Memory Safety Mitigations

CMPSC 403 Fall 2021

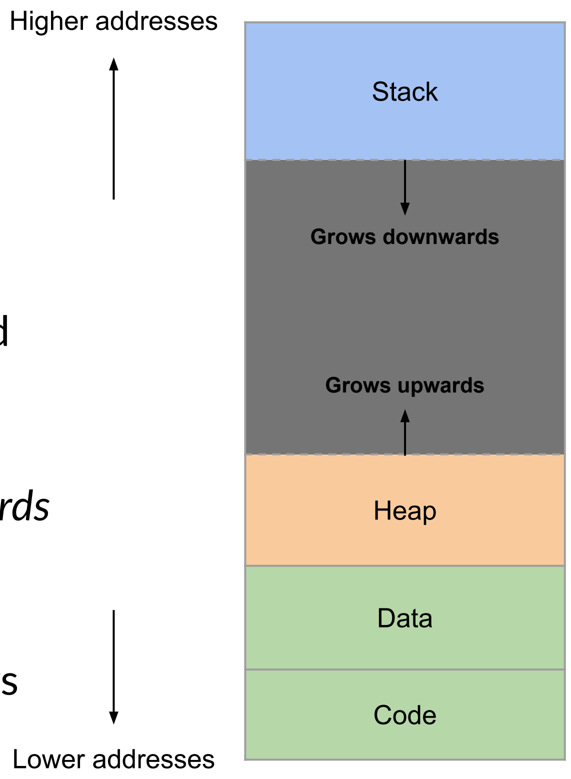
September 28 - 30, 2021

Review: x86 Function Call, Buffer Overflows

- Calling convention
  - At the start of the function, the callee saves the register values on the stack
  - During the function, the callee can now change those registers
  - At the end of the function, the callee will put the saved values on the stack back into the registers
- Important saved registers on the stack
  - When the callee saves the value of EBP on the stack, we call it the **SFP** (**saved frame pointer**)
  - When the callee saves the value of EIP on the stack, we call it the **RIP** (**return instruction pointer**)
- 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

Reminder:x86 Memory Layout

- Code
  - The program code itself (also called “text”)
- Data
  - Static variables, allocated when the program is started
- Heap
  - Dynamically allocated memory using **malloc** and **free**
  - As more and more memory is allocated, it grows *upwards*
- Stack
  - Local variables and stack frames
  - As you make deeper and deeper function calls, it grows *downwards*



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!)

Integer Memory Safety Vulnerabilities

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!

Think Along: 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.

Think Along: Integer Overflows in the Wild

<u>How could this have been prevented?</u>

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

Format String Vulnerabilities

`printf` behavior

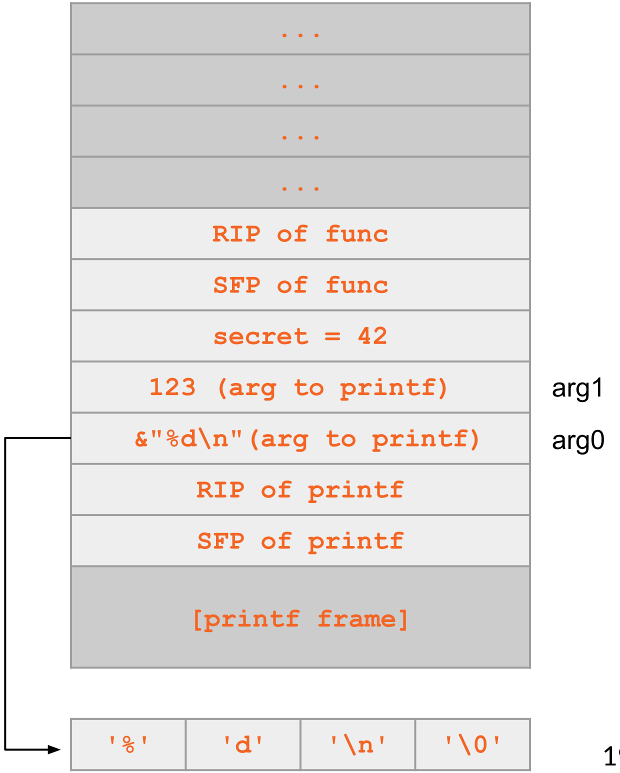
- `printf` takes in a variable number of arguments
  - How does it know how many arguments that it received?
  - It infers it from the first argument: the format string!
  - Example: `printf("One %s costs %d", fruit, price)`
  - What happens if the arguments are mismatched?

`printf` behavior

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

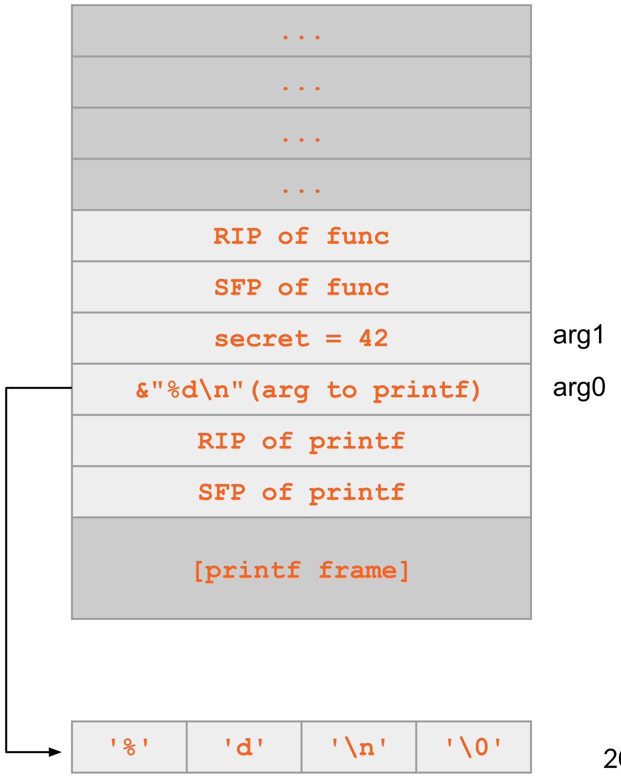
`printf` *assumes* that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?



`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

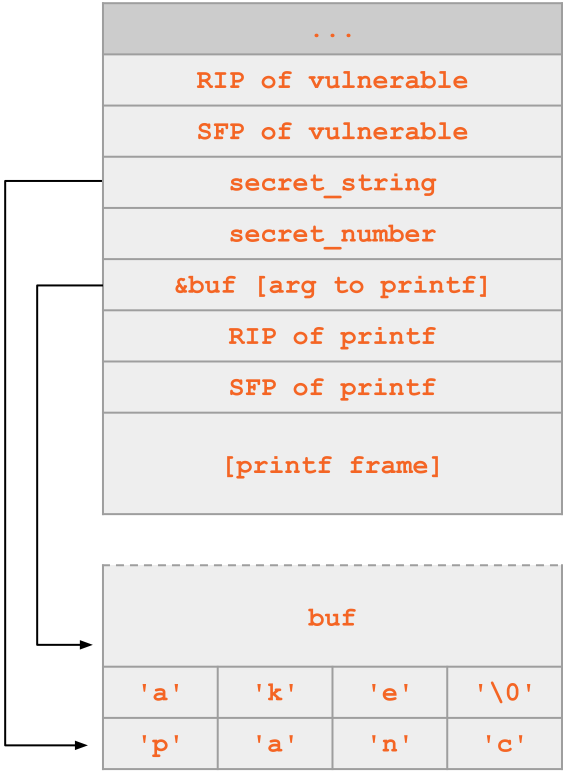
- Now, the attacker can specify any format string they want:
  - `printf("100% done!")`
    - Prints 4 bytes on the stack, 8 bytes above the RIP of `printf`
  - `printf("100% stopped.")`
    - Print the bytes **pointed to** by the address located 8 bytes above the RIP of `printf`, until the first NULL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

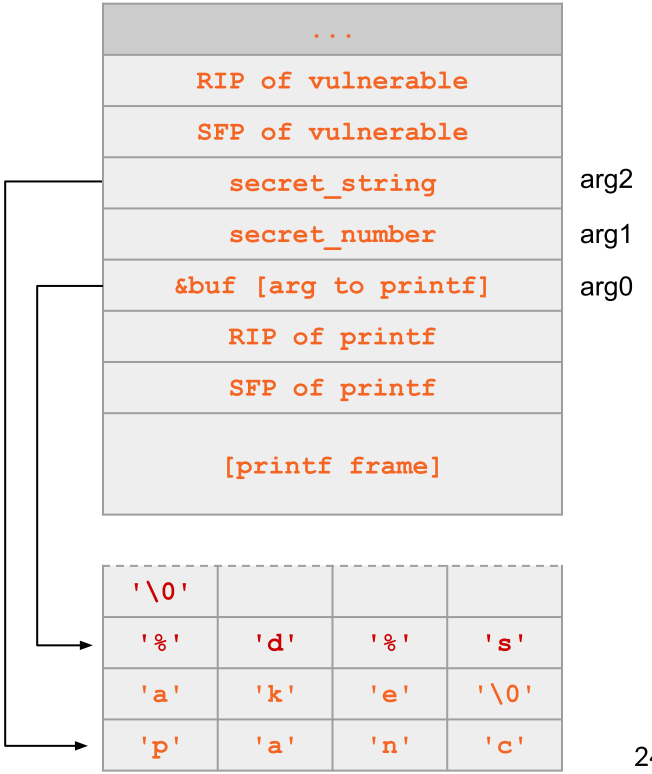


Format String Vulnerability Walkthrough

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);
}
```



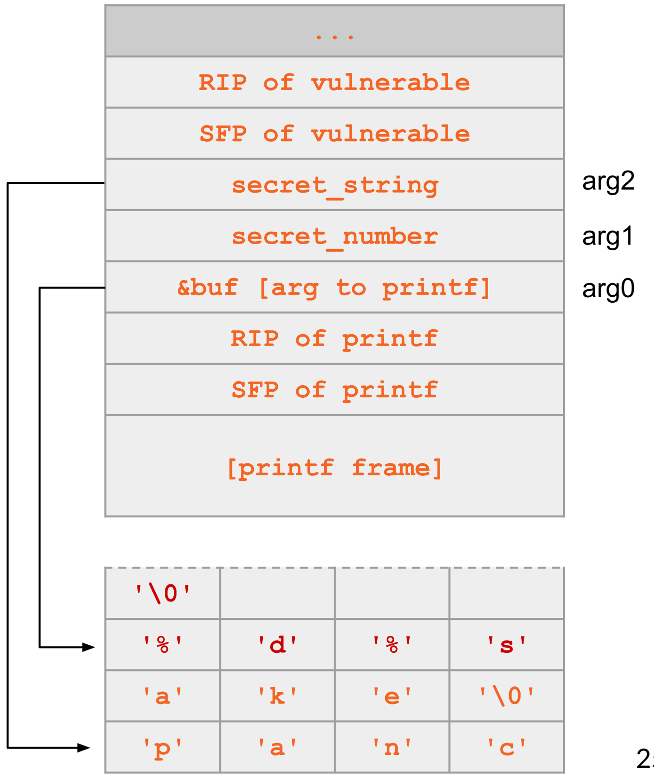
We're calling **printf("%d%s")**. **printf** reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

Format String Vulnerability Walkthrough

Input: `%d%s`

Output:
`42`

```
char buf[64];

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



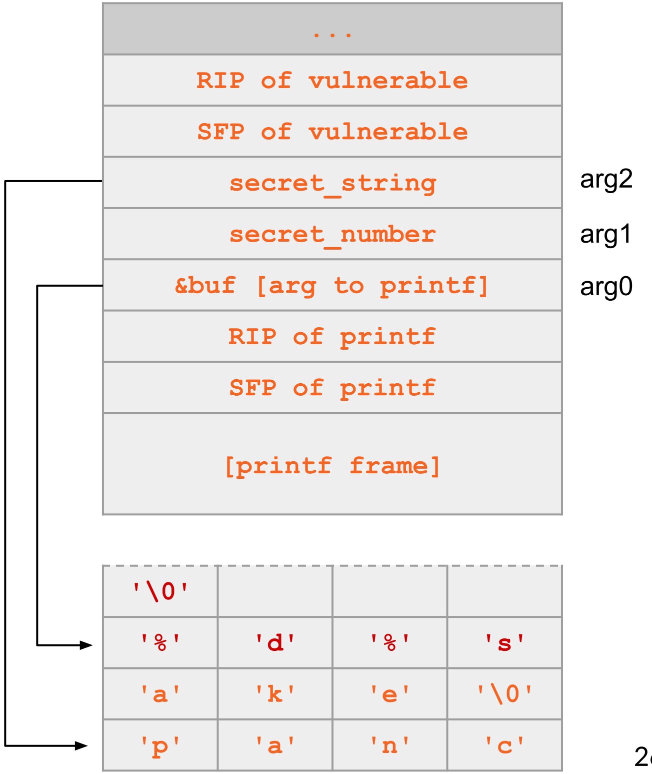
The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

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 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!

Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Using Memory-Safe Languages

Memory-Safe Languages

- **Memory-safe languages** are designed to check bounds and prevent undefined memory accesses
- By design, memory-safe languages are not vulnerable to memory safety vulnerabilities
  - Using a memory-safe language is the **only** way to stop 100% of memory safety vulnerabilities
- Examples: Java, Python, C#, Go, Rust
  - Most languages besides C, C++, and Objective C

Why Use Non-Memory-Safe Languages?

- Most commonly-cited reason: **performance**
- Comparison of memory allocation performance
  - C and C++ (not memory safe): **`malloc`** usually runs in (amortized) constant-time
  - Java (memory safe): the garbage collector may need to run at any arbitrary point in time, adding a 10–100 ms delay as it cleans up memory

The Cited Reason: The Myth of Performance

- For most applications, the performance difference from using a memory-safe language is insignificant
  - Possible exceptions: Operating systems, high performance games, some embedded systems
- C's improved performance is not a direct result of its security issues
  - Historically, safer languages were slower, so there was a tradeoff
  - Today, safe alternatives have comparable performance (e.g. Go and Rust)
  - Secure C code (with bounds checking) ends up running as quickly as code in a memory-safe language anyway
  - You don't need to pick between security and performance: You can have both!

The Cited Reason: The Myth of Performance

- Programmer time matters too
  - You save more time writing code in a memory-safe language than you save in performance
- “Slower” memory-safe languages often have libraries that plug into fast, secure, C libraries anyway
  - Example: NumPy in Python (memory-safe)

The Real Reason: Legacy

- Most common actual reason: inertia and **legacy**
- Huge existing code bases are written in C, and building on existing code is easier than starting from scratch
  - If old code is written in {language}, new code will be written in {language}!

Example of Legacy Code: iPhones

- When Apple created the iPhone, they modified their existing OS and environment to run on a phone
- Although there may be very little code dating back to 1989 on your iPhone, many of the programming concepts remained!
- If you want to write apps on an iPhone, you still often use Objective C
- 2014: Swift, a new memory-safe language, introduced
- **Takeaway**: Non-memory-safe languages are still used for legacy reasons

Writing Memory-Safe Code

Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Writing Memory-Safe Code

- Defensive programming: Always add checks in your code just in case
  - Example: Always check a pointer is not null before dereferencing it, even if you're sure the pointer is going to be valid
  - Relies on programmer discipline
- Use safe libraries
  - Use functions that check bounds
  - Example: Use **`fgets`** instead of **`gets`**
  - Example: Use **`strncpy`** or **`strlcpy`** instead of **`strcpy`**
  - Example: Use **`snprintf`** instead of **`sprintf`**
  - Relies on programmer discipline or tools that check your program

Writing Memory-Safe Code

- Structure user input
  - Constrain how untrusted sources can interact with the system
  - Implement a reference monitor
  - Example: When asking a user to input their age, only allow digits (0–9) as inputs
- Reason carefully about your code
  - When writing code, define a set of *preconditions*, *postconditions*, and *invariants* that must be satisfied for the code to be memory-safe
  - Very tedious and rarely used in practice, so it's out of scope for this class

Building Secure Software

Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Approaches for Building Secure Software/Systems

- Run-time checks
  - Automatic bounds-checking
  - May involve performance overhead
  - Crash if the check fails
- Monitor code for run-time misbehavior
  - Example: Look for illegal calling sequences
  - Example: Your code never calls **`execve`**, but you notice that your code is executing **`execve`**
  - Probably too late by the time you detect it
- Contain potential damage
  - Example: Run system components in sandboxes or virtual machines (VMs)
  - Think about privilege separation

Approaches for Building Secure Software/Systems

- Bug-finding tools
  - Excellent resource, as long as there aren't too many false bugs
  - Too many false bugs = wasted programmer time
- Code review
  - Hiring someone to look over your code for memory safety errors
  - Can be very effective… but also expensive
- Vulnerability scanning
  - Probe your systems for known flaws
- Penetration testing ("pen-testing")
  - Pay someone to break into your system
  - Take notes on how they did it

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - We're testing for the *absence* of vulnerabilities
  - Normal inputs rarely reveal security vulnerabilities
- How can we test programs for memory safety vulnerabilities?
  - Fuzz testing: Random inputs
  - Use tools like Valgrind (tool for detecting memory leaks)
  - Test corner cases
- How do we tell if we've found a problem?
  - Look for a crash or other unexpected behavior
- How do we know that we've tested enough?
  - Hard to know, but code-coverage tools can help

Working Towards Secure Systems

- Modern software often imports lots of different libraries
  - Libraries are often updated with security patches
  - It's not enough to keep your own code secure: You also need to keep libraries updated with the latest security patches!
- What's hard about patching?
  - Can require restarting production systems
  - Can break crucial functionality
  - Management burden (the "patch treadmill" never stops)

Exploit Mitigations

Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Exploit Mitigations

- Scenario
  - Someone has just handed you a large, existing codebase
  - It’s not written in a memory-safe language, and it wasn’t written with memory safety in mind
  - How can you protect this code from exploits without having to completely rewrite it?
- **Exploit mitigations (code hardening)**: Compiler and runtime defenses that make common exploits harder
  - Find ways to turn attempted exploits into program crashes
  - Crashing is safer than exploitation: The attacker can crash our system, but at least they can’t execute arbitrary code
  - Mitigations are cheap (low overhead) but not free (some costs associated with them)

Exploit Mitigations

- Mitigations involve a large back-and-forth arms race
  - Security researchers find a new mitigation to make an exploit harder
  - Attackers find a way to defeat the mitigation
- Mitigations make attacks harder, but not impossible
  - The only way to prevent all buffer overflow attacks is to use a memory-safe language

Recall: 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
4. Return from the function
5. Begin executing malicious shellcode

Recall: 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
4. Return from the function
5. Begin executing malicious shellcode

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!