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

https://nosec.org/home/detail/4970.html
len是将要写入的内容，PAGE_SIZE是总的缓冲区长度，而size是已经写入的数据的长度。实际上，我们应该加上额外的2个字节，它们对应于开始位置的逗号和结束位置的null字节。

Yes, not a blue slide!

Format String Vulnerabilities

Textbook Chapter 3.3

Review: `printf` behavior

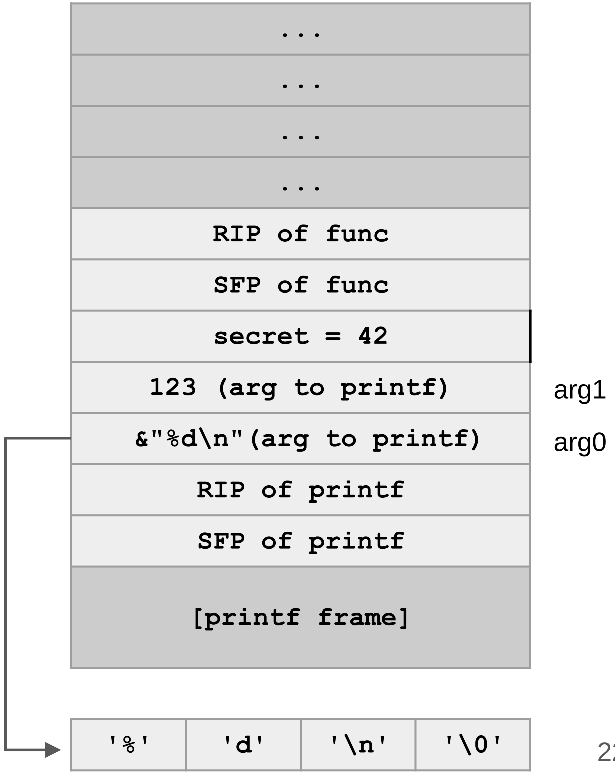
- Recall: `printf` takes in an variable number of arguments
  - How does it know how many arguments that it received?
  - It infers it from the first argument: the format string!
  - Example: `printf("One %s costs %d", fruit, price)`
  - What happens if the arguments are mismatched?

Review: `printf` behavior


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

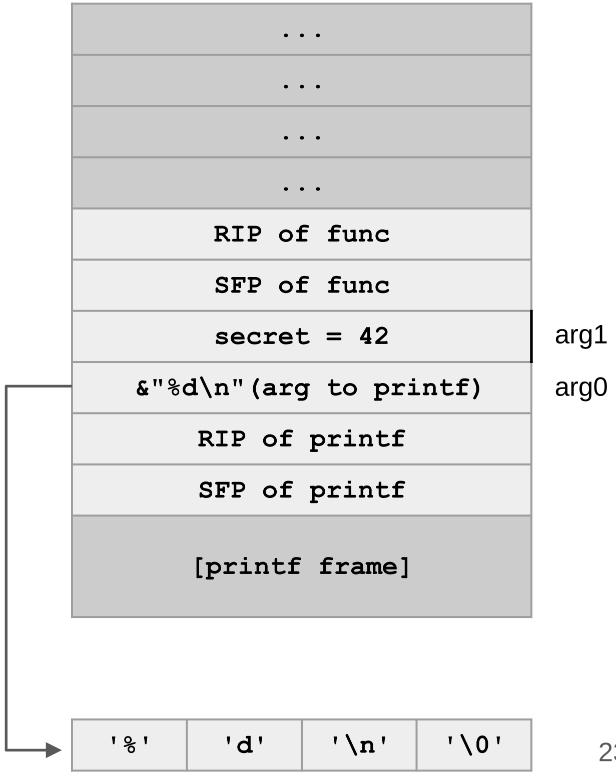
`printf` assumes that there is 1 more argument because there is one format sequence and will look 4 bytes up the stack for the argument

What if there is no argument?



Review: **`printf`** behavior


```c
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

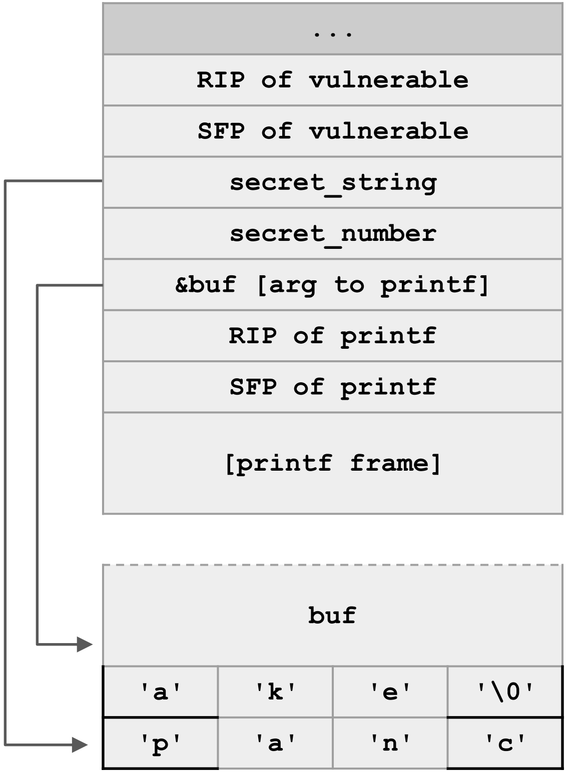
- Now, the attacker can specify any format string they want:
  - `printf("100% done!")`
    - Prints 4 bytes on the stack, 8 bytes above the RIP of **`printf`**
  - `printf("100% stopped.")`
    - Print the bytes **pointed to** by the address located 8 bytes above the RIP of **`printf`**, until the first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

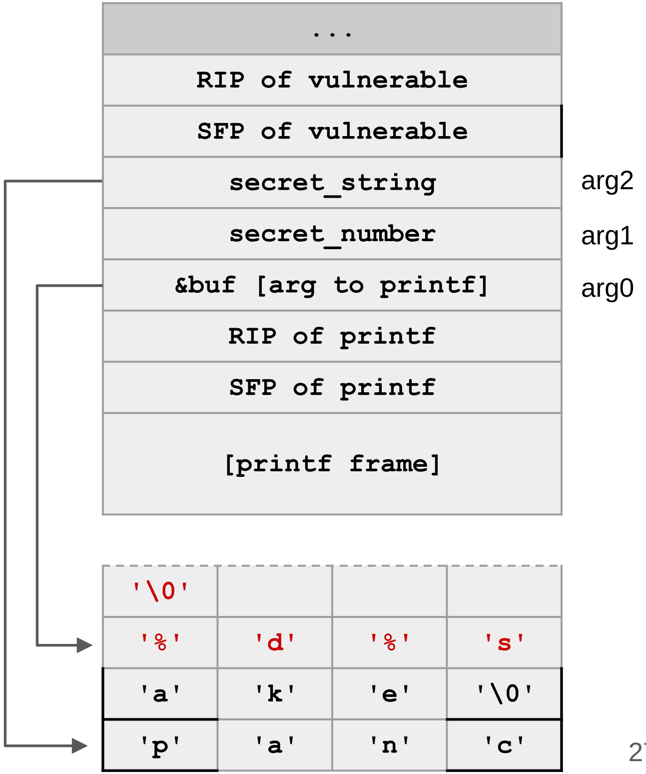


Format String Vulnerability Walkthrough

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);
}
```



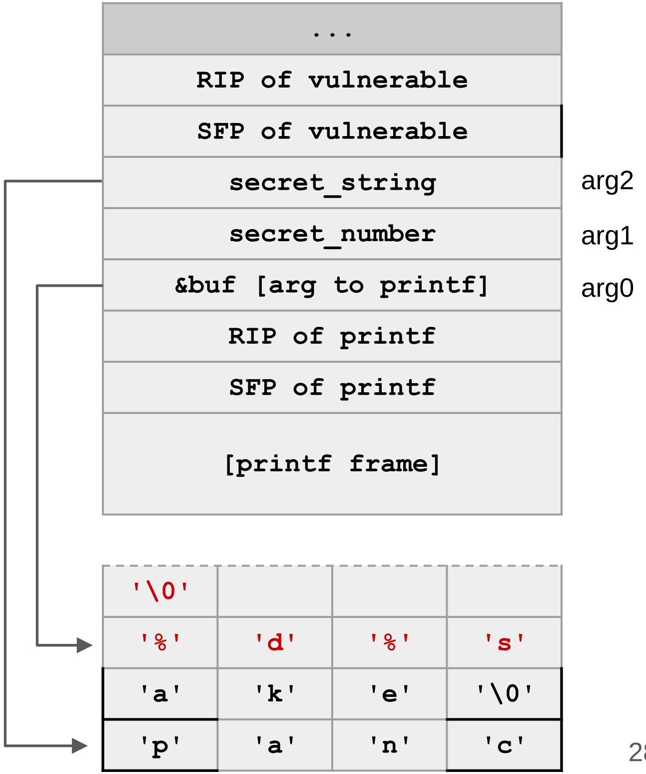
We're calling `printf("%d%s")`. `printf` reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

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);
}
```



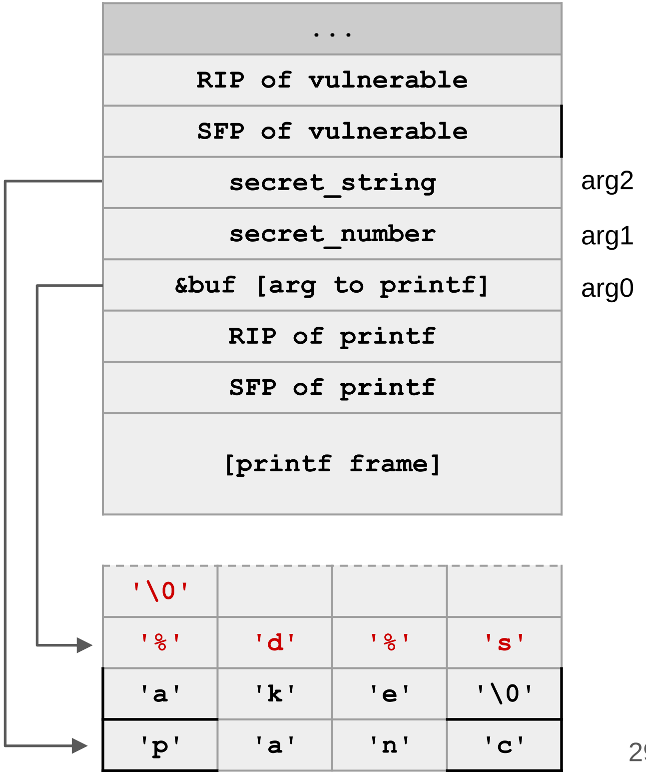
The first format specifier **%d** says to treat the next argument (arg1) as an integer and print it out.

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`**

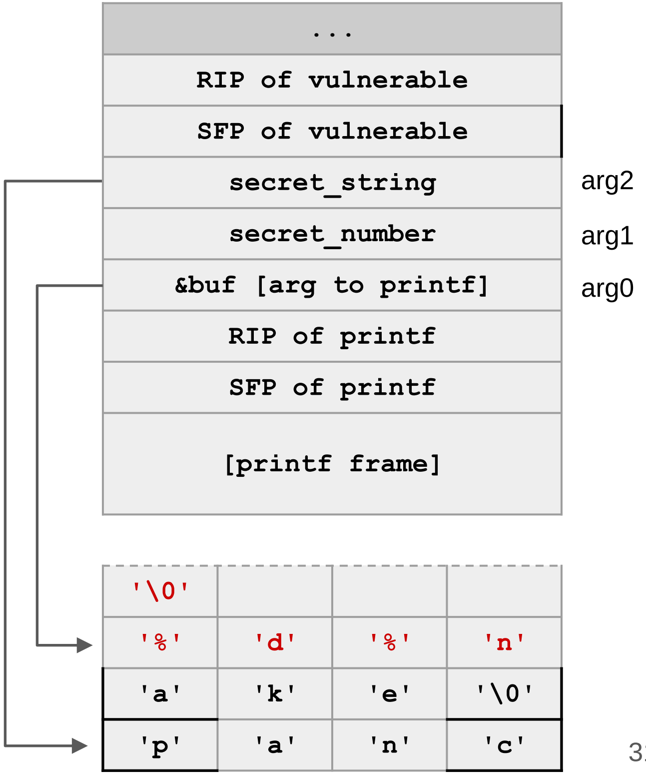
```
void vulnerable(void) {
    char buf[64];
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf(buf);
}
```

Format String 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);
}
```



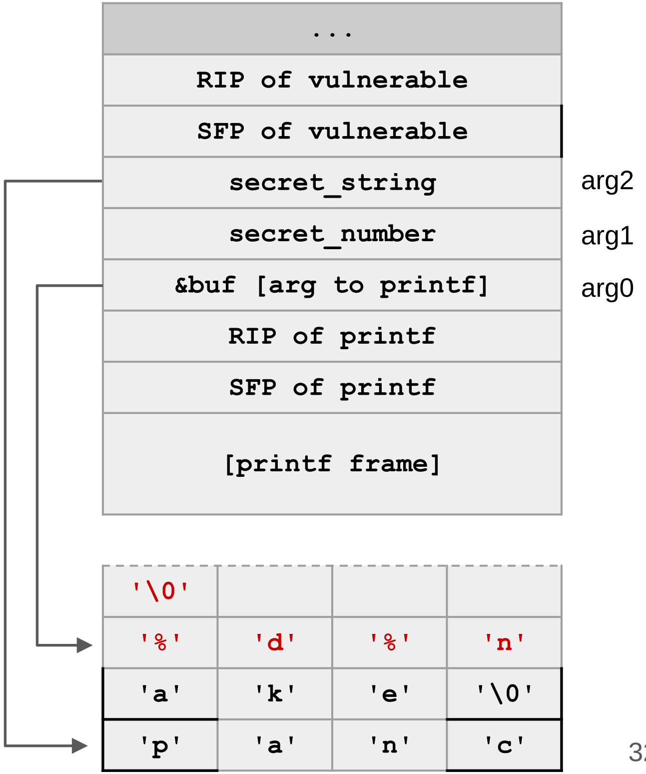
We're calling `printf("%d%n")`. `printf` reads its first argument (arg0), sees two format specifiers, and expects two more arguments (arg1 and arg2).

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);
}
```



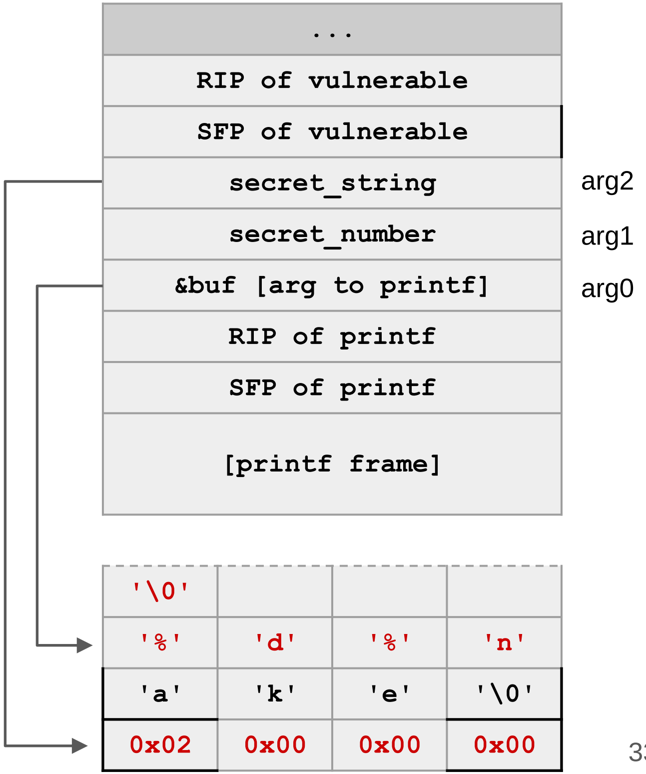
The first format specifier `%d` says to treat the next argument (arg1) as an integer and print it out.

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);
}
```



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!

Next: Memory Safety Mitigations

- Memory-safe languages
- Writing memory-safe code
- Building secure software
- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations

Today: Defending Against Memory Safety Vulnerabilities

- We’ve seen how widespread and dangerous memory safety vulnerabilities can be. Why do these vulnerabilities exist?
  - Programming languages aren’t designed well for security.
  - Programmers often aren’t security-aware.
  - Programmers write code without designing security in from the start.
  - Programmers are humans. Humans make mistakes.

Today: 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

Textbook Chapter 4.1

Today: 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.

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
- **Takeaway**: Non-memory-safe languages are still used for legacy reasons

Writing Memory-Safe Code

Textbook Chapter 4.2

Today: 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
  - 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

Textbook Chapter 4.3

Today: 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
- 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

Testing for Software Security Issues

- 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