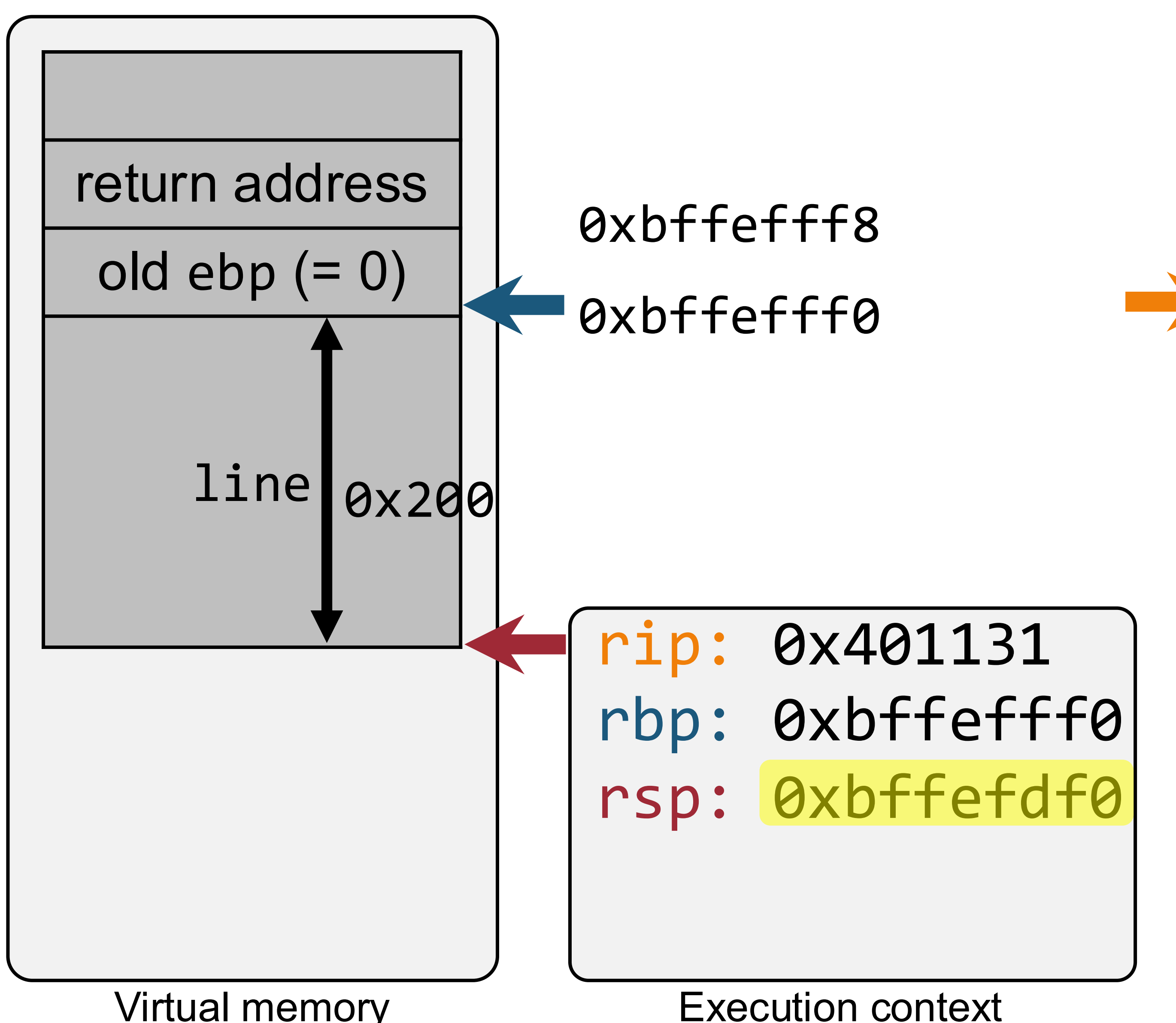
Virtual memory

```
rip: 0x40112a
rbp: 0xbffefff0
rsp: 0xbffefff0
```

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability




```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability

return address

old ebp (= 0)

line

0x200

0xbffefff8

0xbffefff0

Virtual memory

rip: 0x40

rbp: 0xbf

rsp: 0xbf

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
```

```
int main(void) {
  char line[512];
  gets(line);
  return 0;
}
```

Analyzing the Vulnerability

return address
old ebp (= 0)
line ↕ 0x200

0xbffefff8

0xbffefff0

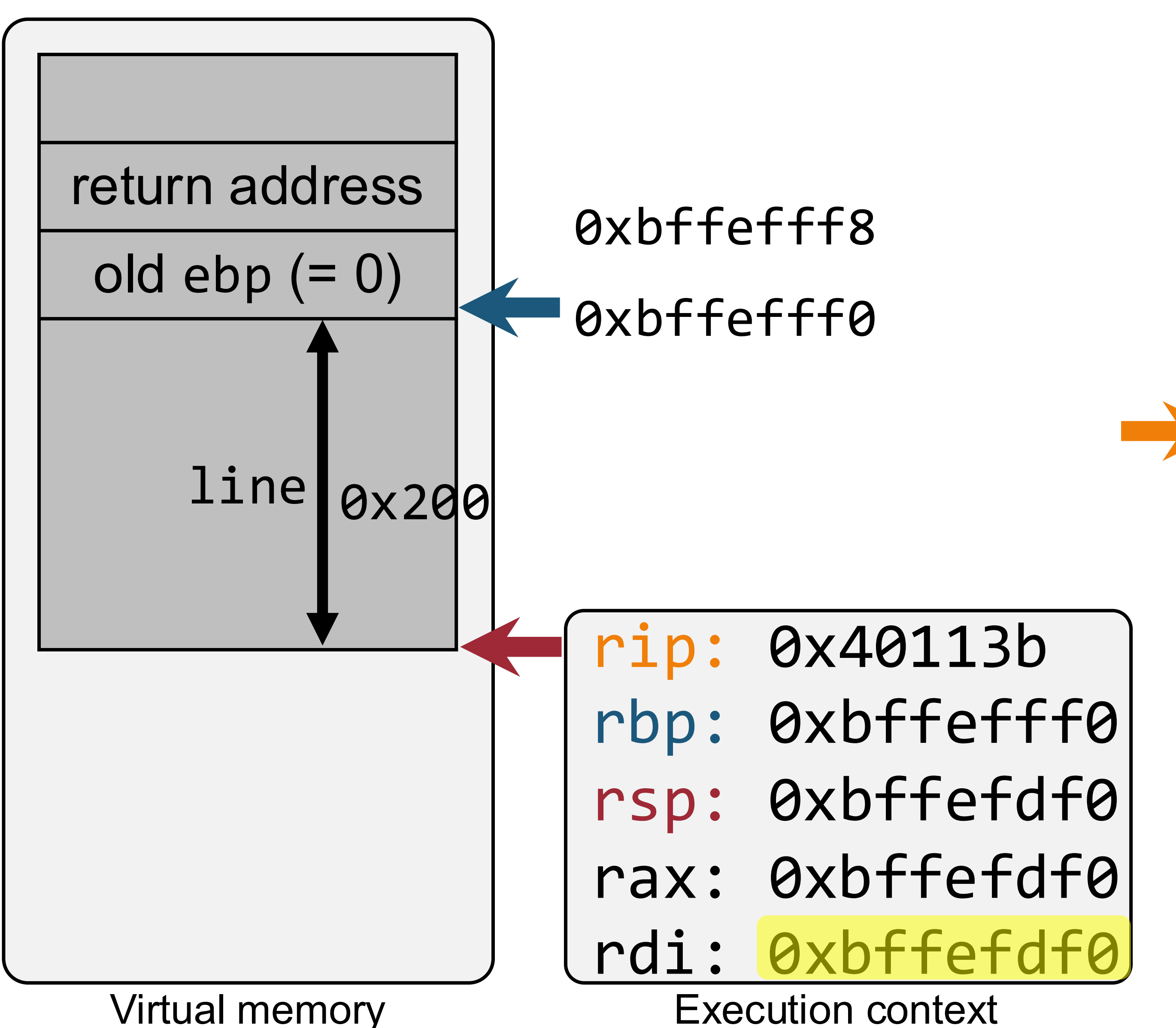
Virtual memory

```
rip:  0x401138
rbp:  0xbffefff0
rsp:  0xbffefdf0
rax:  0xbffefdf0
```

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability




```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

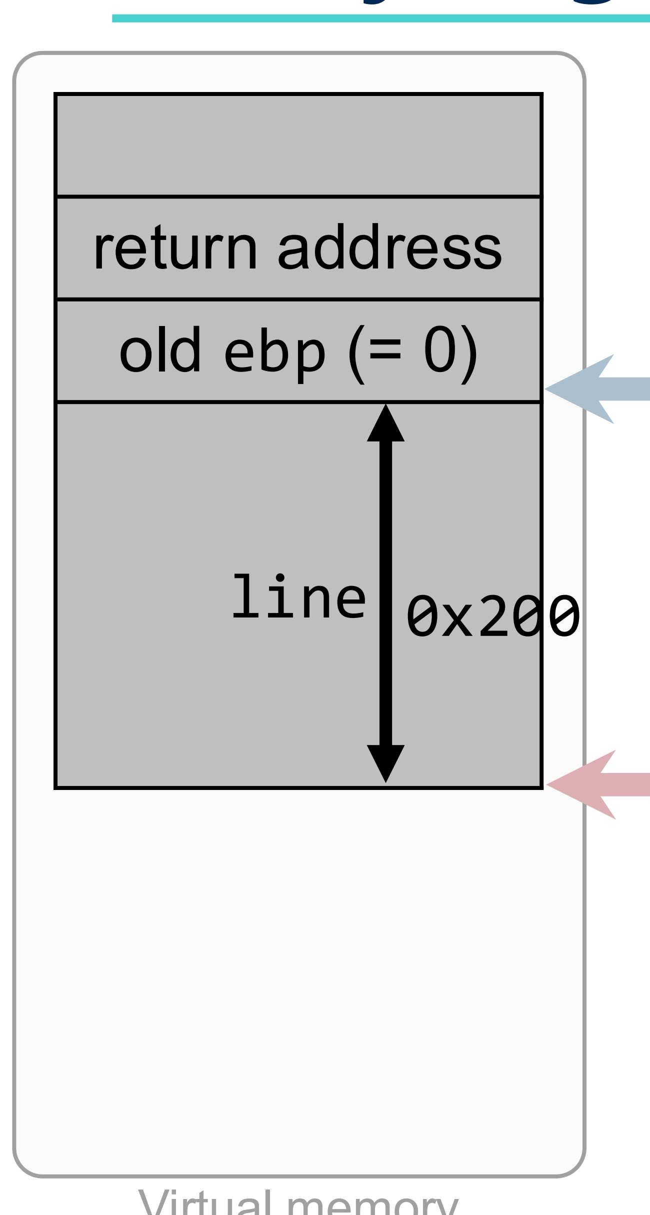
Analyzing the Vulnerability

rip:	0x401145
rbp:	0xbffefff0
rsp:	0xbffefdf0
rax:	0x00000000
rdi:	0xbffefdf0

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability



0xbffefff8

0xbffefff0

```c
int main(void) {
   char line[512];
   gets(line);
   return 0;
}
```

Address of the array `line`

```
rip: 0x401145
rbp: 0xbffefff0
rsp: 0xbffefdf0
rax: 0x00000000
rdi: 0xbffefdf0
```

Execution context

```
40113b: mov   eax,0x0
401140: call  401030 <gets@plt>
401145: mov   eax,0x0
40114a: leave
40114b: ret
```

Analyzing the Vulnerability

```c
int main(void) {
    char line[512];
    gets(line);
    return 0;
}
```

Address of the array `line`

What if user (hacker) input is 520 consecutive 'A's?

Virtual memory:
- return address
- old ebp (= 0)
- line (0x200)
- 41414141 ...

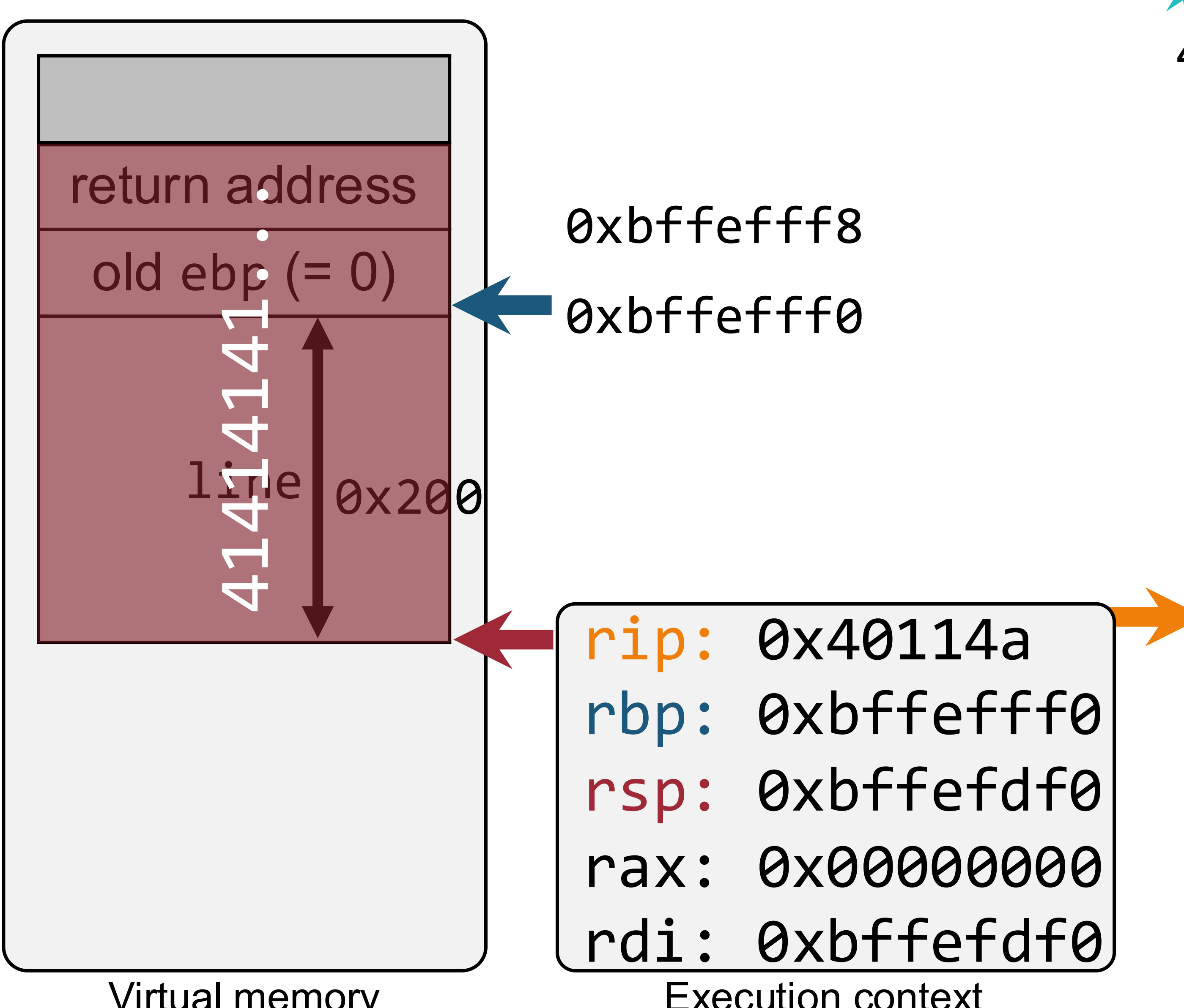
0xbffefff8
0xbffefff0

Execution context:

```
rip:  0x401145
rbp:  0xbffefff0
rsp:  0xbffefdf0
rax:  0x00000000
rdi:  0xbffefdf0
```

```
... 0x0
... 0 <gets@plt>
... x0
40114a: leave
40114b: ret
```

Analyzing the Vulnerability




```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

Analyzing the Vulnerability

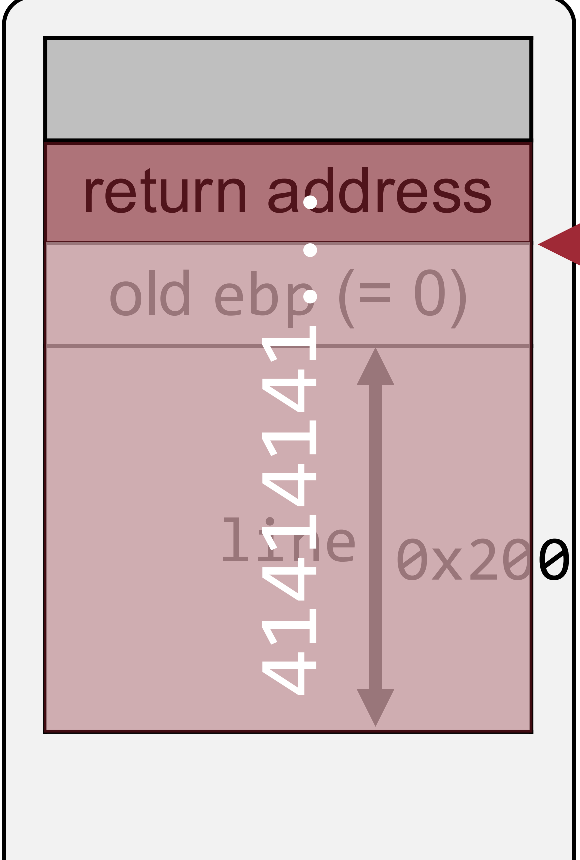
rip:	0x40114b
rbp:	0xbffefff0
rsp:	0xbffefdf0
rax:	0x00000000
rdi:	0xbffefdf0

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

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

Analyzing the Vulnerability



0xbffefff8

Virtual memory

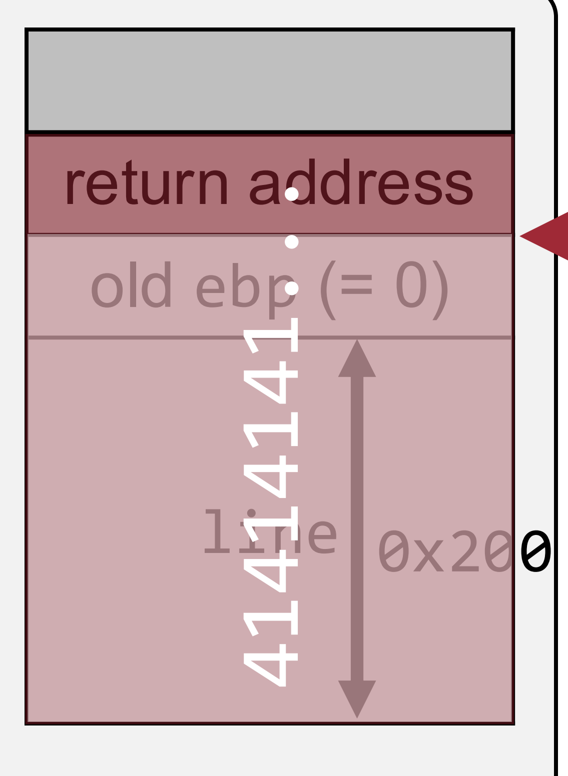
rip:	0x40114b
rbp:	0x41414141
rsp:	0xbffefff8
rax:	0x00000000
rdi:	0xbffefdf0

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

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

Analyzing the Vulnerability



Virtual memory

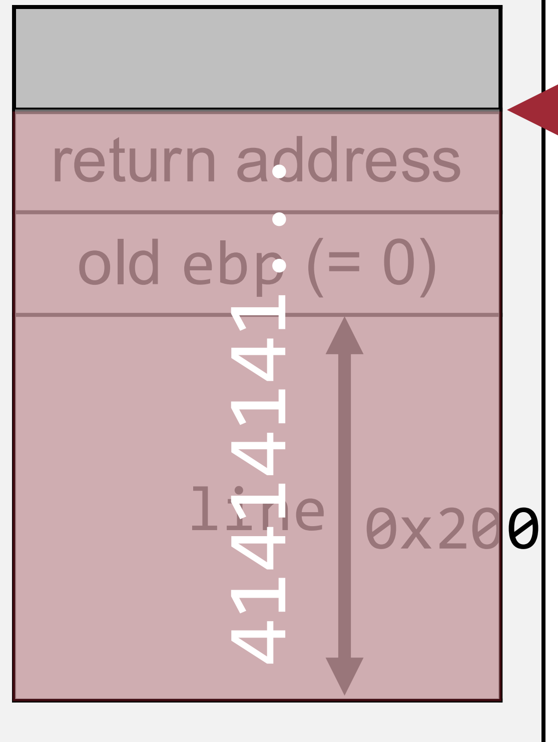
```
rip: 0x40114c
rbp: 0x41414141
rsp: 0xbffefff8
rax: 0x00000000
rdi: 0xbffefdf0
```

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

pop rip

Analyzing the Vulnerability



0xbfff0000

Virtual memory

Control flow hijacked!

rip:	0x41414141
rbp:	0x41414141
rsp:	0xbfff0000
rax:	0x00000000
rdi:	0xbffefdf0

Execution context

```
401126 <main>:
  401126: push rbp
  401127: mov  rbp,rsp
  40112a: sub  rsp,0x200
  401131: lea  rax,[rbp-0x200]
  401138: mov  rdi,rax
  40113b: mov  eax,0x0
  401140: call 401030 <gets@plt>
  401145: mov  eax,0x0
  40114a: leave
  40114b: ret
```

pop rip

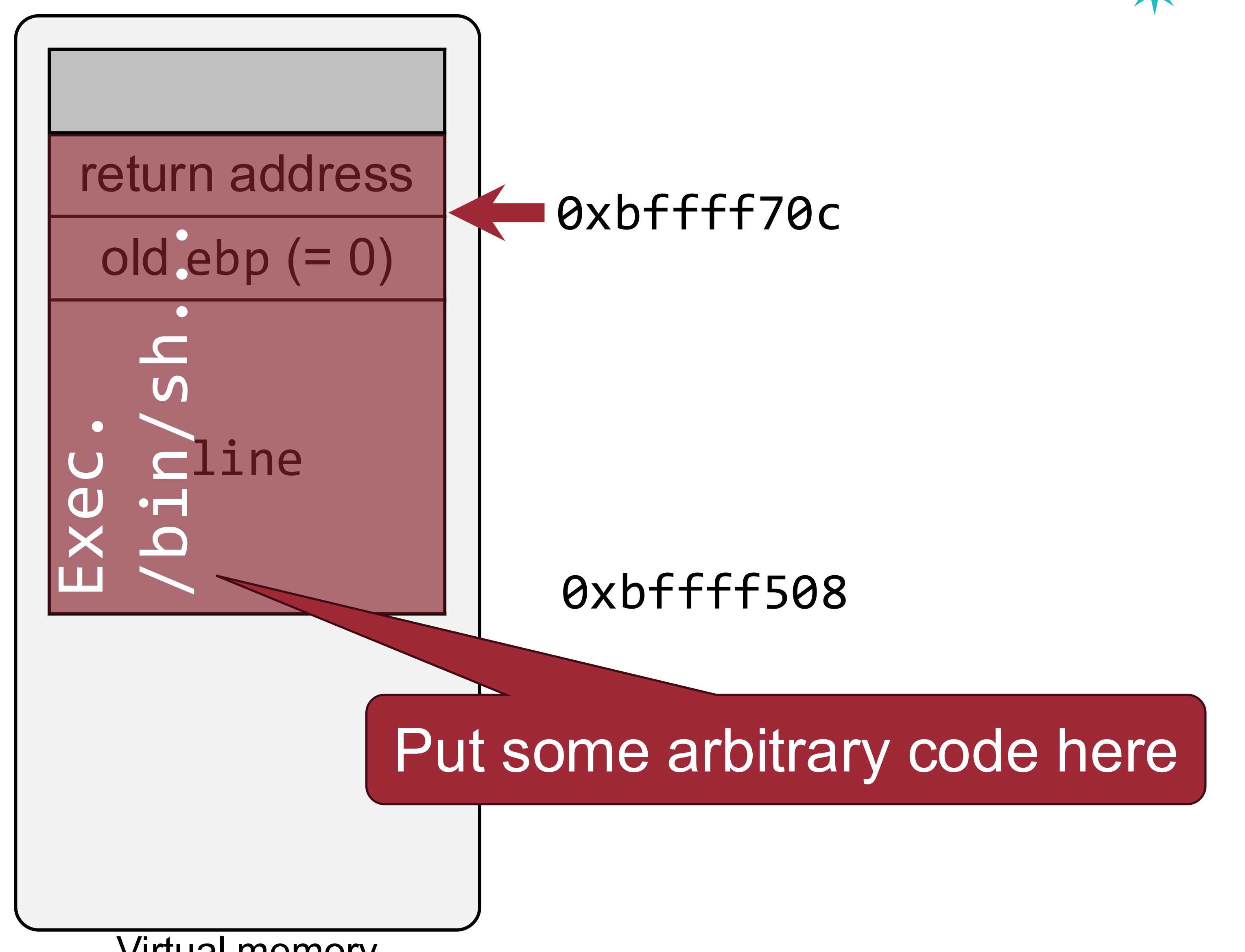
So Far …

- 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!

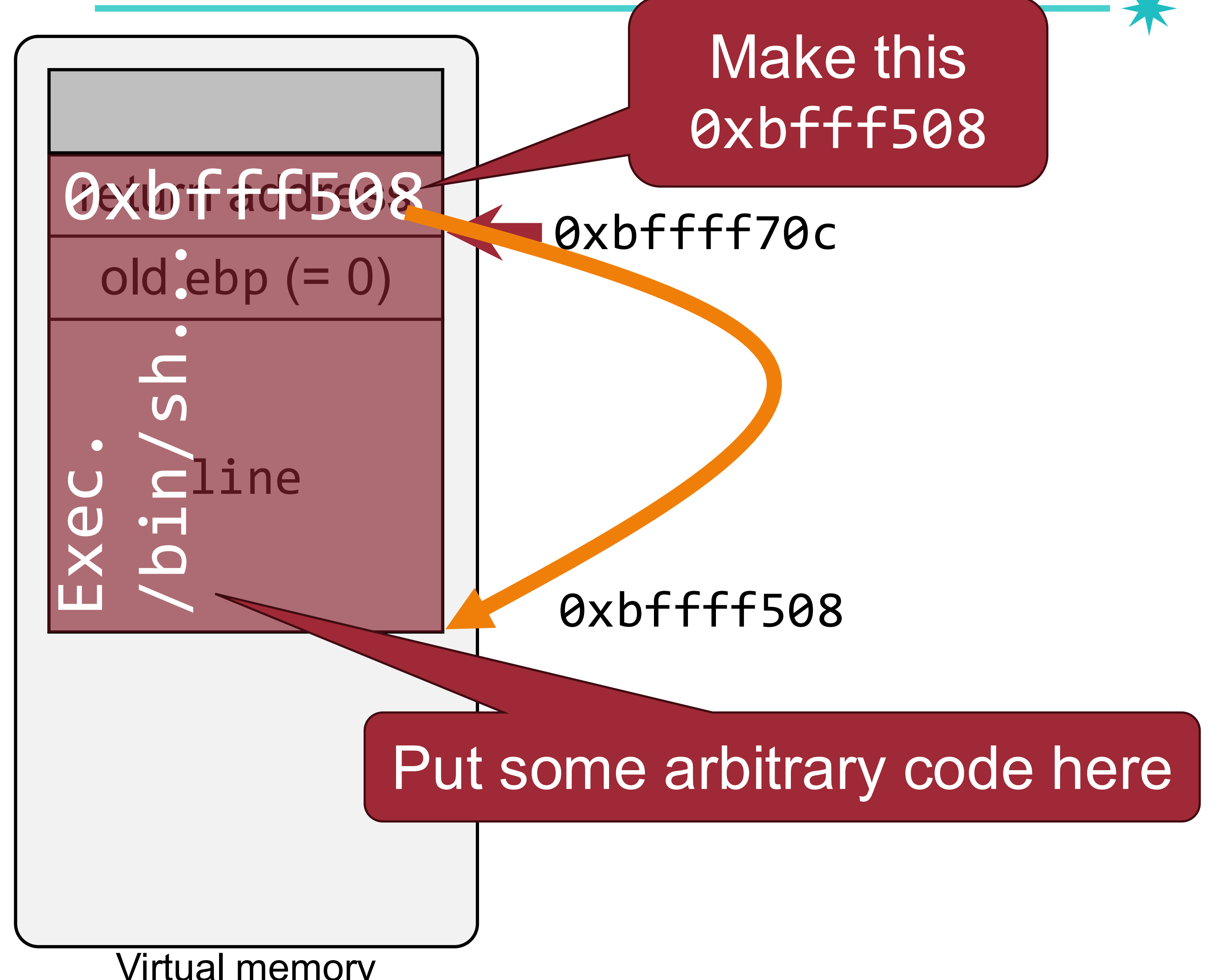
(FYI) Return-to-Stack Attack

return address
old ebp (= 0)
41414141...
line
0xbffff70c
0xbffff508
Put some arbitrary code here
Virtual memory

(FYI) Return-to-Stack Attack



(FYI) Return-to-Stack Attack



(FYI) 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)

How to Defend against Memory Corruption Bugs?

Mitigation #1: Canary

argv
argc
return address
old ebp
Canary value
buf
0xbffff508

Check value before executing return!

Mitigation #2: NX (No eXcute)

Corrupted memory

Hijacked control flow

Attacker's code (Shellcode)

Make this region non-executable! (e.g., stack should be non-executable)

HW2 Tutorial

Let’s look into the handbook!

GDB Usage

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

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

- Breakpoint setting:

```
(gdb) b *<address>
(gdb) i b # See the breakpoint information
```

- Run:

```
(gdb) r
```

- Step:

```
(gdb) si # Go to next instruction, diving into function
(gdb) ni # Go to next instruction but don't dive into function
(gdb) c # 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://websec-lab.github.io/courses/2026s-cse251/metarials/gdb_reference.pdf

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

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

Question?