

Buffer Overflow

Attacker Goal

- Take over target machine (such as a web server)
 - Execute arbitrary (*bad*) code on target by altering application control flow
- Examples:
 - Buffer overflows ← Today
 - Format string vulnerability
 - Other hijacking attacks (e.g., Integer overflow)

Buffer Overflows

- Result from mistakes in memory management when writing code
 - very common *coding flaws* because C functions are exposed to memory management
 - Common even from experienced programmers!
- They often happen in programs written in C/C++
 - Why?
 - Why not in programs written with other languages such as Java or Go?

Buffer overflows are common in languages/runtimes that let programmers manage the memory

Buffer Overflows

- One of the most used attack techniques
- From attacker perspective:
 - Pros
 - very effective: attack code runs with privileges of exploited process
 - can be exploited locally and remotely
 - Cons
 - Architecture-dependent: inject bytecode
 - OS-dependent: use of system calls
 - guesswork involved (correct addresses)

History: Morris Worm

- Released in 1988 by Robert Morris
 - Grad student at Cornell
 - First felony conviction in the US under cybersecurity law
 - Now a professor at MIT
- Unintentional harm:
 - Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- **Due to a coding error**, it created new copies as fast as it could and overloaded infected machines
- \$10-100M worth of damage



History: Morris Worm and Buffer Overflow

- One of the propagation techniques was a **buffer overflow attack** against a vulnerable version of `fingerd` on VAX systems
 - By sending special string to finger daemon, worm caused it to execute code creating a new worm copy

```
> char buffer[512];  
> gets(buffer);
```

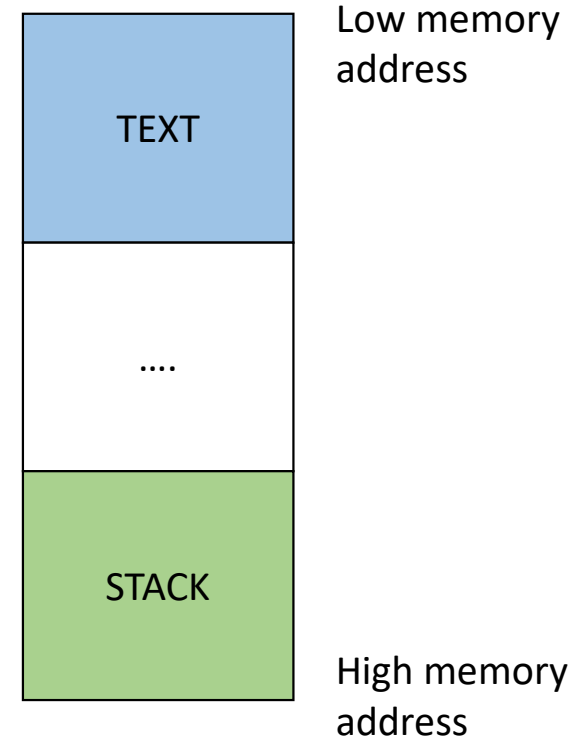
Recent Incidents

- WhatsApp
- “...the phone starts revealing its encrypted content, mirrored on a computer screen halfway across the world. It then transmits back the most intimate details such as private messages and location, and even turns on the camera and microphone to live-stream meetings.”
- The vulnerability was reported as a buffer overflow bug.
- Vulnerabilities reported as “memory corruption”, or “memory safety” are sometimes also buffer overflow bugs

<https://www.ft.com/content/7f2f39b2-733e-11e9-bf5c-6eeb837566c5>

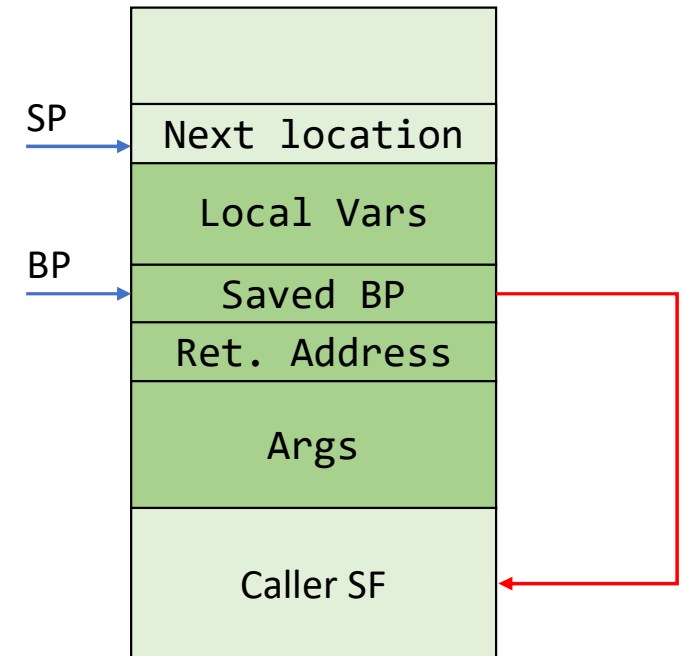
Recall: Process Memory Organization

- A process is divided into four regions.
- For now, the two important ones are:
- Text segment
 - Or code segment
 - Read and execute-only, cannot write
- Stack
 - Maintains state of caller/callee of functions
 - Used for storing:
 - Local statically allocated variables (int, char, char[5], ...)
 - Function parameters
 - Return value (pointer to next instruction)
 - And?



Overflow Attacks

- Stack overflow
 - overflow function return address
 - overflow function base pointer
- Heap overflow (overflow dynamically allocated memory region)
- Overflow virtual function pointers



We will focus on Stack overflow for now

Stack Region: Function Call

```
int func(int a, int b) {  
    int i = 3;  
    return (a+b)*i;  
}
```

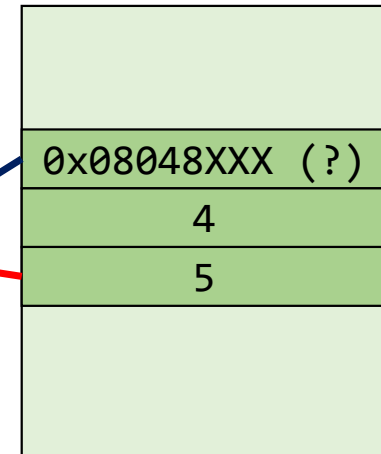
```
int main() {  
    int result = 0;  
    result = func(4, 5);  
    printf("%d\n", result);  
}
```

3
Saved BP
Ret. Address
4
5

func() Stack Frame

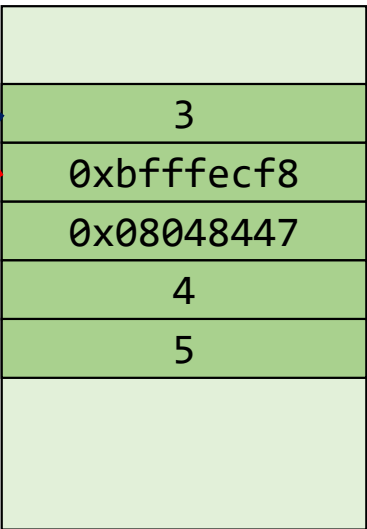
A Closer Look

```
gdb-peda$ disas main
Dump of assembler code for function main:
0x08048426 <+0>:    lea    ecx,[esp+0x4]
0x0804842a <+4>:    and    esp,0xffffffff0
0x0804842d <+7>:    push  DWORD PTR [ecx-0x4]
0x08048430 <+10>:   push  ebp
0x08048431 <+11>:   mov    ebp,esp
0x08048433 <+13>:   push  ecx
0x08048434 <+14>:   sub    esp,0x14
0x08048437 <+17>:   mov    DWORD PTR [ebp-0xc],0x0
0x0804843e <+24>:   push  0x5
0x08048440 <+26>:   push  0x4
0x08048442 <+28>:   call  0x804840b <func>
0x08048447 <+33>:   add    esp,0x8
```



```
gdb-peda$ break func
Breakpoint 1 at 0x8048411
gdb-peda$ run
Starting program: /home/seed/CMPT_479_980/stack_ex
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/l

[-----registers-----]
EAX: 0xb7f1ddbc --> 0xbfffedac --> 0xbfffebbe ("XDG_VT
EBX: 0x0
ECX: 0xbfffed10 --> 0x1
EDX: 0xbfffed34 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffecf8 --> 0xbfffecf8 --> 0x0
ESP: 0xbfffecc0 --> 0xb7fd44e8 --> 0xb7fd3aa8 --> 0xb7
    ebx)
EIP: 0x8048411 (<func+6>:      mov     DWORD PTR [ebp-
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTE
[-----code-----]
0x804840b <func>:      push   ebp
0x804840c <func+1>:   mov    ebp,esp
0x804840e <func+3>:   sub    esp,0x10
=> 0x8048411 <func+6>:  mov    DWORD PTR [ebp-0x4],0x3
0x8048418 <func+13>:  mov    edx,DWORD PTR [ebp+0x8]
0x804841b <func+16>:  mov    eax,DWORD PTR [ebp+0xc]
0x804841e <func+19>:  add    eax,edx
0x8048420 <func+21>:  imul  eax,DWORD PTR [ebp-0x4]
```



main() SF

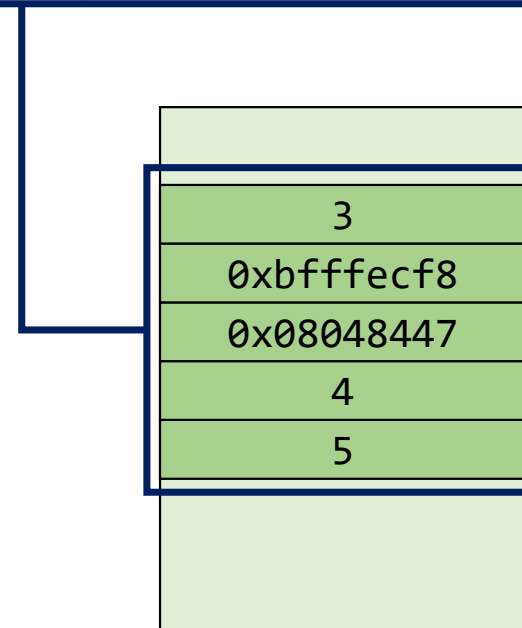
The **func** Stack Frame

```
gdb-peda$ x/12wx $ebp-16
0xbfffecc0: 0xb7fd44e8 0xb7fd445c 0xb7fd27bc 0x00000003
0xbfffecd0: 0xbfffecf8 0x08048447 0x00000004 0x00000005
```

x/<fmt> <address>

<fmt>: is a repeat count followed by a format letter and a size letter.

- **Format letters** are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left).
- **Size letters** are b(byte), h(halfword), w(word), g(giant, 8 bytes).



Let's Take Control of a Program

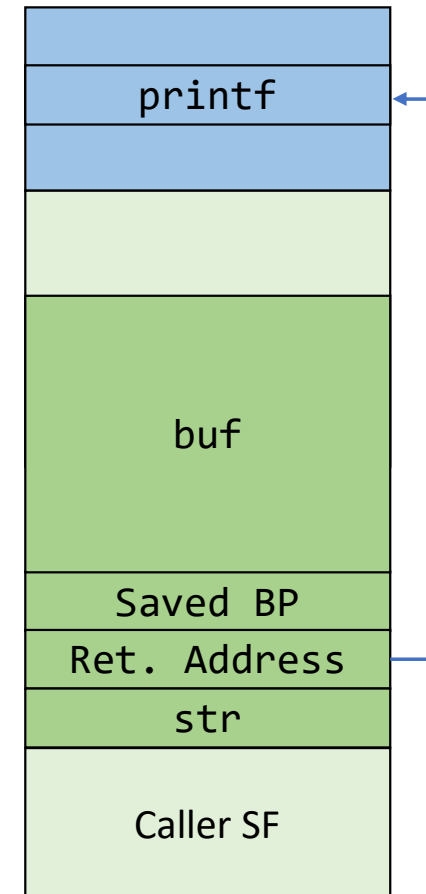
- Code (or parameters) get injected because
 - program accepts more input than there is space allocated
- In particular, an array (or buffer) has not enough space
 - especially easy with C strings (character arrays)
 - plenty of vulnerable library functions
`strcpy`, `strcat`, `gets`, `fgets`, `sprintf` ...
- Input spills to adjacent regions and modifies, two possibilities:
 1. “normally”, this just crashes the program (e.g., SIGSEGV)
 2. code pointer or application data
 - all the possibilities that we have enumerated before

Example: Simple Web Server

```
void serve(char *str) {  
    char buf[100];  
    strcpy(buf, str);  
}  
int main(int argc, char* argv[]) {  
    serve(argv[1]);  
    printf("Bye\n");  
}
```

Allocate 100 bytes on the stack

Copy str to local buffer

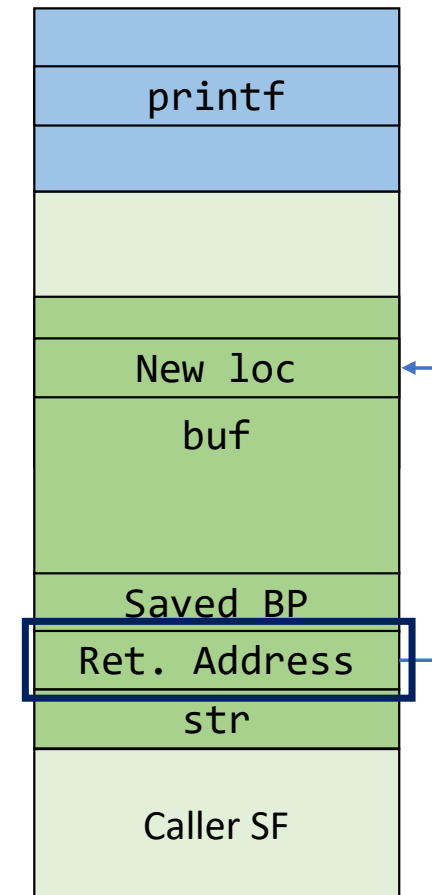


What if **buf** exceeds the 100 bytes?

```
void serve(char *str) {  
    char buf[100];  
    strcpy(buf, str);  
}  
  
int main(int argc, char* argv[]) {  
    serve(argv[1]);  
    printf("Bye\n");  
}
```

strcpy does NOT check whether the string at *str contains fewer than 100 bytes!

- If a string longer than 100 bytes is copied into buffer, it will overwrite adjacent stack locations.



Example: Let's Crash the Server

```
$ ./server_vuln hello  
$ Bye
```

Input length < 100 bytes

```
$ ./server_vuln
```

Input length > 100 bytes

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

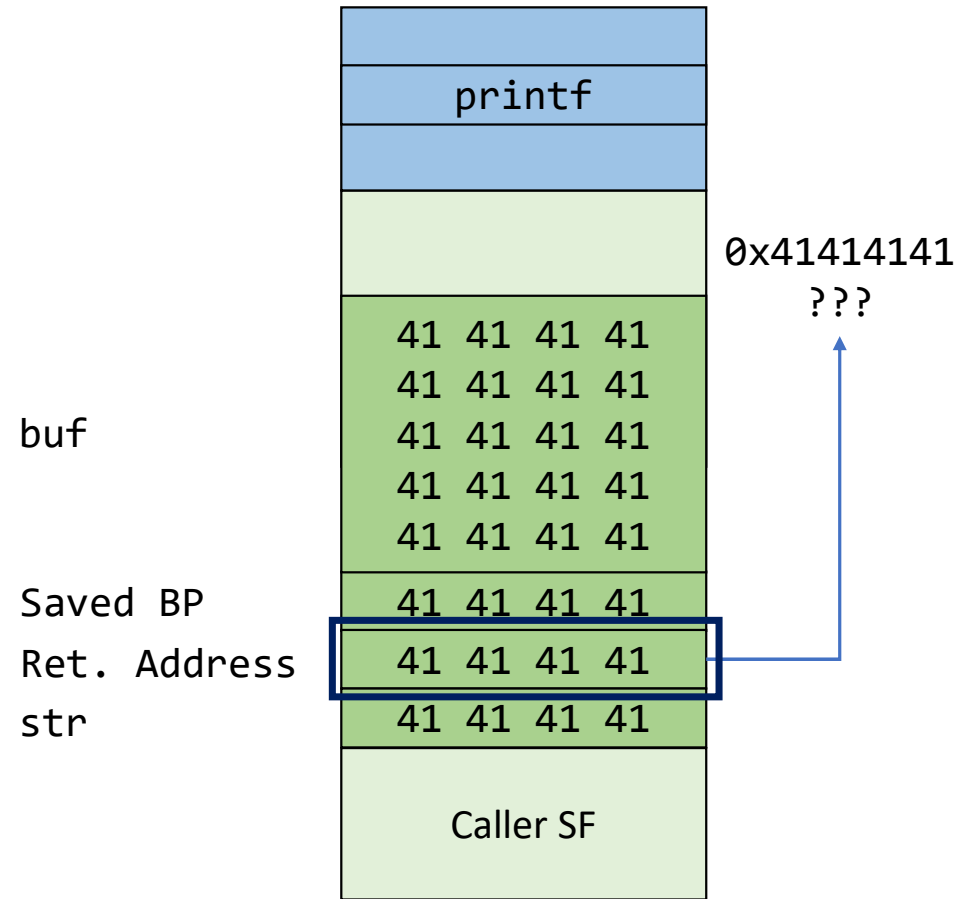
Segmentation fault

```
$
```

Why?

What happened?

```
[-----registers-----
EAX: 0xbfffeb5c ('A' <repeats 200 times>...)
EBX: 0x0
ECX: 0xbfffefb0 ("AAAAAAAAAA")
EDX: 0xbfffec5e ("AAAAAAAAAA")
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x41414141 ('AAAA')
ESP: 0xbfffebd0 ('A' <repeats 152 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN t
[-----code-----
Invalid $PC address: 0x41414141
[-----stack-----
0000| 0xbfffebd0 ('A' <repeats 152 times>)
0004| 0xbfffebd4 ('A' <repeats 148 times>)
0008| 0xbfffebd8 ('A' <repeats 144 times>)
0012| 0xbfffebd0 ('A' <repeats 140 times>)
0016| 0xbfffebe0 ('A' <repeats 136 times>)
0020| 0xbfffebe4 ('A' <repeats 132 times>)
0024| 0xbfffebe8 ('A' <repeats 128 times>)
0028| 0xbfffebec ('A' <repeats 124 times>)
[-----
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
```



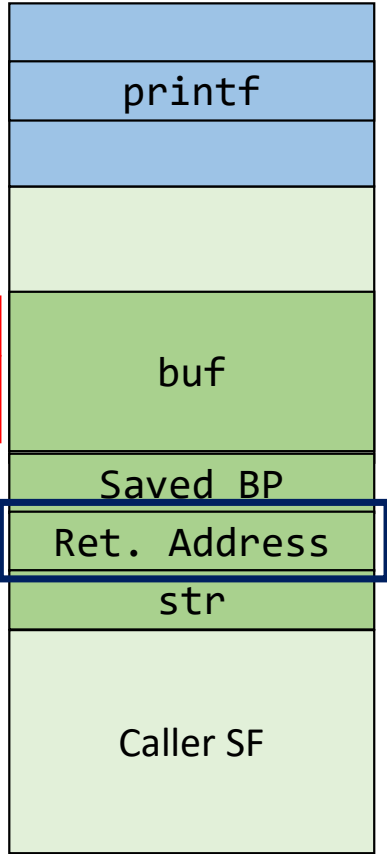
What if **buf** contains bad code?



```
void serve(char *str) {  
    char buf[100];  
    strcpy(buf, str);  
}  
  
int main(int argc, char* argv[]) {  
    serve(argv[1]);  
    printf("Bye\n");  
}
```

strcpy does NOT check whether the string at *str contains fewer than 100 bytes!

New loc.
execve("sh")



- When function returns, code in the buffer will be executed, giving attacker a shell
 - **Root shell** if the victim program is setuid root

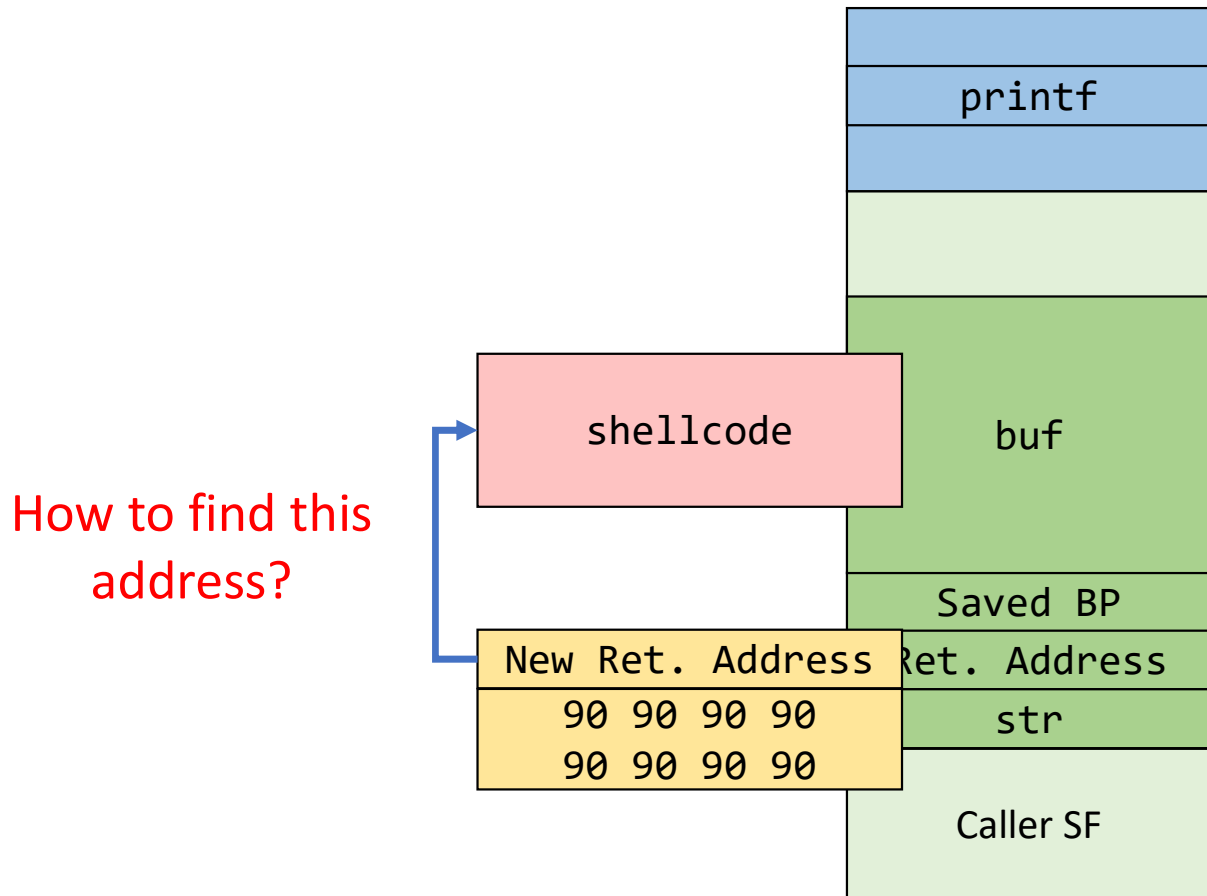
Problem: Choosing Where to Jump

- **Address inside a buffer of which the attacker controls the content**
 - works for remote attacks
 - the attacker need to know the address of the buffer, the memory page containing the buffer must be executable
- **Address of a function inside the program**
 - works for remote attacks, does not require an executable stack
 - need to find the right code, one or more fake frames must be put on the stack
- **Address of a environment variable**
 - easy to implement, works with tiny buffers
 - only for local exploits, some programs clean the environment, the stack must be executable

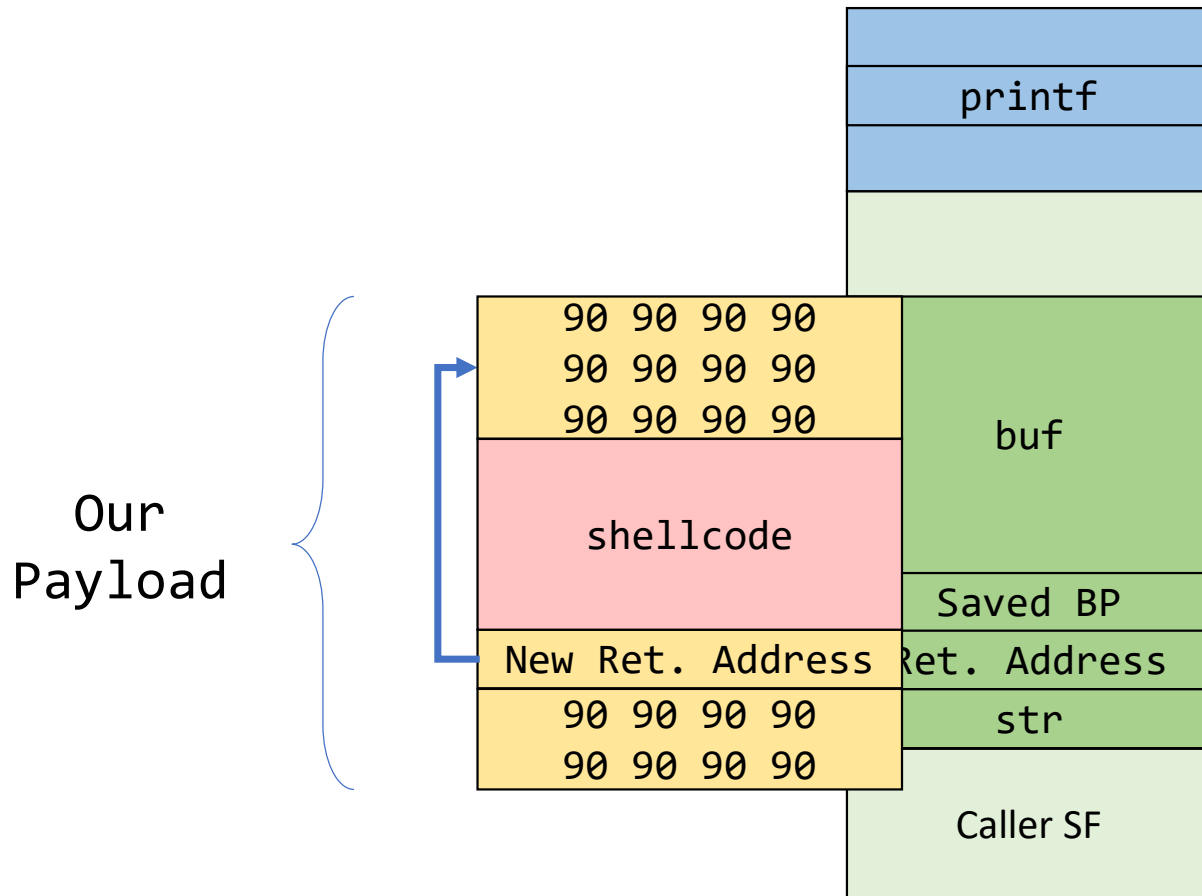
Jumping into the Buffer

- The buffer that we are overflowing is usually a good place to put the code (bytecode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
 - The address must be precise: jumping one byte before or after could make the application crash
 - NOP sled (later) partly weakens this requirement
 - On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine

Scenario 1 – Jump to Shellcode



Scenario 1 – Jump to Shellcode

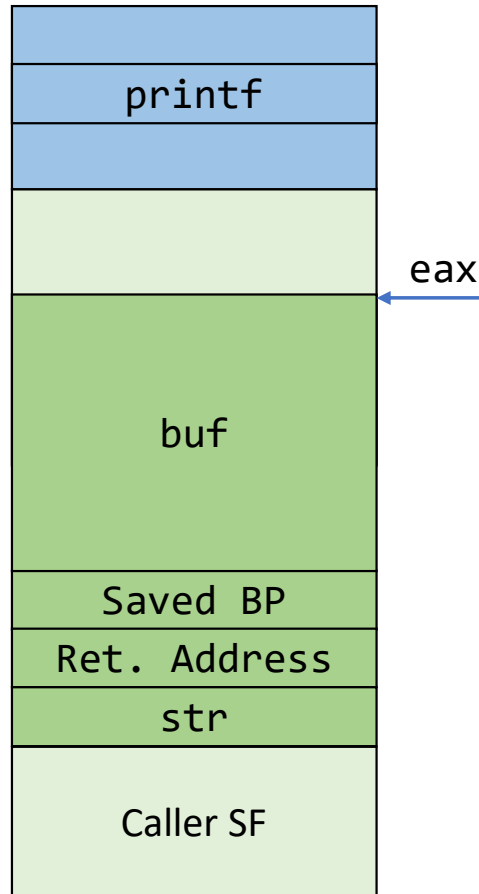


We don't need the **exact** address

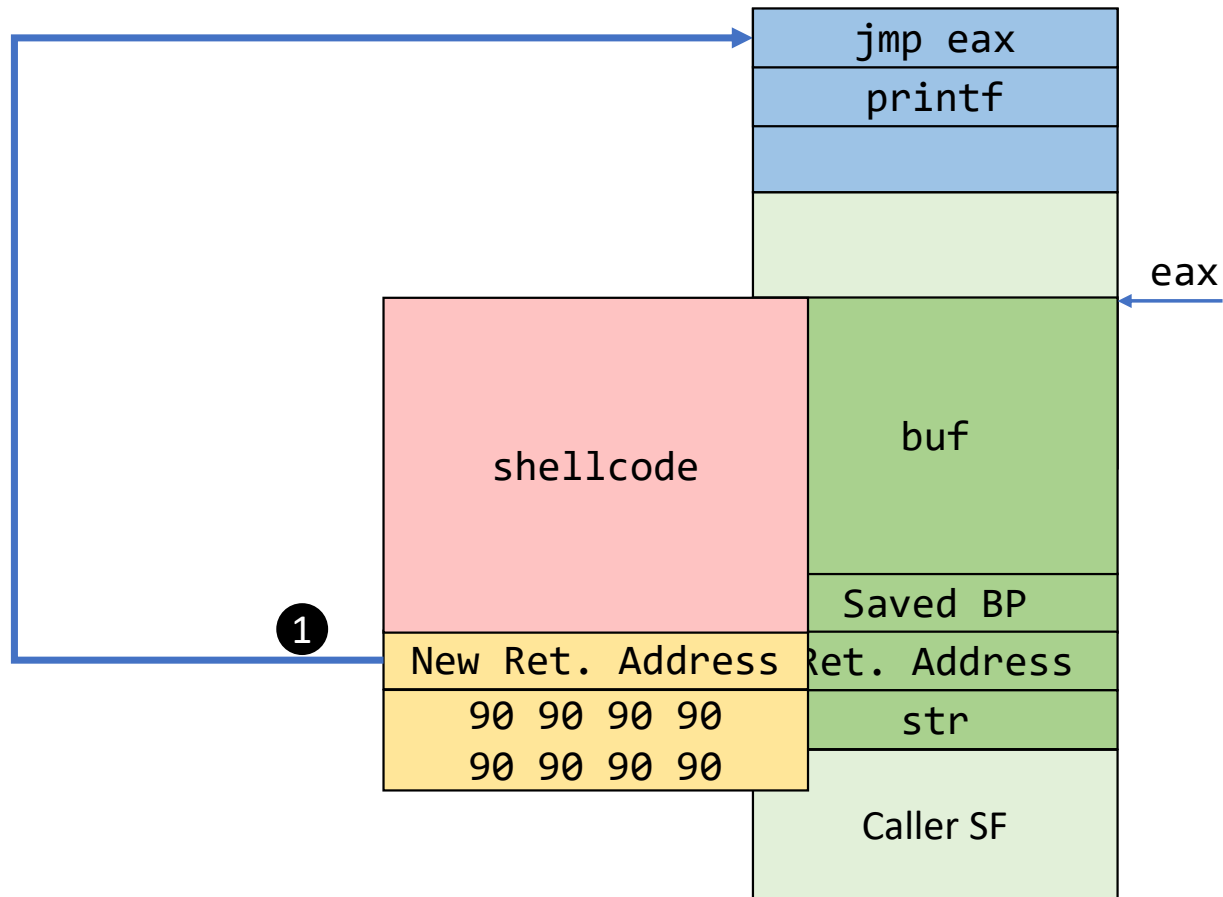
We need to add **dummy** instructions that will lead to our shellcode!

Our
Payload

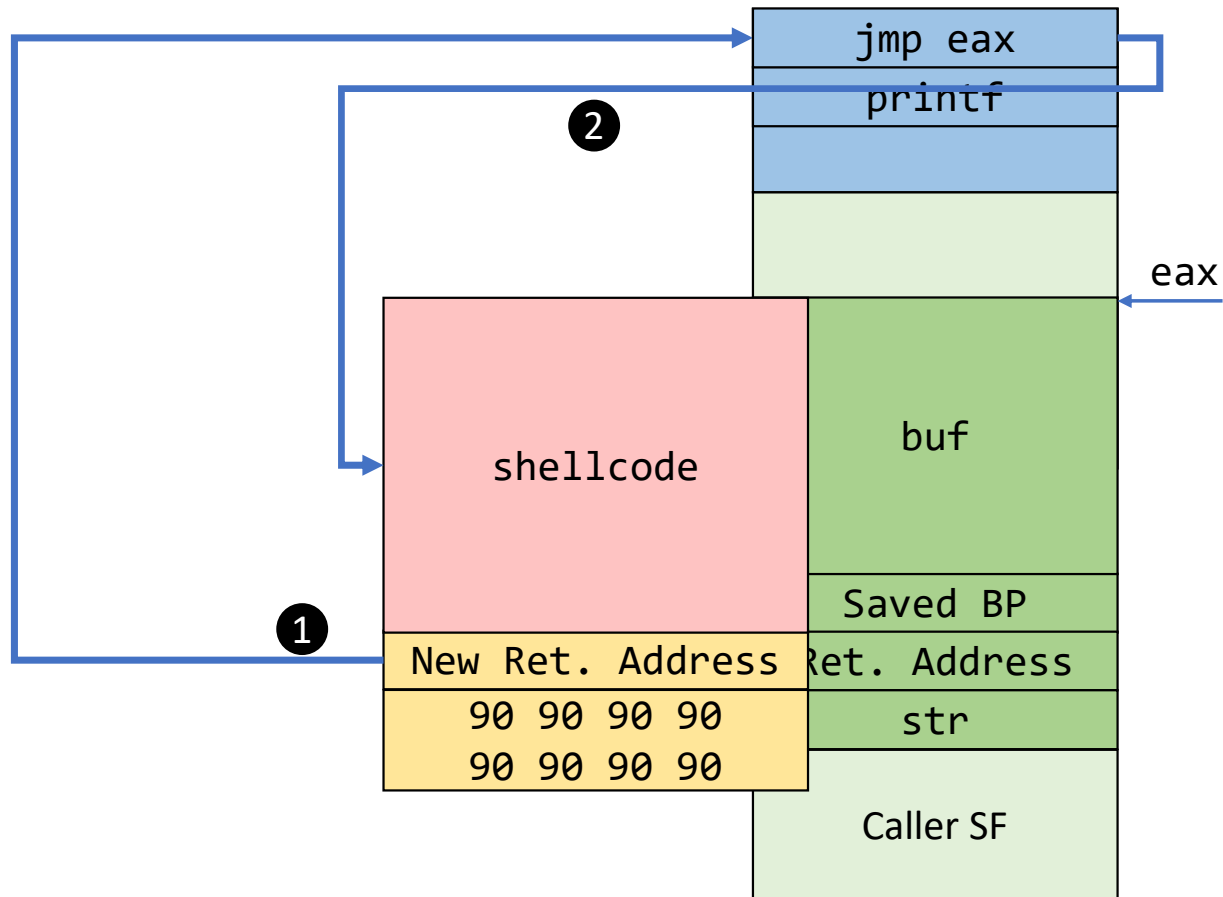
Scenario 2 – JMP to Register (ret2reg)



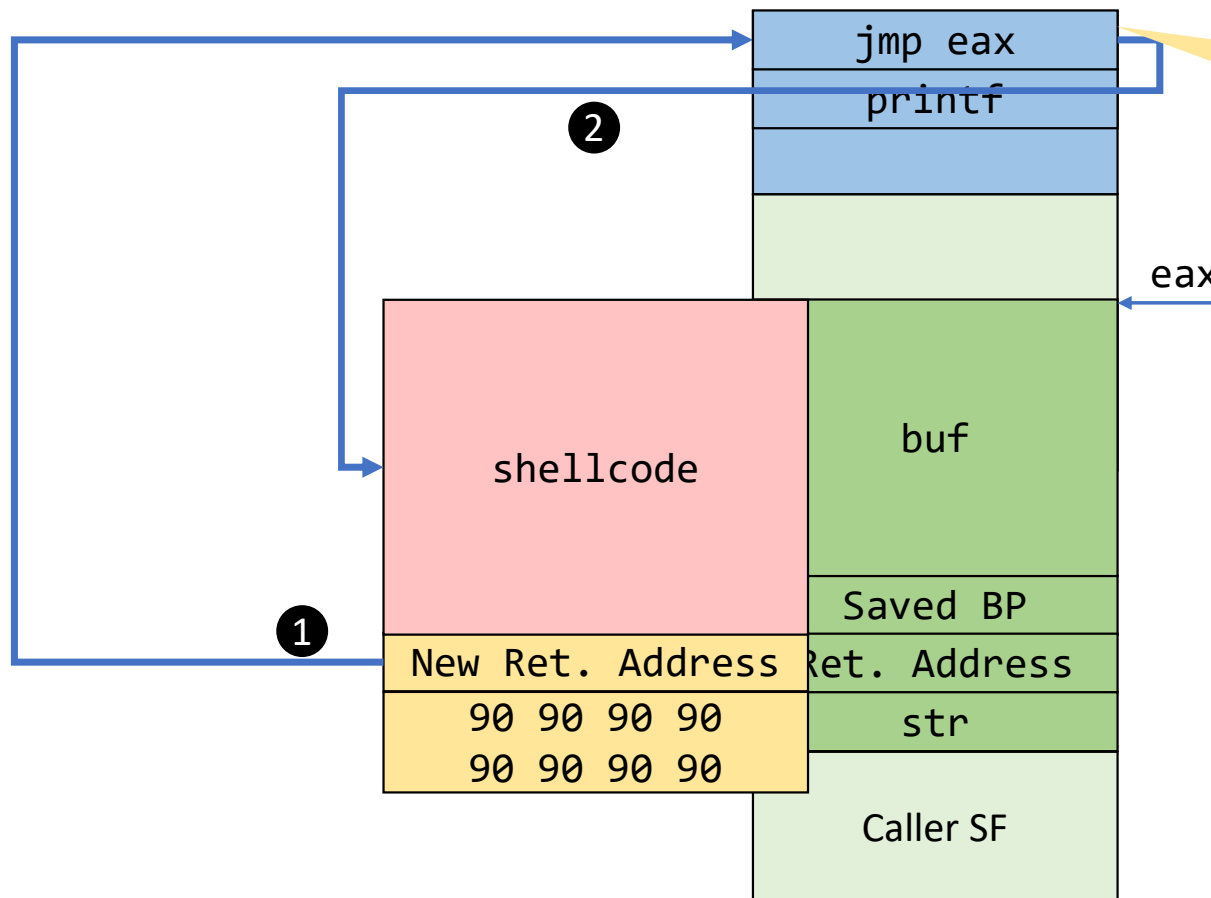
Scenario 2 – JMP to Register (ret2reg)



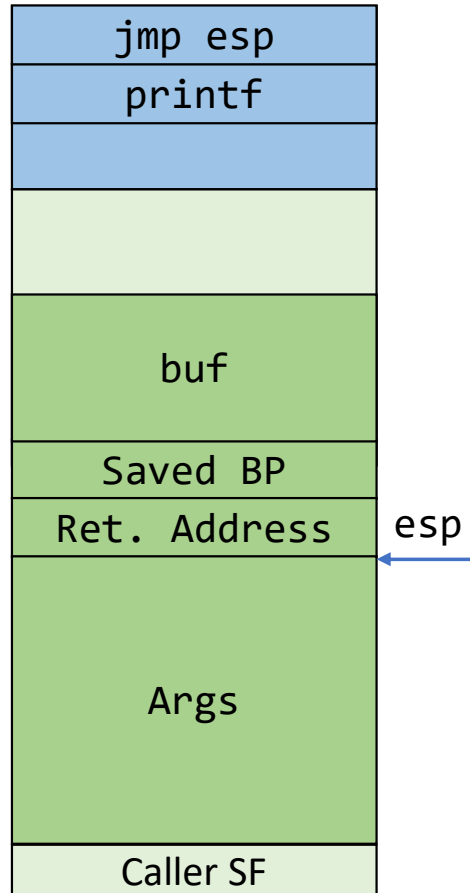
Scenario 2 – JMP to Register (ret2reg)



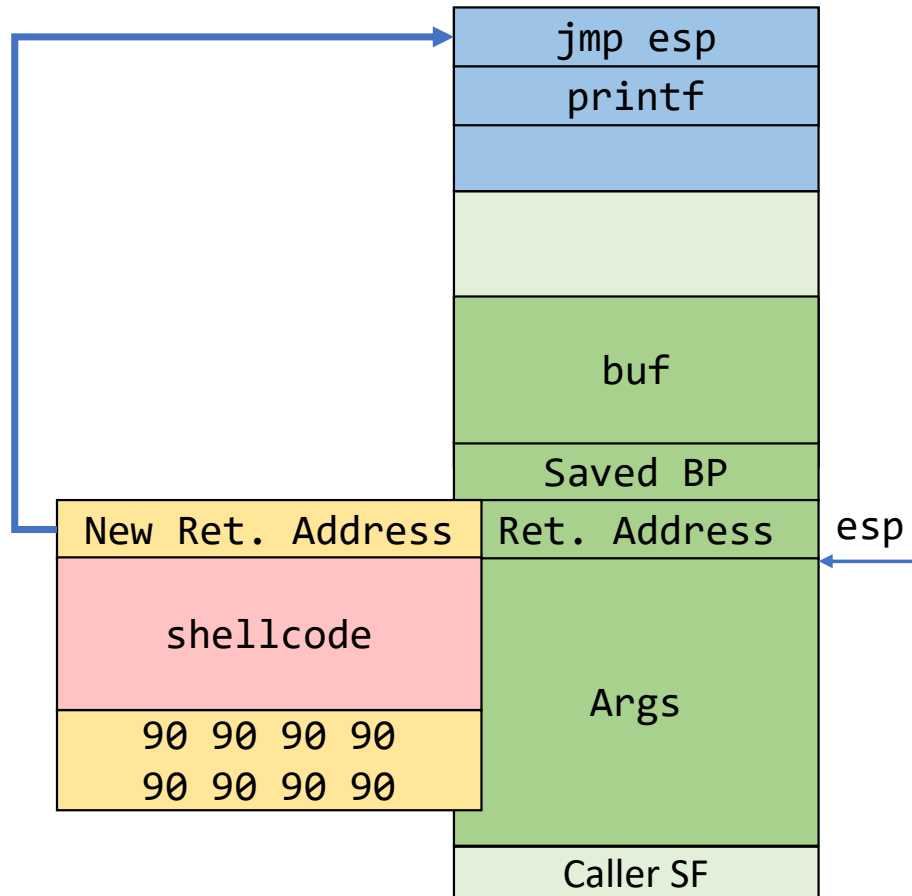
Scenario 2 – JMP to Register (ret2reg)



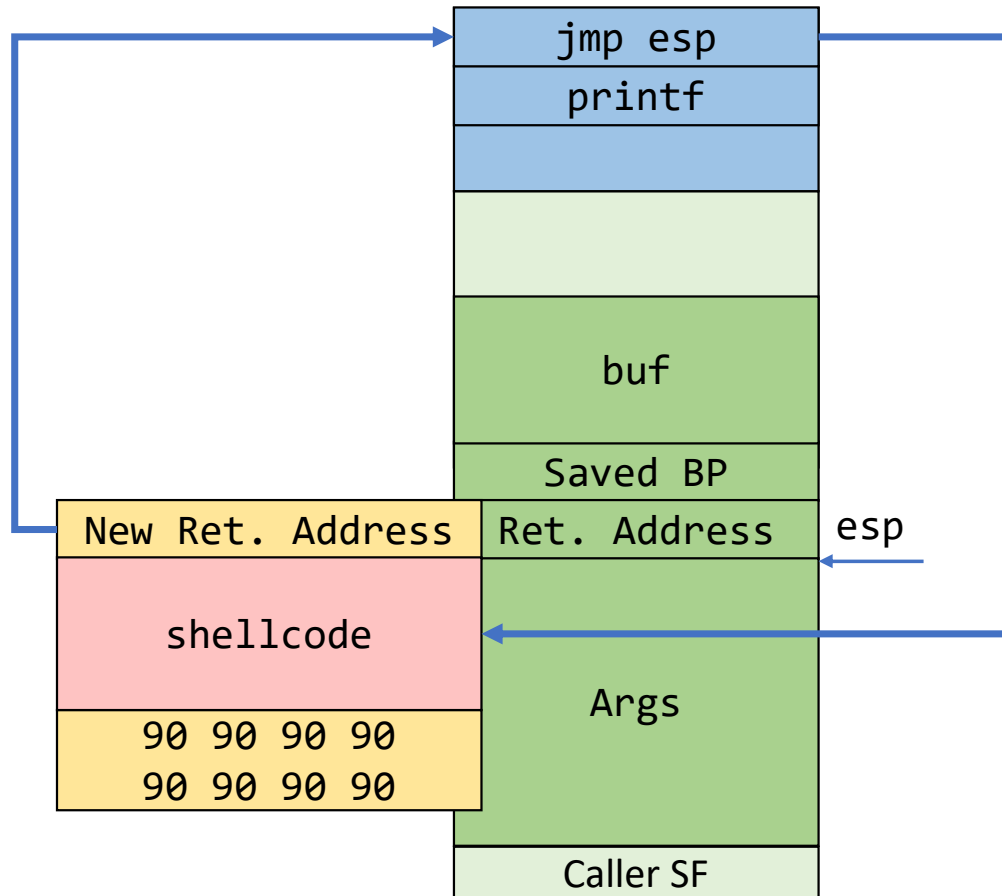
Scenario 3 – JMP to ESP



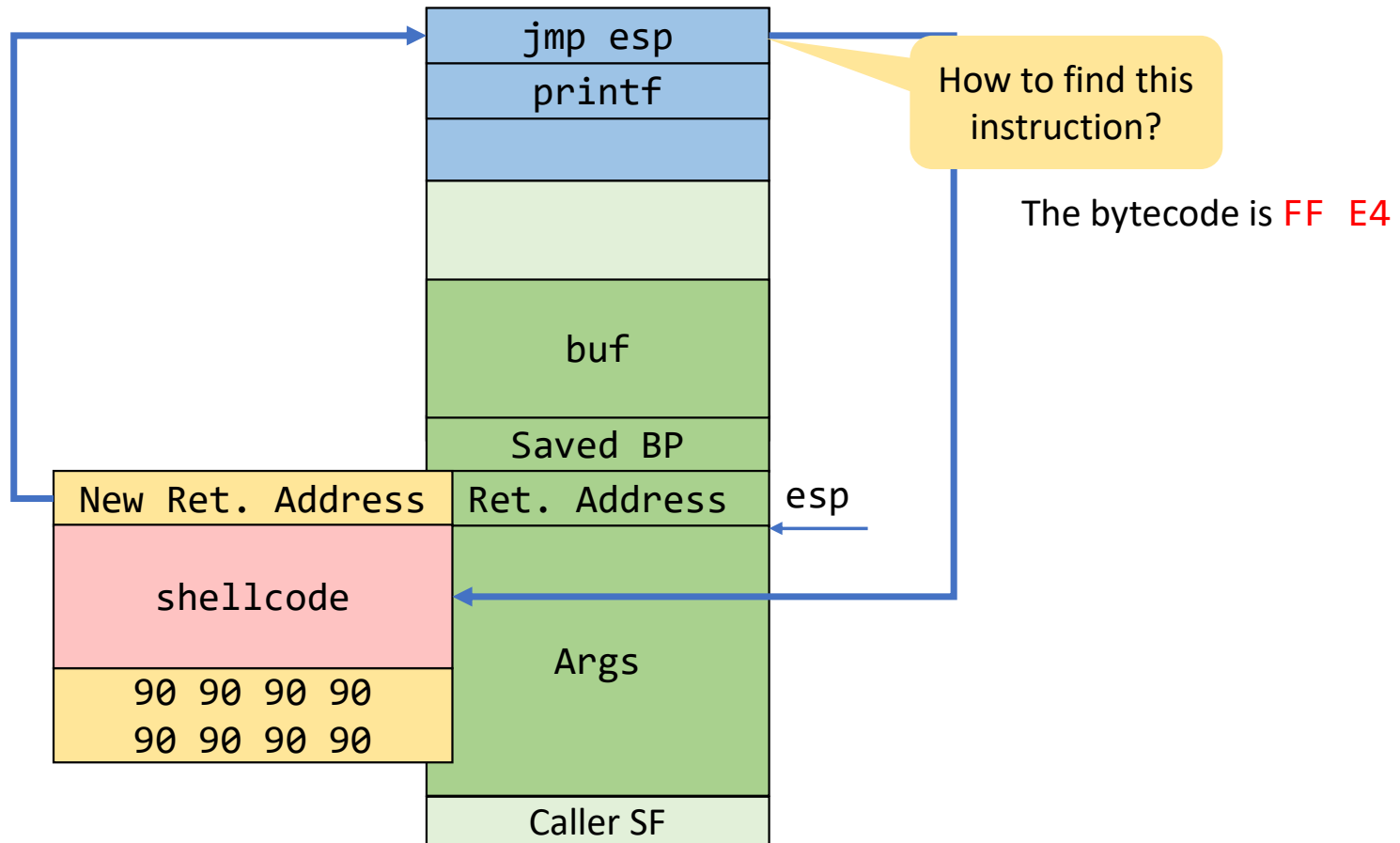
Scenario 3 – JMP to ESP



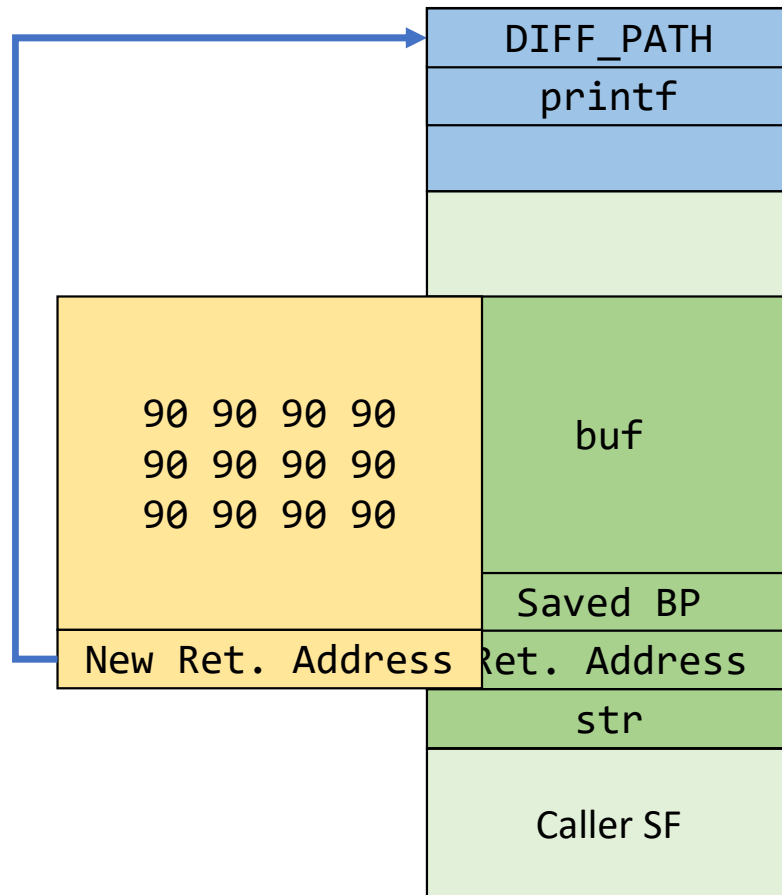
Scenario 3 – JMP to ESP



Scenario 3 – JMP to ESP



Scenario 4 – JMP to different code path



Recap: The NOP Sled (0x90)

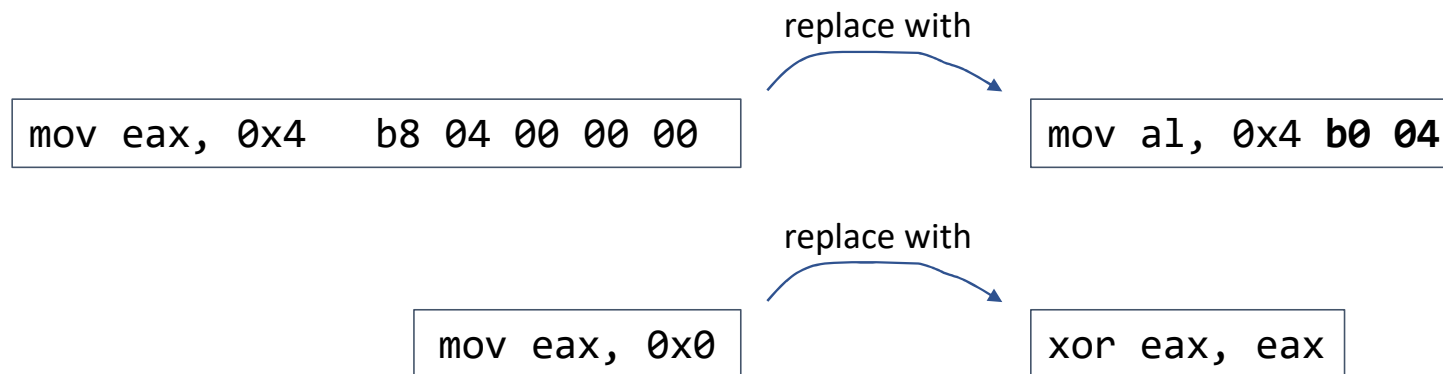
- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever the program jump into it always
 1. finds a valid instruction
 2. reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of no operation (NOP) instructions
 - single byte instruction (0x90) that does not do anything
- It mitigates the problem of finding the **exact address** to the buffer by increasing the size of the target area

Recap: JMP using a register

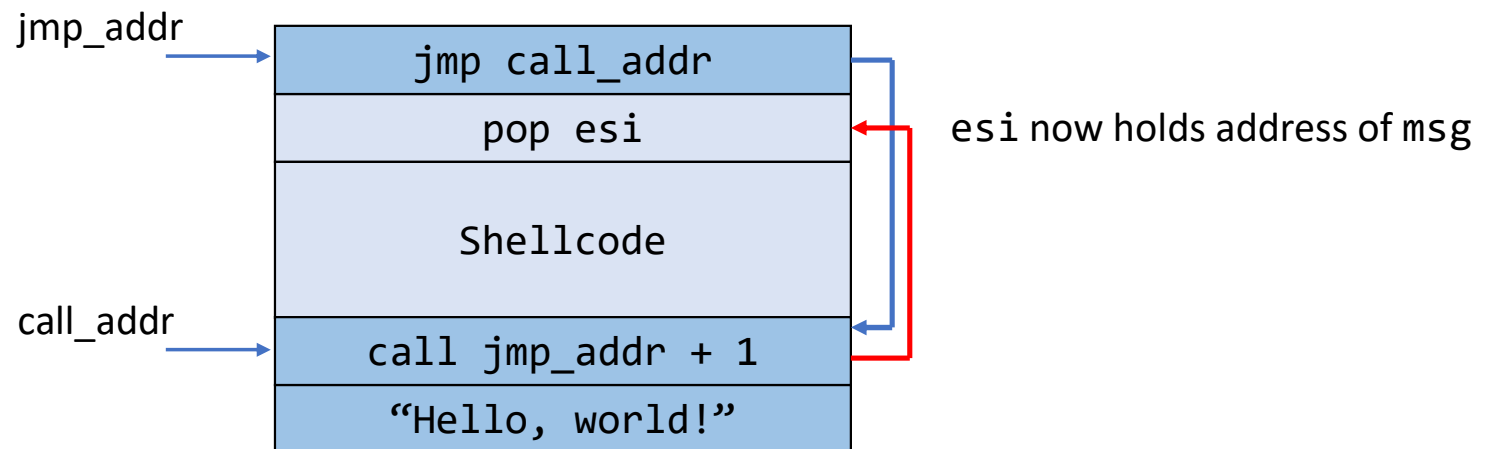
1. Find a register that points to the buffer (or somewhere into it)
 - ESP
 - EAX (return value of a function call)
2. Locate an instruction that jump/call using that register
 - can also be in one of the libraries
 - does not need to be a real instruction (**just the right sequence of bytes**)
 - you can search for a pattern with `gdb find jmp ESP = 0xFF 0xE4`
3. Overwrite the return address with the address of that instruction

Tip 1: Copying Shellcode

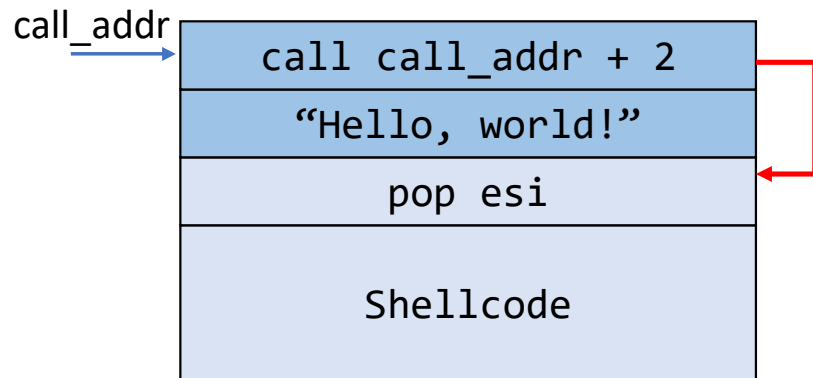
- Shellcode is usually copied into a string buffer
- Problem
 - any null byte would stop copying
 - null bytes must be eliminated from the shellcode!



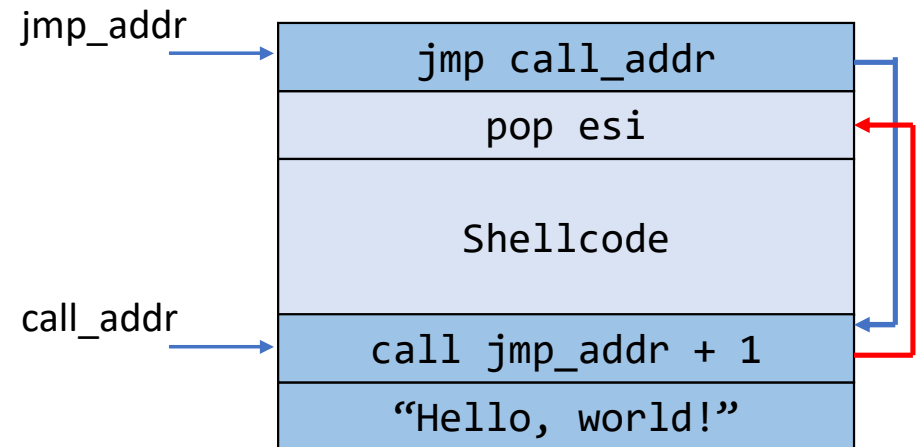
Tip 2: Relative Addressing Technique



Relative Addressing Technique



Why not this one?



Tip 3: Enable Privileges

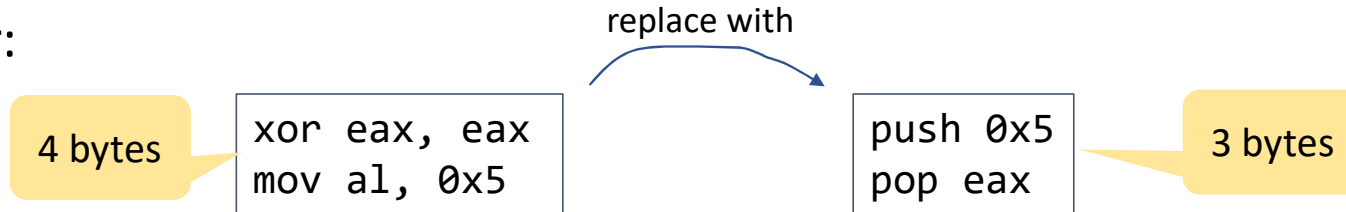
- Problem:
 - exploited program could have temporarily dropped privileges
- Technique:
 - Shellcode has to enable privileges again (using setuid)

Small Buffers

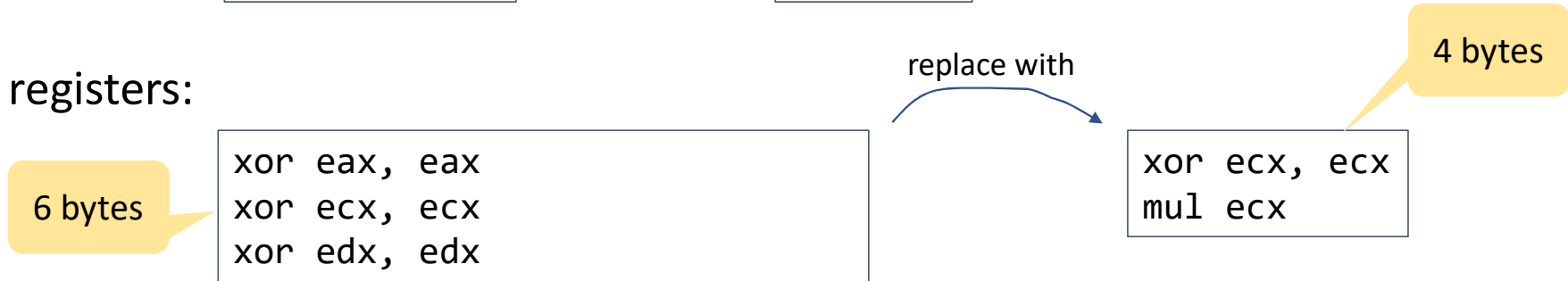
- Buffer can be too small to hold exploit code
- Store exploit code in environment variable
 - environment stored on stack
 - return address has to be redirected to environment variable
- Advantage
 - exploit code can be arbitrary long
- Disadvantage
 - access to environment needed

Tip 4: Every Byte Matters (Examples)

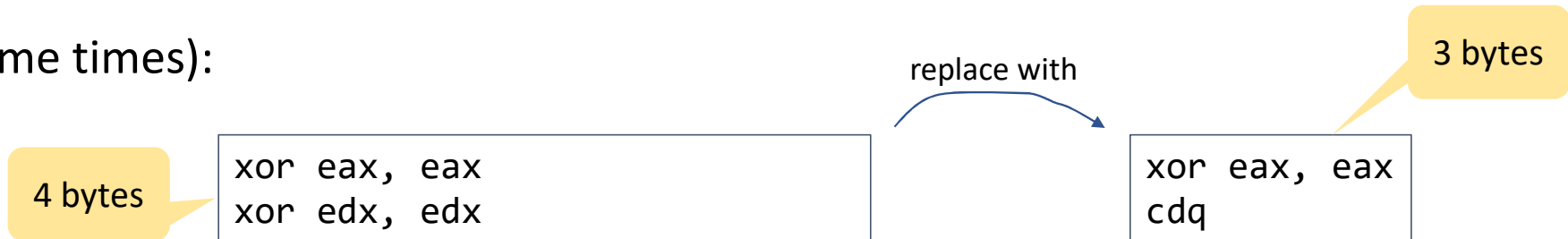
Initialize a register:



Zero out multiple registers:



Zero out edx (some times):



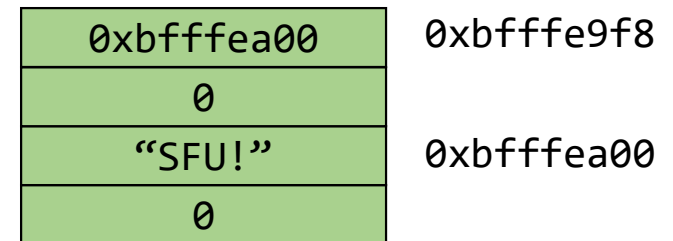
Tip 5: Strings and their addresses

- Instead of jmp-call-pop technique, we can directly push bytes to the stack

```
xor eax, eax
push eax
push 0x21756673 ; little-endian

mov ebx, esp    ; ebx = 0xbfffea00
push eax
push ebx

mov ecx, esp    ; ecx = 0xbfffe9f8
```



Recap: Requirements for Shellcode

- No zero bytes!
- Position-independent code (PIC)
- Doesn't use absolute addresses
- Better: be as small as possible

Questions?

Lab 3

- Shellcode and buffer overflow
- Task 1: Exploit a buffer overflow vulnerability
- Task 2: Defeat a countermeasure in one application
- Task 3: Write shellcode to manipulate a file

Assignment 1: Setup

- Two options:
 1. Provided VM: Ubuntu 16.04 (32-bit)
 - nasm, gcc, gdb, objdump
 - Address space randomization is disabled
 - /bin/sh → /bin/zsh
 2. Use your own VM (should be a 32-bit OS)
- Your code should run **smoothly** in our VM:
 - E.g., we will not fix any runtime/compilation errors