Memory Safety Vulnerabilities

Today: Memory Safety Vulnerabilities

- Buffer overflows
  - Stack smashing
  - Memory-safe code
- Integer memory safety vulnerabilities
- Format string vulnerabilities
- Heap vulnerabilities
- Writing robust exploits

Review: x86 Calling Convention

Review: Registers

- **EIP**: instruction pointer, points to the *next* instruction to be executed
- **EBP**: base pointer, points to top of the current stack frame
- **ESP**: stack pointer, points to lowest item on the stack

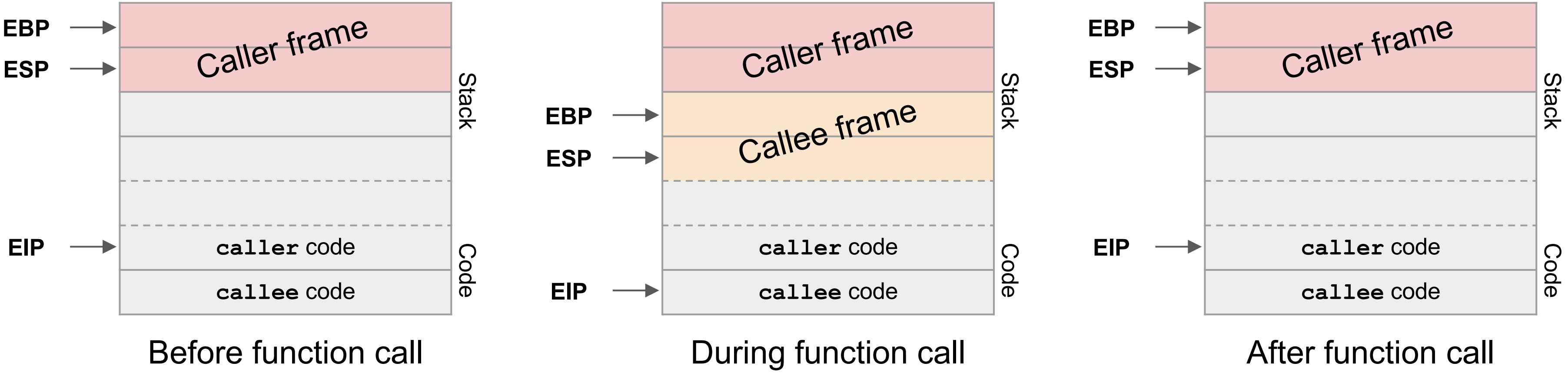
Review: Instructions

- **`push src`**
  - ESP moves one word down
  - Puts the value in **`src`** at the current ESP

- **`pop dst`**
  - Copies the lowest value on the stack (where ESP is pointing) into **`dst`**
  - ESP moves one word up

- **`mov src dst`**
  - Copies the value in **`src`** into **`dst`**

Calling a Function in x86



Steps of an x86 Function Call

caller

1. Push arguments on the stack
2. Push old EIP (RIP) on the stack
3. Move EIP

callee

4. Push old EBP (SFP) on the stack
5. Move EBP
6. Move ESP
7. Execute the function
8. Move ESP
9. Pop (restore) old EBP (SFP)
10. Pop (restore) old EIP (RIP)

caller

11. Remove arguments from stack

x86 Function Call


```c
int callee(int a, int b) {
    int local;
    return 42;
}

void caller(void) {
    callee(1, 2);
}
```

Here is a snippet of C code

Here is the code compiled into x86 assembly

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


The instruction that was just executed is in **red**

The EIP points to the address of the *next* instruction!

```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


Here is a diagram of the stack. Remember, each row represents 4 bytes (32 bits).

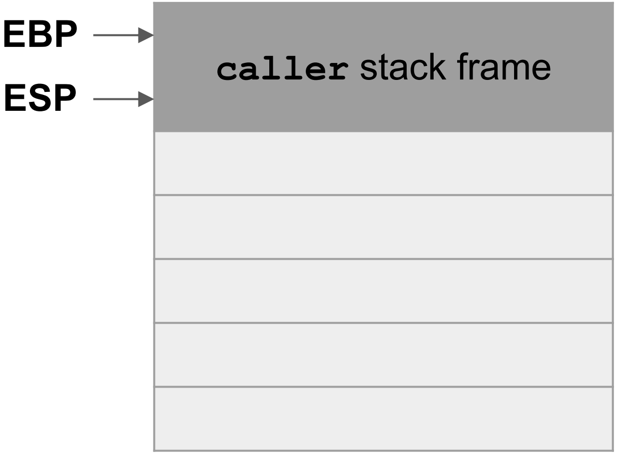
```
caller:
    ...
EIP →   push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


- The EBP and ESP registers point to the top and bottom of the current stack frame.




```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

x86 Function Call

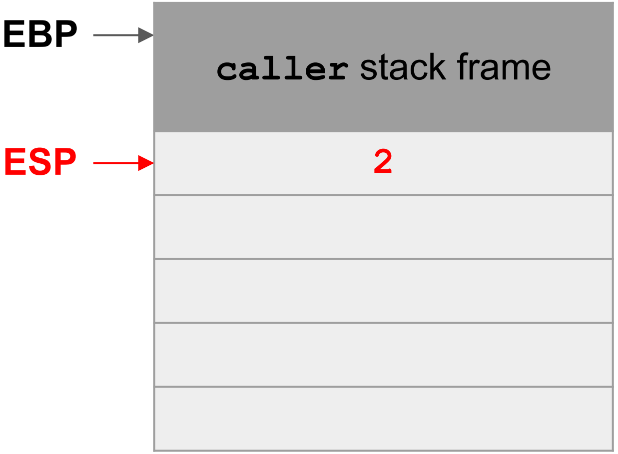
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

1. Push arguments on the stack

- The **`push`** instruction decrements the ESP to make space on the stack
- Arguments are pushed in reverse order




```
caller:
        ...
        push $2
EIP →   push $1
        call callee
        add $8, %esp
        ...

callee:
        push %ebp
        mov %esp, %ebp
        sub $4, %esp

        mov $42, %eax

        mov %ebp, %esp
        pop %ebp
        ret
```

x86 Function Call

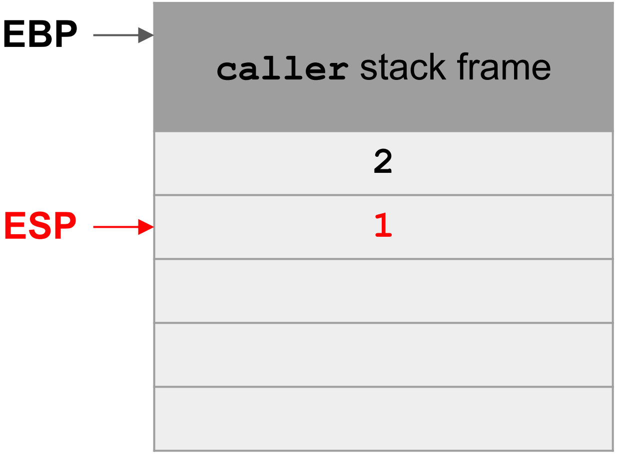
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

1. Push arguments on the stack

- The **`push`** instruction decrements the ESP to make space on the stack
- Arguments are pushed in reverse order




```
caller:
    ...
    push $2
    push $1
EIP → call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

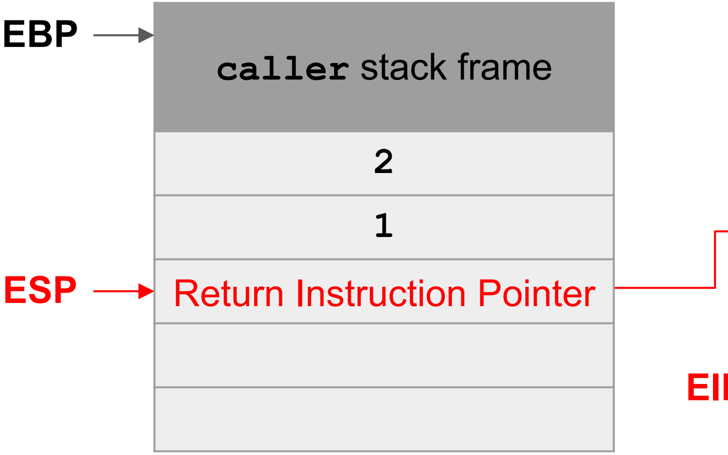
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


2. Push old EIP (RIP) on the stack
3. Move EIP

- The `call` instruction does 2 things
- First, it pushes the current value of EIP (the address of the next instruction in `caller`) on the stack.
- The saved EIP value on the stack is called the RIP (return instruction pointer).
- Second, it changes EIP to point to the instructions of the callee.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

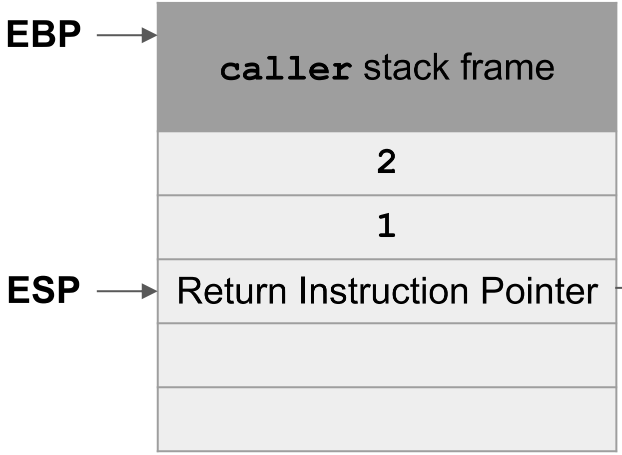
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

- The next 3 steps set up a stack frame for the callee function.
- These instructions are sometimes called the function prologue, because they appear at the start of every function.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...
```

callee: Function prologue

EIP →

```
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

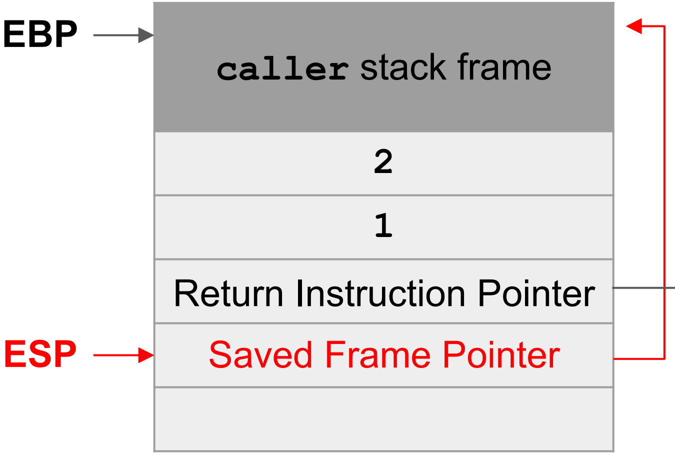
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


4. Push old EBP (SFP) on the stack

- We need to restore the value of the EBP when returning, so we push the current value of the EBP on the stack.
- The saved value of the EBP on the stack is called the saved frame pointer (SFP).




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
EIP → mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

x86 Function Call

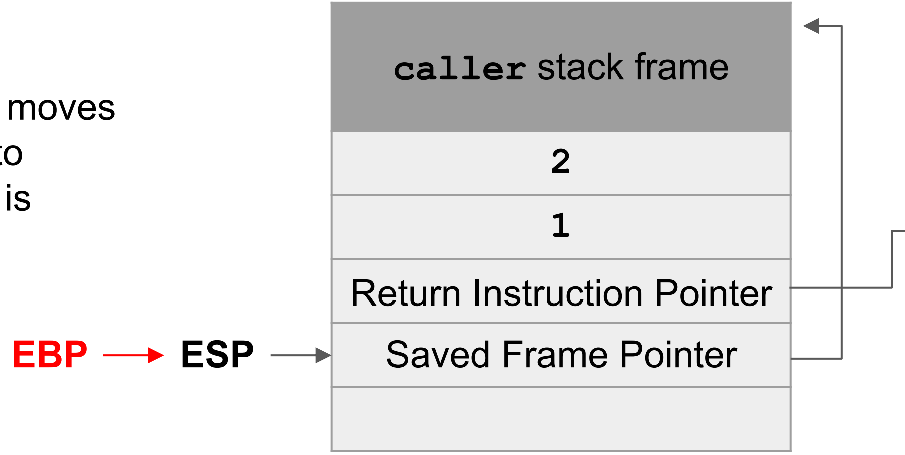
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

5. Move EBP

- This instruction moves the EBP down to where the ESP is located.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
EIP → sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

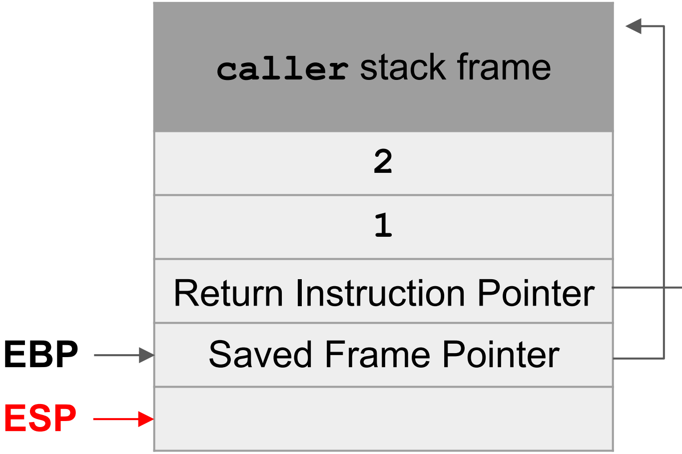
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


6. Move ESP

- This instruction moves **`esp`** down to create space for a new stack frame.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

EIP mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

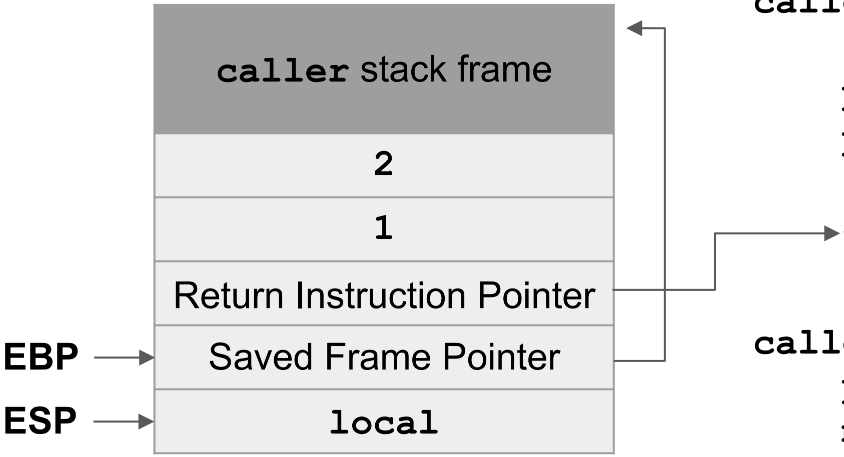
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


7. Execute the function

- Now that the stack frame is set up, the function can begin executing.
- This function just returns 42, so we put 42 in the EAX register. (Recall the return value is placed in EAX.)




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

EIP → mov %ebp, %esp
    pop %ebp
    ret
```

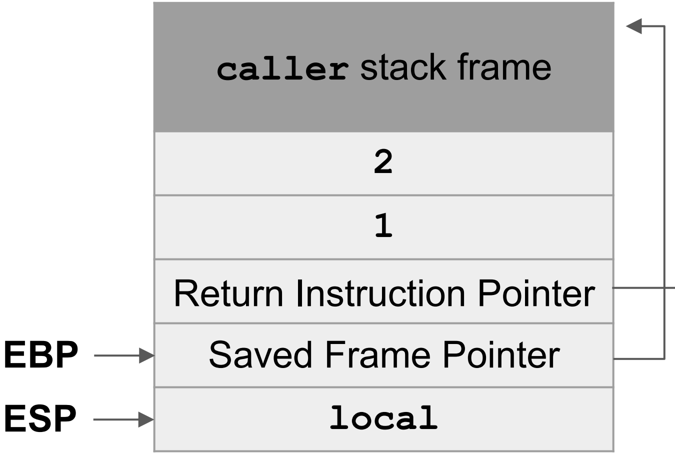
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

- The next 3 steps restore the caller's stack frame.
- These instructions are sometimes called the function epilogue, because they appear at the end of every function.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP → mov %ebp, %esp

Function epilogue

x86 Function Call

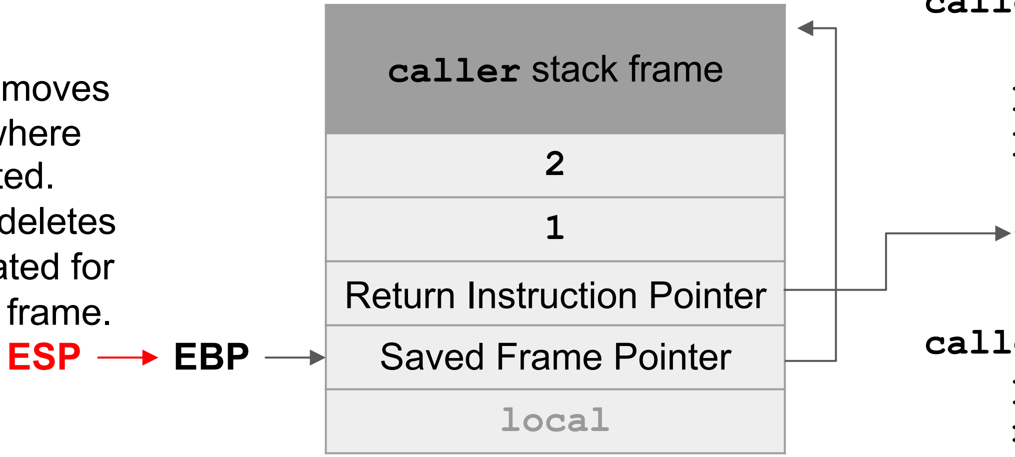
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

ITIS 6200 / 8200

8. Move ESP

- This instruction moves the ESP up to where the EBP is located.
- This effectively deletes the space allocated for the callee stack frame.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

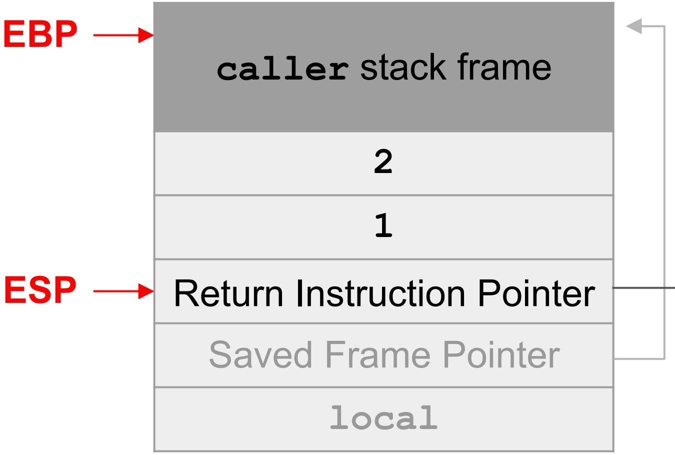
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


9. Pop (restore) old EBP (SFP)

- The `pop` instruction puts the SFP (saved EBP) back in EBP.
- It also increments ESP to delete the popped SFP from the stack.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
EIP → ret
```

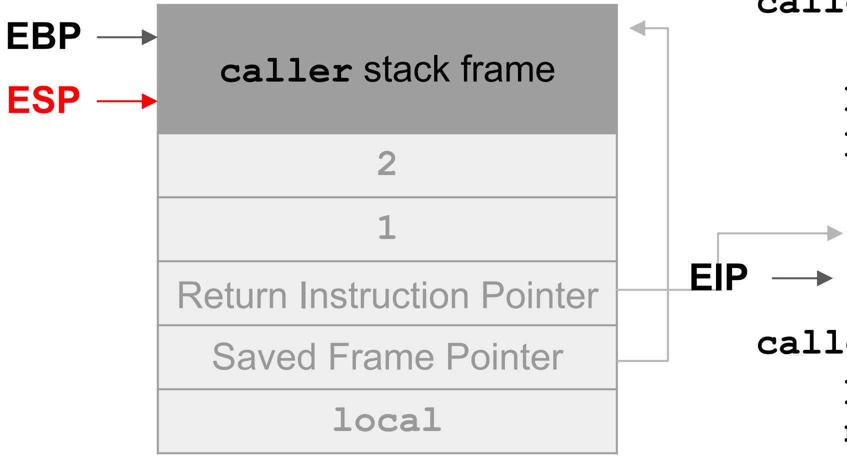
x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```


11. Remove arguments from stack

- Back in the caller, we increment ESP to delete the arguments from the stack.
- The stack has returned to its original state before the function call!



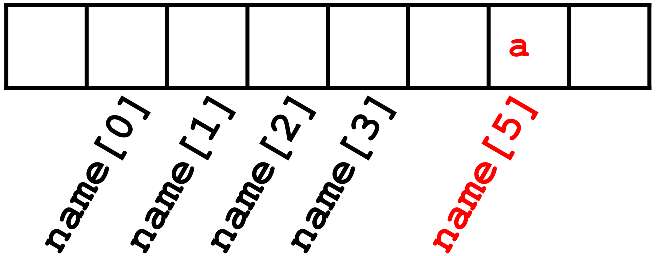

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

Buffer Overflow Vulnerabilities

Buffer Overflow Vulnerabilities

- Recall: C has no concept of array length; it just sees a sequence of bytes
- If you allow an attacker to start writing at a location and don't define when they must stop, they can overwrite other parts of memory!

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



This is technically valid C code, because C doesn't check bounds!

Vulnerable Code


```
char name[20];

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

The **`gets`** function will write bytes until the input contains a newline (**`'\n'`**), *not* when the end of the array is reached!

Okay, but there's nothing to overwrite—for now…

Vulnerable Code


```
char name[20];
char instrux[20] = "none";

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

What does the memory diagram of static data look like now?

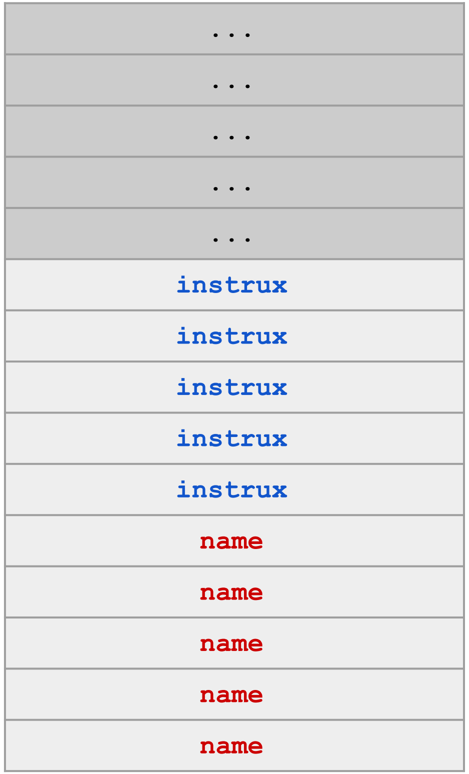
Vulnerable Code

What can go wrong here?

`gets` starts writing here and can overwrite anything above **`name`**!

```
char name[20];
char instrux[20] = "none";

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

Note: **`name`** and **`instrux`** are declared in static memory (outside of the stack), which is why **`name`** is below **`instrux`**

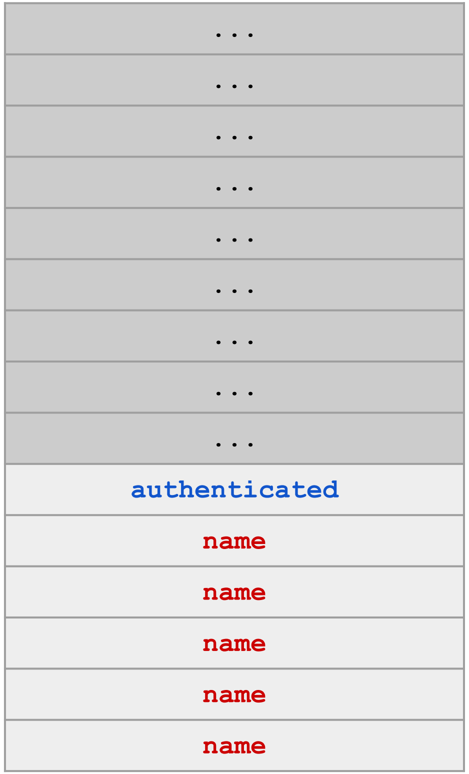


Vulnerable Code

What can go wrong here?

gets starts writing here and can overwrite the **authenticated** flag!

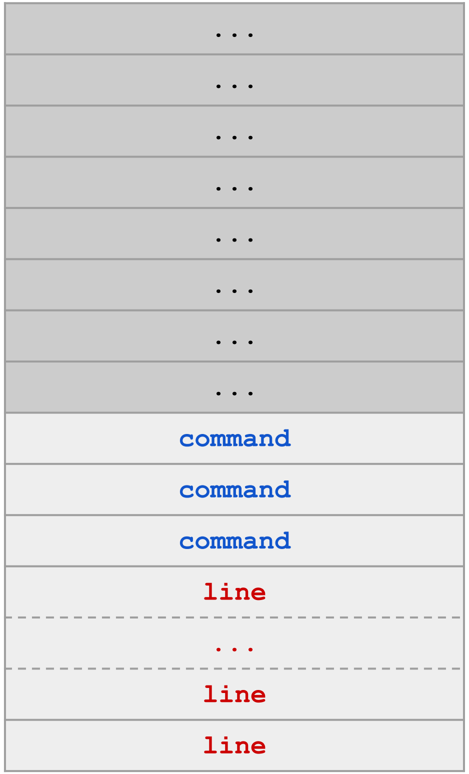
```
char name[20];
int authenticated = 0;

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



Vulnerable Code

What can go wrong here?

```
char line[512];
char command[] = "/usr/bin/ls";

int main(void) {
    ...
    gets(line);
    ...
    execv(command, ...);
}
```

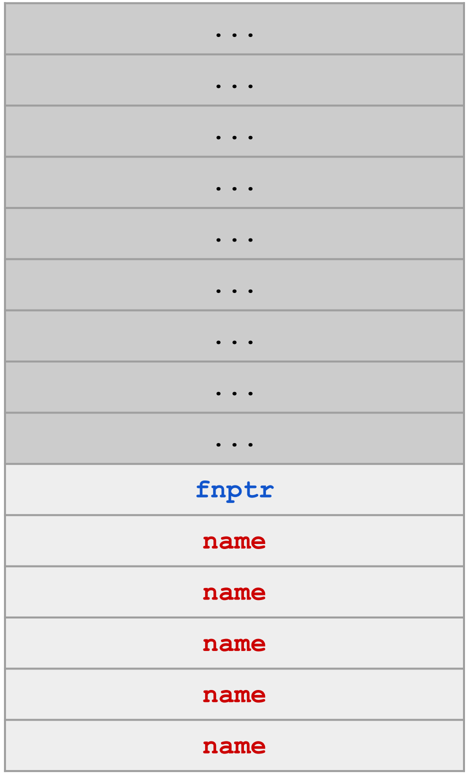


Vulnerable Code

What can go wrong here?

fnptr is called as a function, so the EIP jumps to an address of our choosing!

```
char name[20];
int (*fnptr)(void);

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



Top 25 Most Dangerous Software Weaknesses (2020)

ITIS 6200 / 8200

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

Stack Smashing

Stack Smashing

- The most common kind of buffer overflow
- Occurs on stack memory
- Recall: What does are some values on the stack an attacker can overflow?
  - Local variables
  - Function arguments
  - Saved frame pointer (SFP)
  - Return instruction pointer (RIP)
- Recall: When returning from a program, the EIP is set to the value of the RIP saved on the stack in memory
  - Like the function pointer, this lets the attacker choose an address to jump (return) to!

Note: Python Syntax

- We will see Python syntax used to represent sequences of bytes

- Adding strings: Concatenation
  - `'abc' + 'def' == 'abcdef'`

- Multiplying strings: Repeated concatenation
  - `'a' * 5 == 'aaaaa'`
  - `'ITIS6200' * 3 == 'ITIS6200ITIS6200ITIS6200'`

Note: Python Syntax

- Raw bytes
  - `len('\xff') == 1`
- Characters can be represented as bytes too
  - `'\x41' == 'A'`
  - ASCII representation: All characters are bytes, but not all bytes are characters

Overwriting the RIP

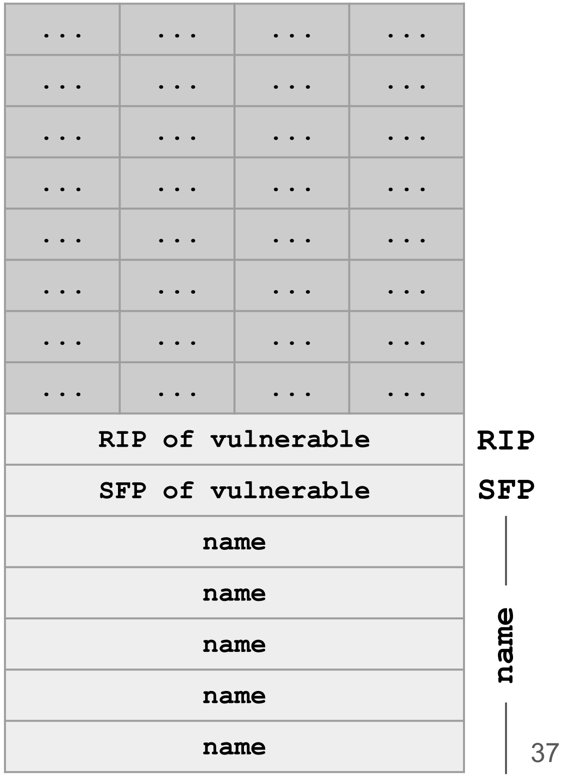
Assume that the attacker wants to execute instructions at address **`0xdeadbeef`**.

What value should the attacker write in memory? Where should the value be written?

What should an attacker supply as input to the **`gets`** function?

`gets` starts writing here and can overwrite anything above **`name`**, including the RIP!

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

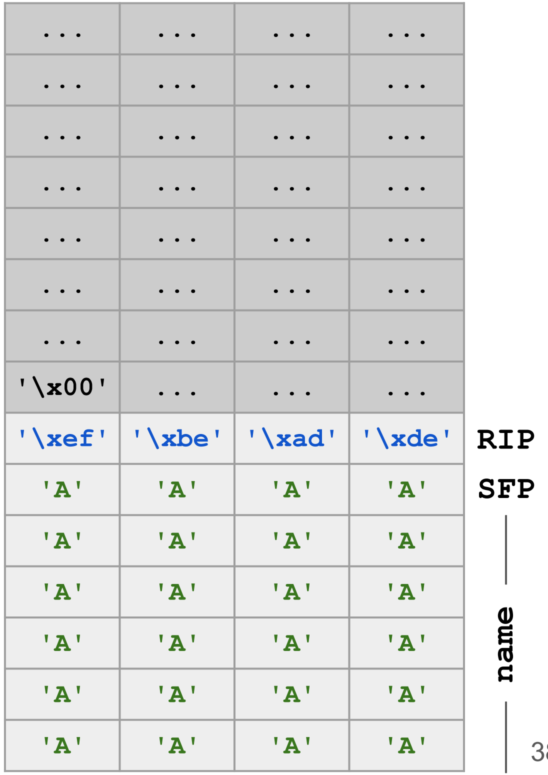


Overwriting the RIP

Note the NULL byte that terminates the string, automatically added by **`gets`**!

- Input: `'A' * 24 + '\xef\xbe\xad\xde'`
  - 24 garbage bytes to overwrite all of **`name`** and the SFP of **`vulnerable`**
  - The address of the instructions we want to execute
    - Remember: Addresses are little-endian!
- What if we want to execute instructions that aren't in memory?

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Writing Malicious Code

- The most common way of executing malicious code is to place it in memory yourself
    - Recall: Machine code is made of bytes
- **Shellcode**: Malicious code inserted by the attacker into memory, to be executed using a memory safety exploit
    - Called shellcode because it usually spawns a shell (terminal)
    - Could also delete files, run another program, etc.

```
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
mov %eax, %ecx
mov %eax, %edx
mov $0xb, %al
int $0x80
```

Assembler

```
0x31 0xc0 0x50 0x68
0x2f 0x2f 0x73 0x68
0x68 0x2f 0x62 0x69
0x6e 0x89 0xe3 0x89
0xc1 0x89 0xc2 0xb0
0x0b 0xcd 0x80
```

Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
   - Often, the shellcode can be written and the RIP can be overwritten in the same function call (e.g. `gets`), like in the previous example
4. Return from the function
5. Begin executing malicious shellcode

Constructing Exploits

Let **`SHELLCODE`** be a 12-byte shellcode. Assume that the address of **`name`** is **`0xbfffcd40`**.

What values should the attacker write in memory? Where should the values be written?

What should an attacker supply as input to the **`gets`** function?

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

Constructing Exploits

- Input: **SHELLCODE** + `'A' * 12` + `'\x40\xcd\xff\xbf'`
  - 12 bytes of shellcode
  - 12 garbage bytes to overwrite the rest of `name` and the SFP of `vulnerable`
  - The address of where we placed the shellcode

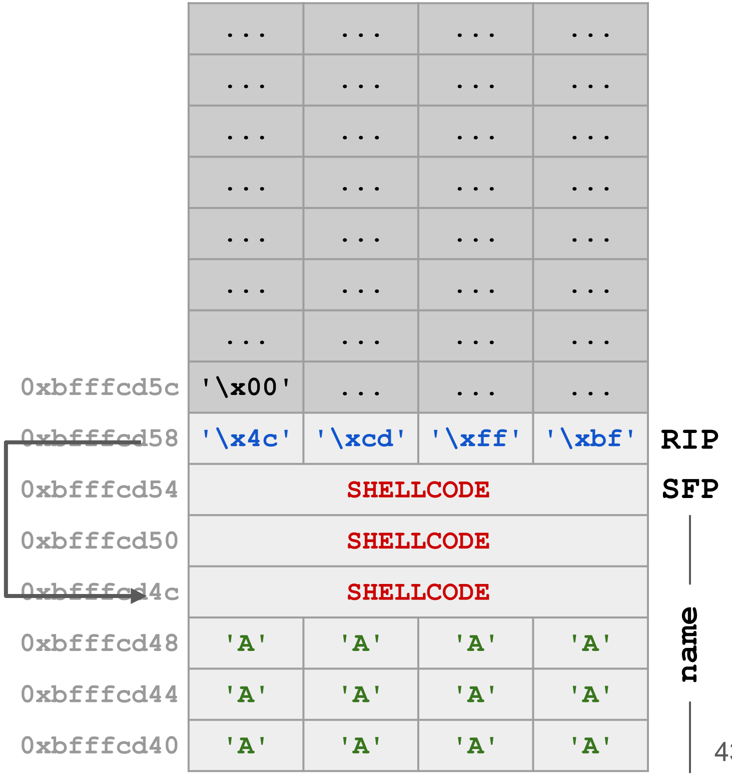
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	'\x00'	...	...	...	
0xbfffcd58	'\x40'	'\xcd'	'\xff'	'\xbf'	RIP
0xbfffcd54	'A'	'A'	'A'	'A'	SFP
0xbfffcd50	'A'	'A'	'A'	'A'	name
0xbfffcd4c	'A'	'A'	'A'	'A'	name
0xbfffcd48	SHELLCODE				name
0xbfffcd44	SHELLCODE				name
0xbfffcd40	SHELLCODE				name

Constructing Exploits

- Alternative: `'A' * 12 + SHELLCODE + '\x4c\xcd\xff\xbf'`
  - The address changed! Why?
    - We placed our shellcode at a different address (`name + 12`)!

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



Constructing Exploits

What if the shellcode is too large? Now let **`SHELLCODE`** be a 28-byte shellcode. What should the attacker input?

```c
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

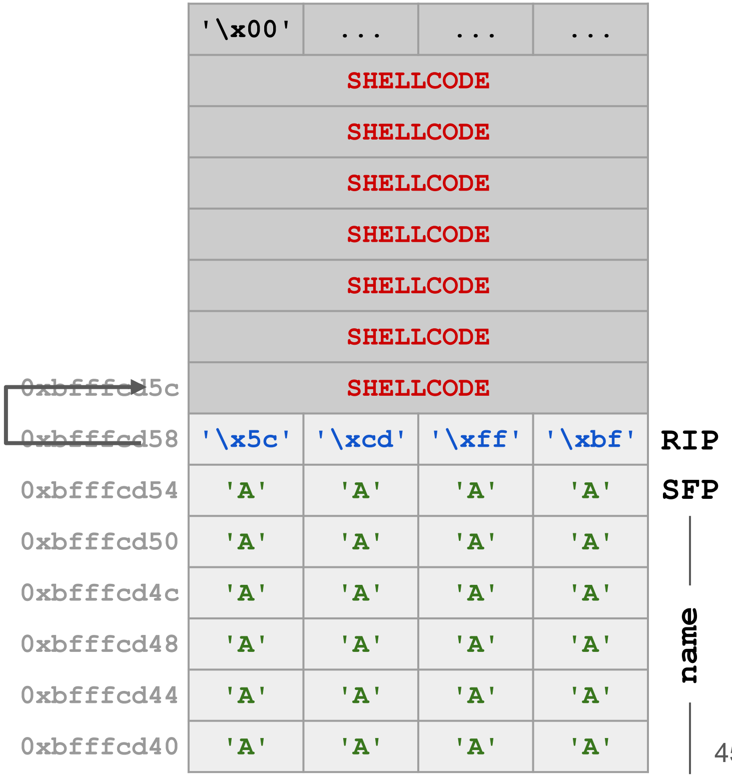
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	...	...	...	...	
0xbfffcd58	RIP of vulnerable				RIP
0xbfffcd54	SFP of vulnerable				SFP
0xbfffcd50	name				name
0xbfffcd4c	name				
0xbfffcd48	name				
0xbfffcd44	name				
0xbfffcd40	name				

Constructing Exploits

ITIS 6200 / 8200

- Solution: Place the shellcode *after* the RIP!
  - This works because **`gets`** lets us write as many bytes as we want
  - What should the address be?
- Input: **`'A' * 24 +`**
  `'\x5c\xcd\xff\xbf' + SHELLCODE`
  - 24 bytes of garbage
  - The address of where we placed the shellcode
  - 28 bytes of shellcode

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



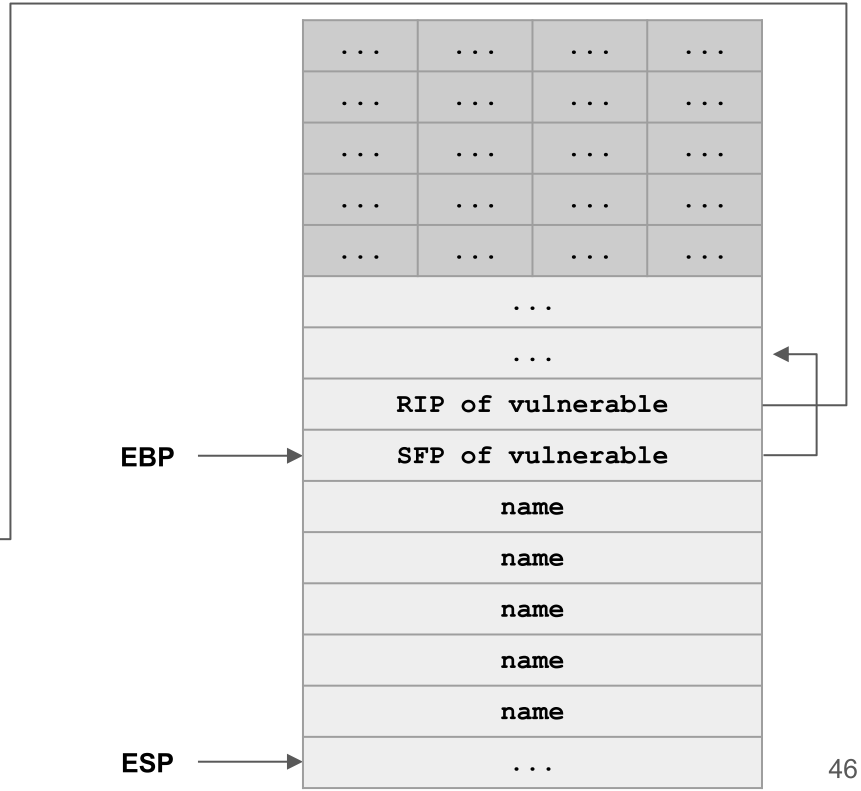
Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```c
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

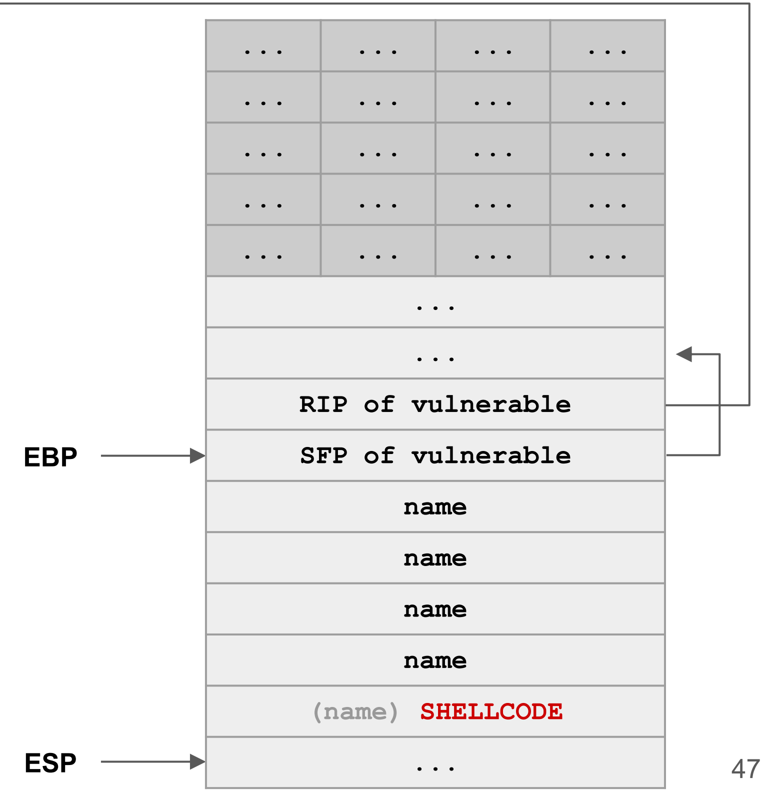


Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



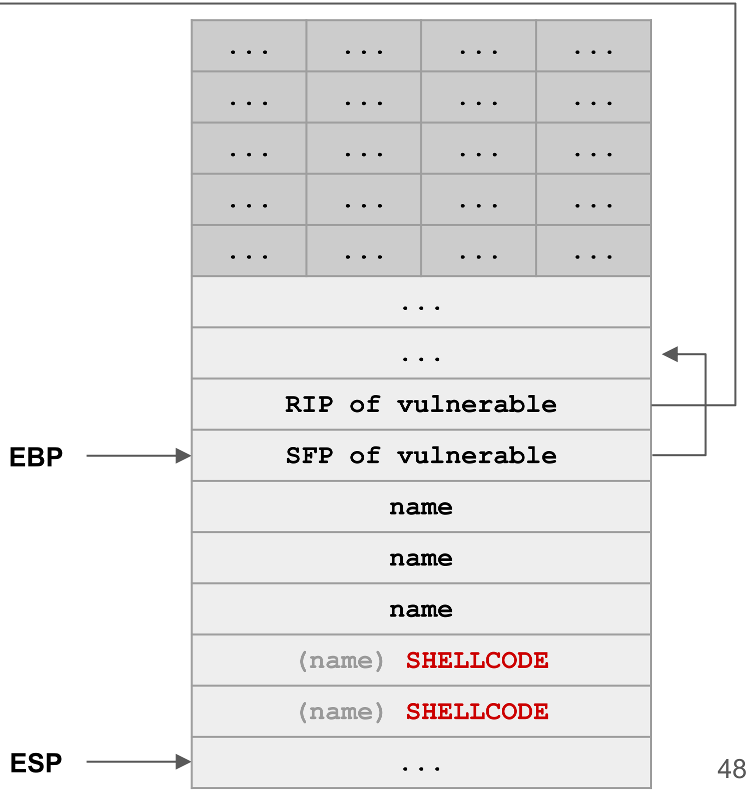
Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

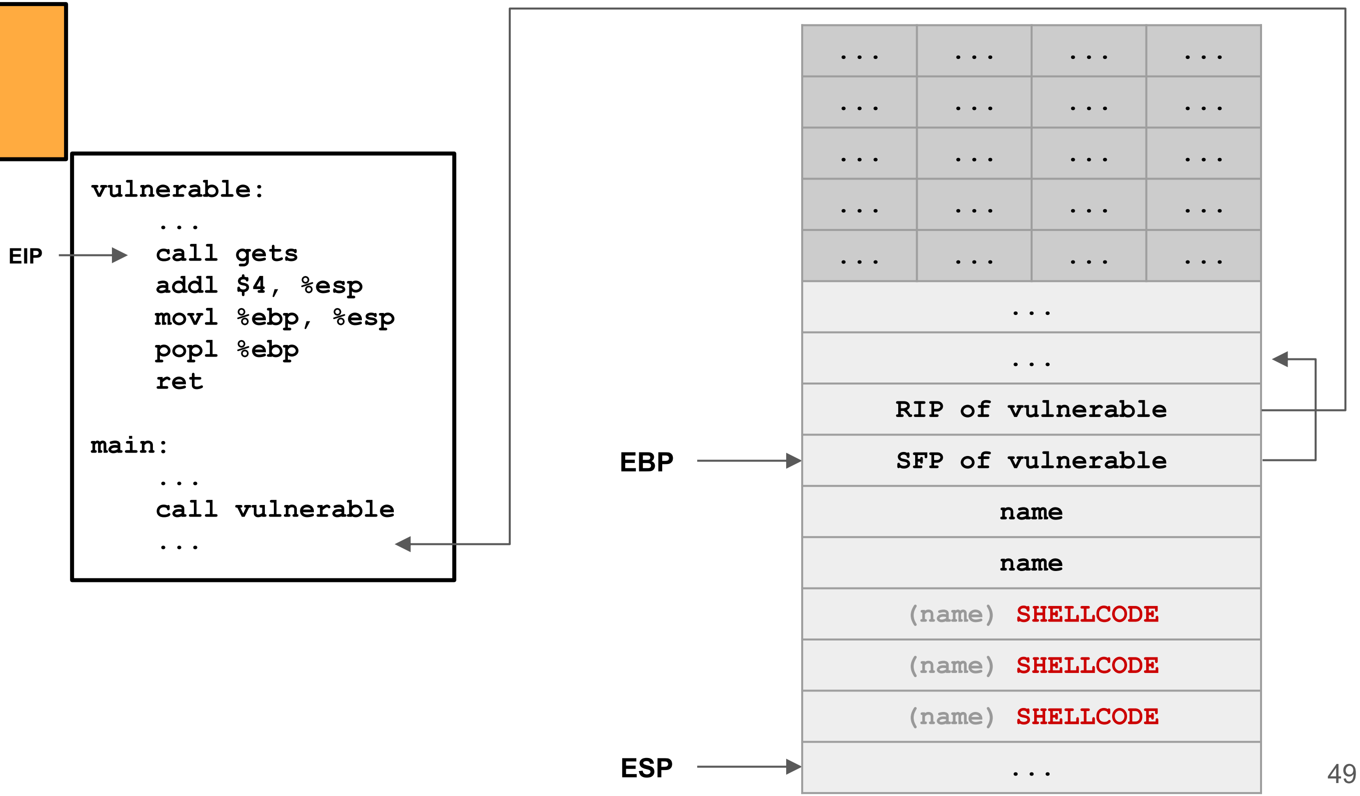


Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



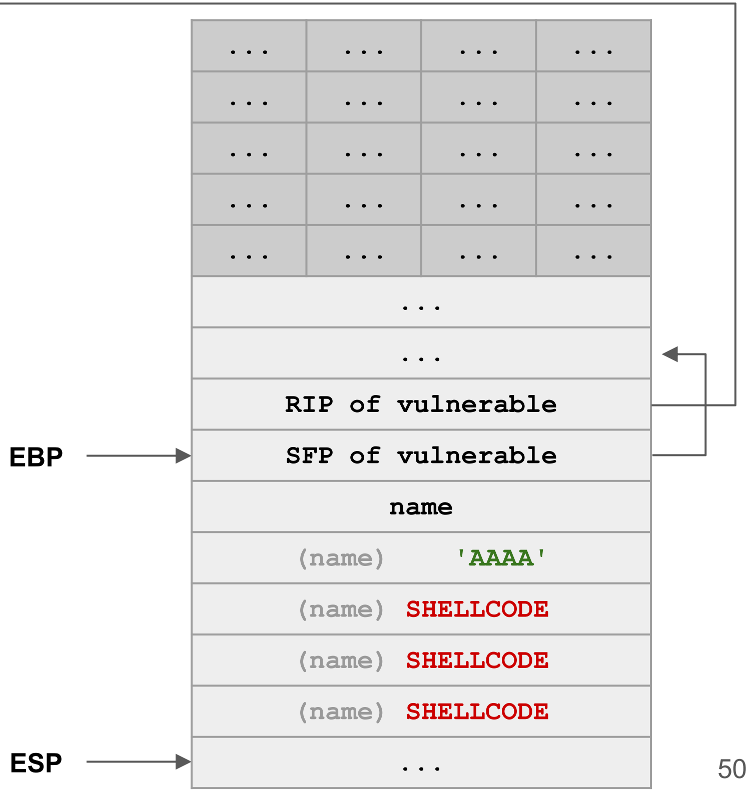
Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



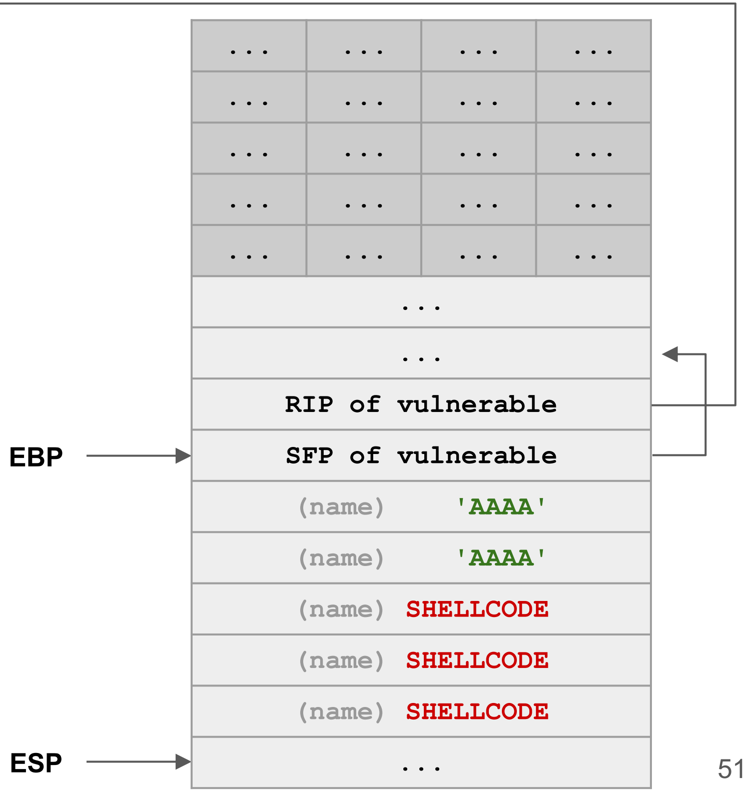
Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets      <- EIP
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...			
	...			
	RIP of vulnerable			
EBP →	(SFP) 'AAAA'			
	(name) 'AAAA'			
	(name) 'AAAA'			
	(name) SHELLCODE			
	(name) SHELLCODE			
	(name) SHELLCODE			
ESP →	...			

We overwrite the SFP (saved EBP) with 'AAAA', so the SFP is now pointing at the (probably invalid) address AAAA (0x41414141)

Walking Through a Buffer Overflow

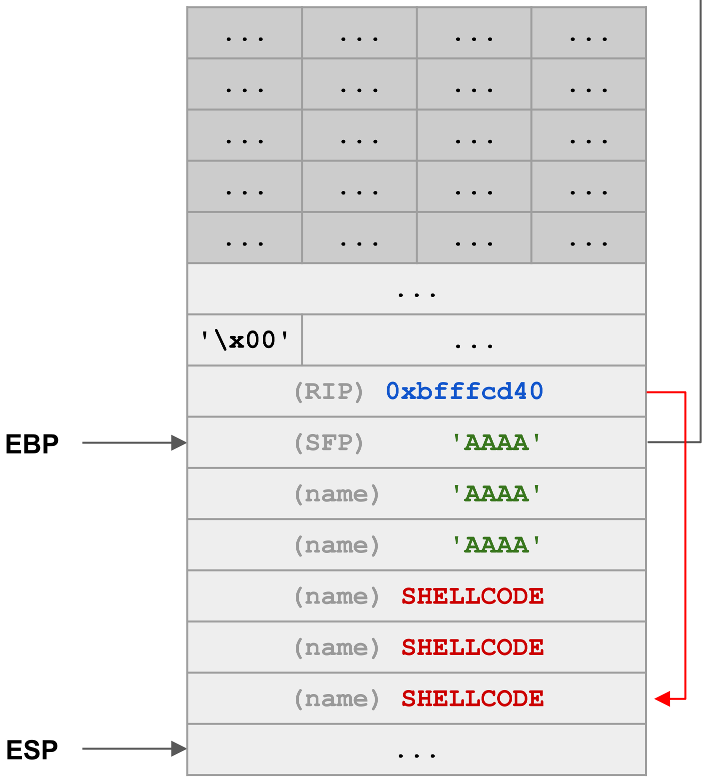
Input:

```
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'
```

```c
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP → `addl $4, %esp`

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



We overwrite the RIP (saved EIP) with the address of our shellcode `0xbfffcd40`, so the RIP is now pointing at our shellcode! Remember, this value will be restored to EIP (the instruction pointer) later.

Walking Through a Buffer Overflow

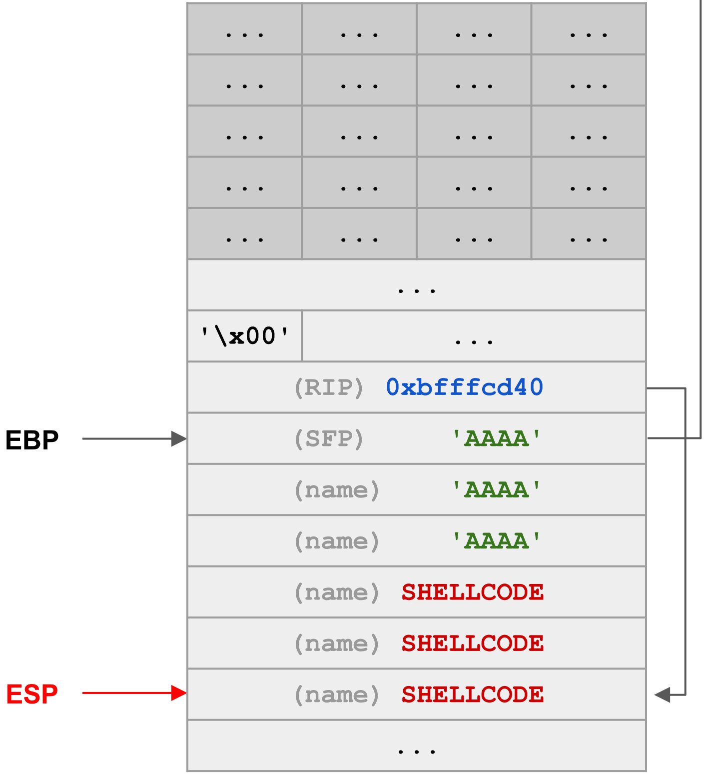
Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Returning from **gets**: Move ESP up by 4.



Walking Through a Buffer Overflow

Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → `popl %ebp`

ESP → EBP →

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...			
'\x00'	...		
(RIP) 0xbfffcd40			
(SFP) 'AAAA'			
(name) 'AAAA'			
(name) 'AAAA'			
(name) SHELLCODE			
(name) SHELLCODE			
(name) SHELLCODE			
...			

Function epilogue: Move ESP to EBP.

Walking Through a Buffer Overflow

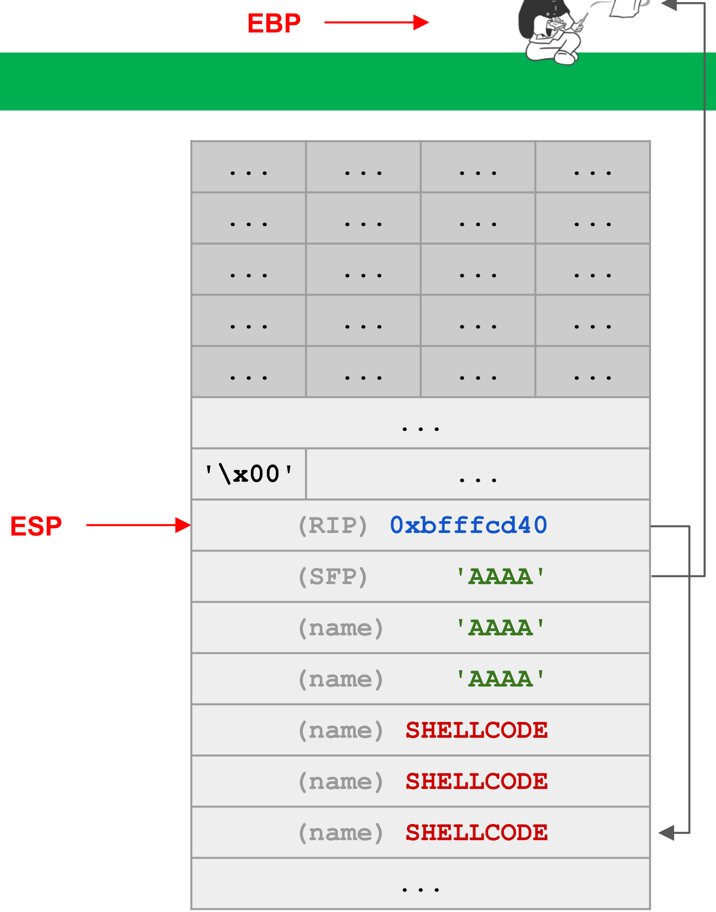
ITIS 6200 / 8200

Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

EIP → ret



Function epilogue: Restore the SFP into EBP. We overwrote SFP to 'AAAA', so the EBP now also points to the address 'AAAA'. We don't really care about EBP, though.

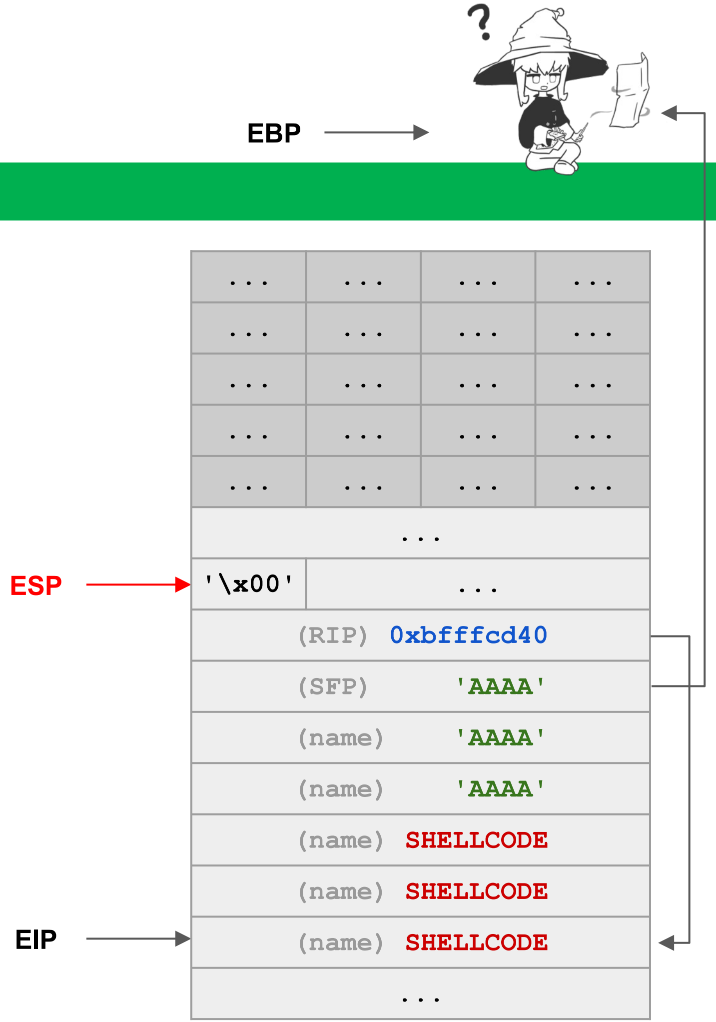
Walking Through a Buffer Overflow

Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Function epilogue: Restore the RIP into EIP. We overwrote RIP to the address of shellcode, so the EIP (instruction pointer) now points to our shellcode!



Walking Through a Buffer Overflow

EBP →

ITIS 6200 / 8200

Inpu
SHELLCODE +
'\x40\xcd\

```
void vulnerable(v
    char name[20]
    gets(name);
}

int main(void) {
    vulnerable()
    return 0;
}
```

sh # _

...	...
...	...
...	...
...	...
...	...

...

...

0xbfffcd40

'AAAA'

'AAAA'

'AAAA'

SHELLCODE

SHELLCODE

SHELLCODE

...

Memory-Safe Code

Still Vulnerable Code?


```
void vulnerable?(void) {
     char *name = malloc(20);
     ...
     gets(name);
     ...
}
```

Heap overflows are also vulnerable!

Solution: Specify the Size


```
void safe(void) {
        char name[20];
    ...
        fgets(name, 20,
stdin);
    ...
}
```

The length parameter specifies the size of the buffer and won't write any more bytes—no more buffer overflows!

Solution: Specify the Size


```
void safer(void) {
        char name[20];
    ...
        fgets(name, sizeof(name),
stdin);
    ...
}
```

`sizeof` returns the size of the variable (does **not** work for pointers)

Vulnerable C Library Functions

- `gets` - Read a string from stdin
  - Use `fgets` instead
- `strcpy` - Copy a string
  - Use `strncpy` (more compatible, less safe) or `strlcpy` (less compatible, more safe) instead
- `strlen` - Get the length of a string
  - Use `strnlen` instead (or `memchr` if you really need compatible code)
- … and more (look up C functions before you use them!)
  - `man` pages are your friend!

Integer Memory Safety Vulnerabilities

Signed/Unsigned Vulnerabilities

Is this safe?

```
void func(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

This is a **signed** comparison, so `len > 64` will be false, but casting **-1** to an unsigned type yields `0xffffffff`: another buffer overflow!

`int` is a **signed** type, but `size_t` is an **unsigned** type. What happens if `len == -1`?

```
void *memcpy(void *dest, const void *src, size_t n);
```

Signed/Unsigned Vulnerabilities


```
void safe(size_t len, char *data)
{
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

Now this is an **unsigned** comparison, and no casting is necessary!

Integer Overflow Vulnerabilities

Is this safe?

What happens if `len == 0xffffffff`?

```
void func(size_t len, char *data)
{
    char *buf = malloc(len + 2);
    if (!buf)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

`len + 2 == 1`, enabling a heap overflow!

Integer Overflow Vulnerabilities


```
void safe(size_t len, char *data)
{
    if (len > SIZE_MAX - 2)
        return;
    char *buf = malloc(len + 2);
    if (!buf)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len + 1] = '\0';
}
```

It's clunky, but you need to check bounds whenever you add to integers!

Integer Overflows in the Wild

WJXT Jacksonville [Link]

Broward Vote-Counting Blunder Changes Amendment Result *November 4, 2004*

The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.

Integer Overflows in the Wild

- 32,000 votes is very close to 32,768, or 2^{15} (the article probably rounded)
    - Recall: The maximum value of a signed, 16-bit integer is $2^{15} - 1$
    - This means that an integer overflow would cause -32,768 votes to be counted!
- **Takeaway**: Check the limits of data types used, and choose the right data type for the job
    - If writing software, consider the largest possible use case.
        - 32 bits might be enough for Broward County but isn't enough for everyone on Earth!
        - 64 bits, however, would be plenty.

Another Integer Overflow in the Wild

9TO5LINUX
Linux news, reviews, tutorials, and more

9 to 5 Linux

Link

New Linux Kernel Vulnerability Patched in All Supported Ubuntu Systems, Update Now

Marius Nestor

January 19, 2022

Discovered by William Liu and Jamie Hill-Daniel, the new security flaw (CVE-2022-0185) is an integer underflow vulnerability found in Linux kernel’s file system context functionality, which could allow an attacker to crash the system or run programs as an administrator.

How Does This Vulnerability Work?

- The entire kernel (operating system) patch:

```
- if (len > PAGE_SIZE - 2 - size)
+ if (size + len + 2 > PAGE_SIZE)
    return invalf(fc, "VFS: Legacy: Cumulative options too large)
```

- Why is this a problem?
  - **`PAGE_SIZE`** and **`size`** are unsigned
  - If **`size`** is larger than **`PAGE_SIZE`**…
  - …then **`PAGE_SIZE - 2 - size`** will trigger a negative overflow to **`0xFFFFFFFF`**
- Result: An attacker can bypass the length check and write data into the kernel

Format String Vulnerabilities

Review: `printf` behavior

ITIS 6200 / 8200

- Recall: `printf` takes in an variable number of arguments
  - How does it know how many arguments that it received?
  - It infers it from the first argument: the format string!
  - Example: `printf("One %s costs %d", fruit, price)`
  - What happens if the arguments are mismatched?

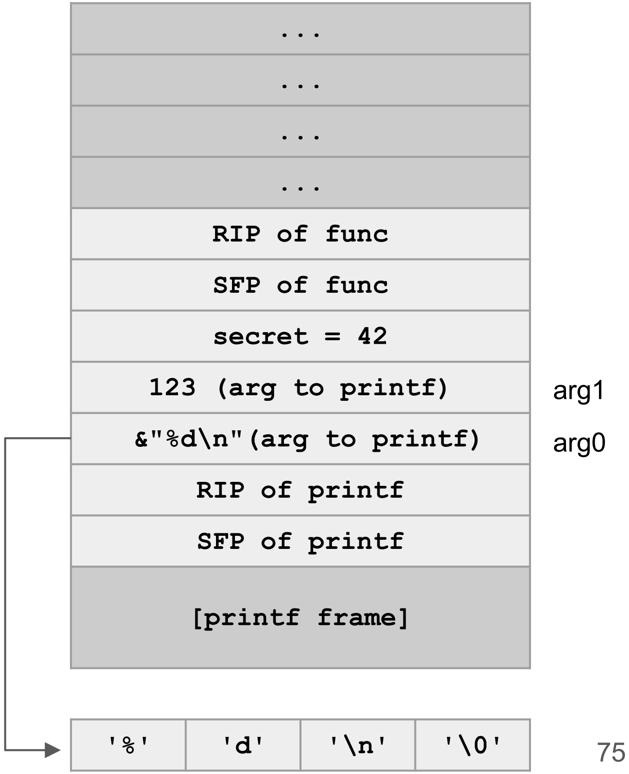
Review: `printf` behavior

ITIS 6200 / 8200

```
void func(void) {
    int secret = 42;
    printf("%d\n", 123);
}
```

`printf` assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?

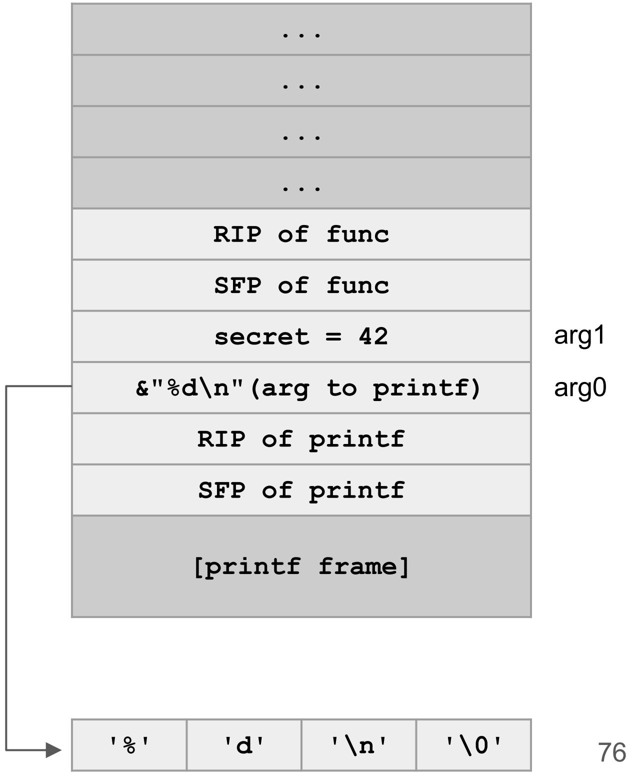


Review: `printf` behavior

ITIS 6200 / 8200

```c
void func(void) {
    int secret = 42;
    printf("%d\n");
}
```

Because the format string contains the `%d`, it will still look 4 bytes up and print the value of **`secret`**!



Format String Vulnerabilities

What is the issue here?

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

Format String Vulnerabilities

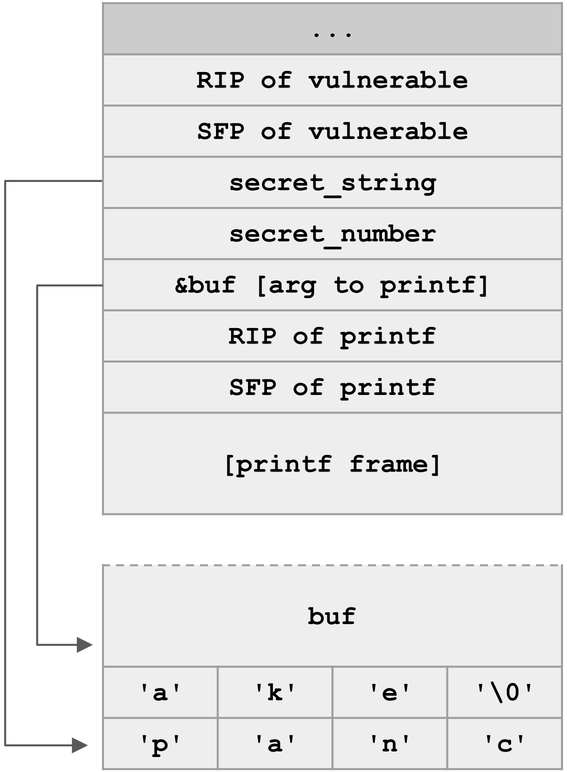
- Now, the attacker can specify any format string they want:
  - `printf("100% done!")`
    - Prints 4 bytes on the stack, 8 bytes above the RIP of `printf`
  - `printf("100% stopped.")`
    - Print the bytes **pointed to** by the address located 8 bytes above the RIP of `printf`, until the first NULL byte
  - `printf("%x %x %x %x ...")`
    - Print a series of values on the stack in hex

```
char buf[64];

void vulnerable(void) {
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

Format String Vulnerability Walkthrough


```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

Note that strings are passed by reference in C, so the argument to **`printf`** is actually a pointer to **`buf`**, which is in static memory.



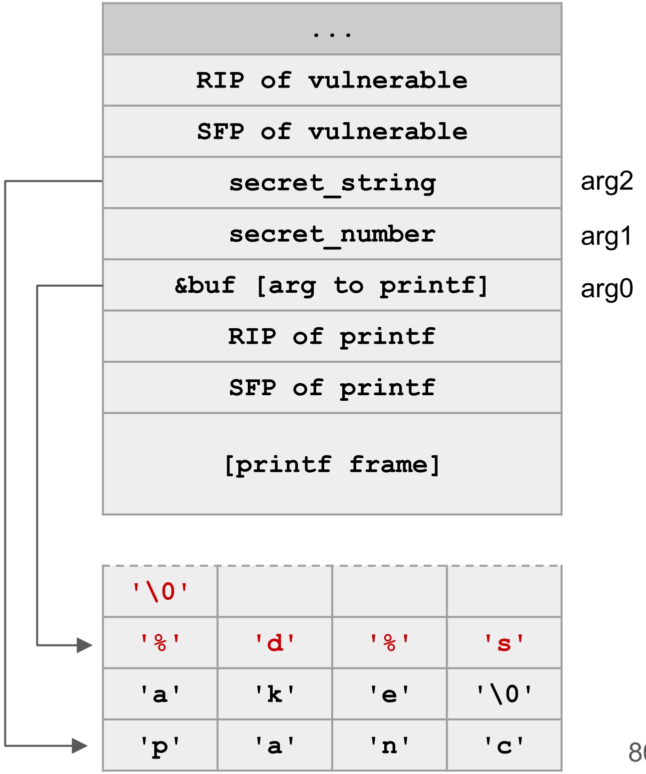
Format String Vulnerability Walkthrough

ITIS 6200 / 8200

Input: %d%s

Output:

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```



We're calling **printf("%d%s")**. **printf** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

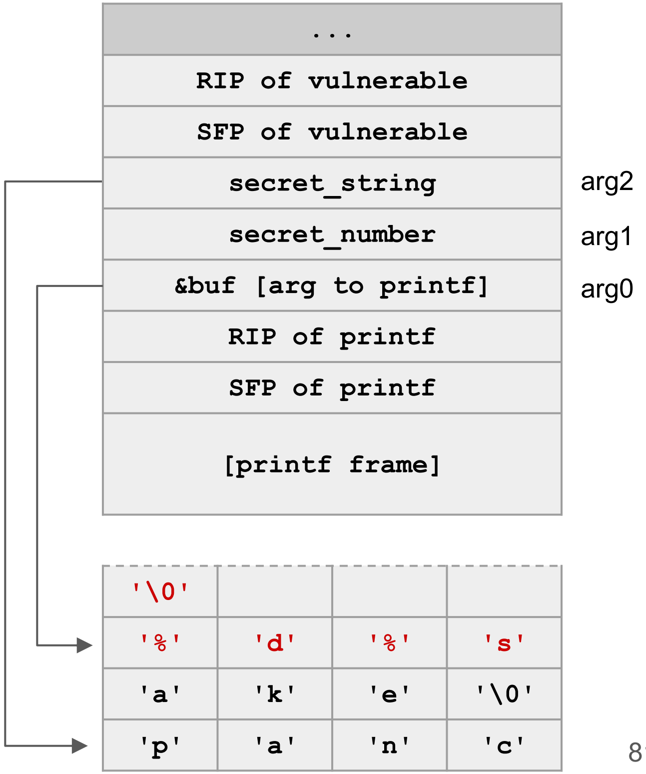
Format String Vulnerability Walkthrough

ITIS 6200 / 8200

Input: `%d%s`

Output:
42

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```



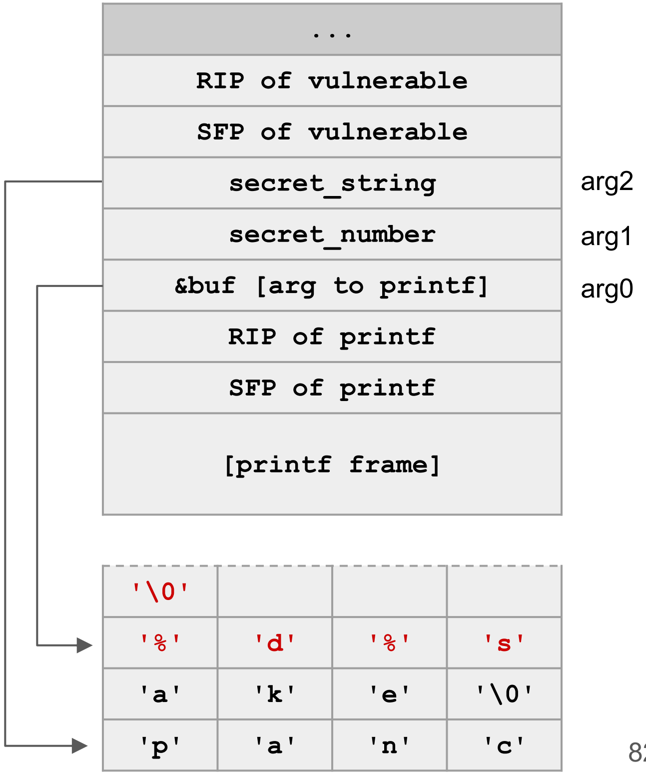
The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

Format String Vulnerability Walkthrough

Input: `%d%s`

Output:
42pancake

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```



The second format specifier `%s` says to treat the next argument (arg2) as an string and print it out.

`%s` will dereference the pointer at arg2 and print until it sees a null byte (`'\0'`)

Format String Vulnerabilities

ITIS 6200 / 8200

- They can also write values using the **`%n`** specifier
  - **`%n`** treats the next argument as a **pointer** and writes the number of bytes printed so far to that address (usually used to calculate output spacing)
    - **`printf("item %d:%n", 3, &val)`** stores 7 in **`val`**
    - **`printf("item %d:%n", 987, &val)`** stores 9 in **`val`**
  - **`printf("000%n")`**
    - **Writes** the value 3 to the integer **pointed to** by address located 8 bytes above the RIP of **`printf`**

```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

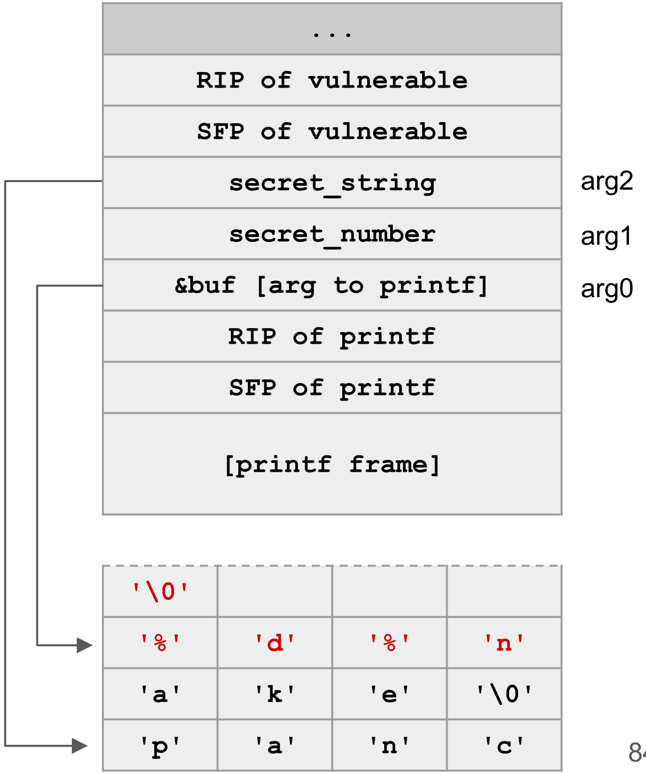
Format String Vulnerability Walkthrough

ITIS 6200 / 8200

Input: `%d%n`

Output:

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```



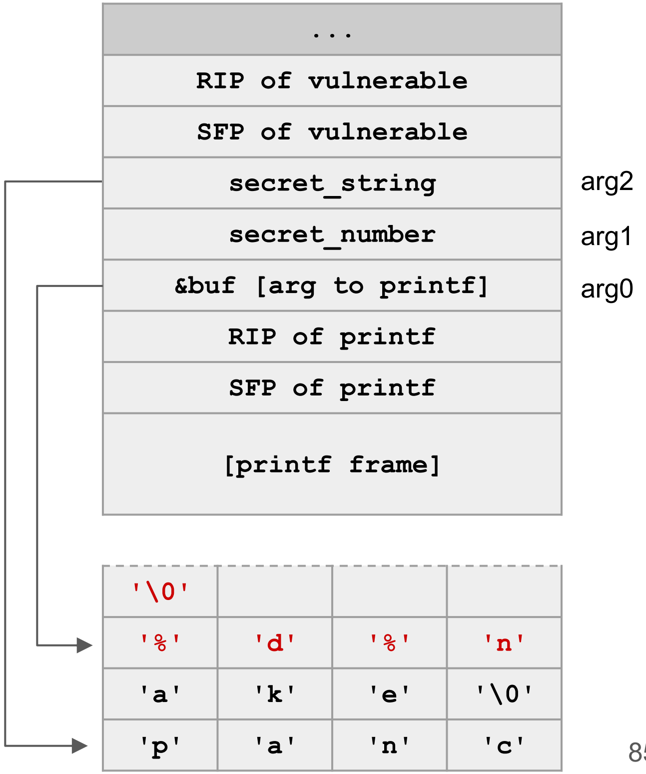
We're calling **`printf("%d%n")`**. **`printf`** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

Format String Vulnerability Walkthrough

Input: `%d%n`

Output:
42

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```



The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

Format String Vulnerability Walkthrough

ITIS 6200 / 8200

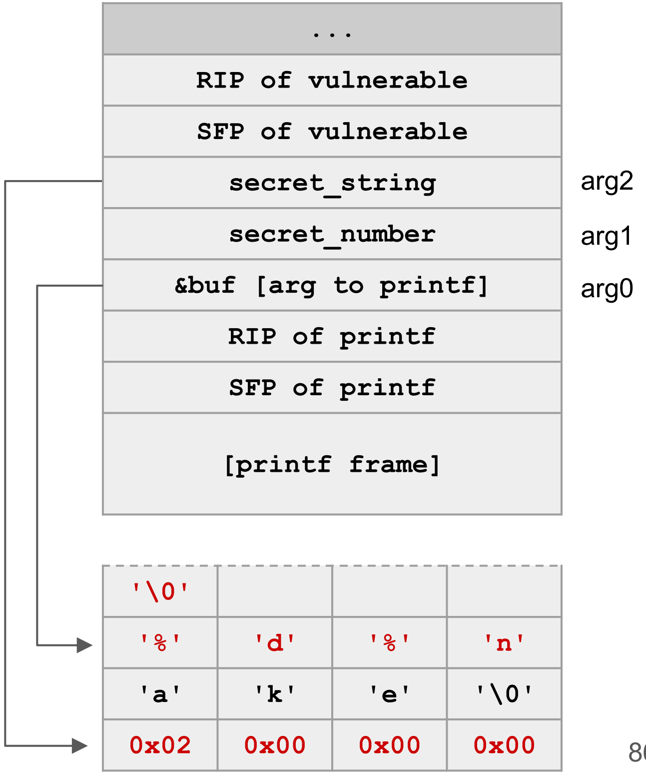
Input: `%d%s`

Output:
42

```
char buf[64];

void vulnerable(void) {
    char *secret_string = "pancake";
    int secret_number = 42;
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

The second format specifier `%n` says to treat the next argument (arg2) as a pointer, and write the number of bytes printed so far to the address at arg2.

We've printed 2 bytes so far, so the number 2 gets written to **`secret_string`**.



Format String Vulnerabilities: Defense


```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf("%s", buf);
}
```

Never use untrusted input in the first argument to **`printf`**.

Now the attacker can't make the number of arguments mismatched!