Code Injection Attacks
- Overview - why do them?
- Buffer Overflows (on the stack)
  - Program memory layout
  - Stack frames
  - A simple overflow
  - & Spawning Shells
- Heap-based overflows
- Putting it together
- Defenses
- Other overflows: Heap-based, return-to-libc
- Format String Exploits
  - C format strings
  - Sketch of exploit
- XSS: Cross-site Scripting
- SQL Injection

Buffer overflows: a type of code injection
- Program accepts input and does not check it properly
- In this sense like XSS/SQL injection
- Buffer overflows work at a lower level, overwriting memory
Why Code Injection:
1. Cause (potentially advantageous) incorrect behavior
2. Gain system privileges (root)
3. Gain access to a system
4. Steal information (XSS and SQL-injection)

Buffer Overflows:
* Common exploit that takes advantage of the fact that C does not perform boundary checks on arrays.
* Also exploits the layout of the program in memory

Basic Program Layout in Memory

```
Low addresses (0x00000000) ───────────────────────────────────────────────────────
| Text                                      | stores program code |
| data                                      | store global variables |
| bss                                       |
| heap                                      | stores dynamically-allocated memory (e.g. via malloc) |
| Stack                                     | stores program execution context (function calls, local variables) |

high addr.  (0xfffffff)
```

- Heap and stack are dynamic — their sizes change as the program runs.
- Heap grows **up** toward higher addresses, the stack grows **down** toward lower addresses.
- Most common buffer overflow occurs on the stack.
A vulnerable program:

```c
void test(char *buf) {
    char flag;
    char buffer[10];
    strcpy(buffer, buf);
    return 0;
}
```

This tries to copy 100 characters into a 10 character buffer!

```c
void main() {
    char *buf = "AAAA...";  // "AAAA..." (100 'A's)
    test(buf);
}
```

- Running this program causes a segmentation fault. Why?

Look at the stack:

- We write 100 chars starting here, meaning all of these variables values are overwritten (since buffer is too small).

- We segfault because the return address was overwritten with 0x41414141 ("A" is 0x41 is ASCII), which is not in the virtual address space.

- This would be more dangerous (and realistic) if buf was filled with user input.
Stack Frames:
- Whenever a function is called in a C program, a stack frame is created and added to the stack.

Example C program:
```c
void test (int a, int b) {
    char foo [int flag];
    char buffer [10];
    ...
}

void main () {
    test (1, 2);
}
```

Overflowing Buffers:
- C doesn't boundary check arrays.
- Strings are character arrays terminated with a null (0) byte.
- Functions like `strcpy` copy bytes until they reach a null byte.
- Putting too much data into a buffer is the basic mechanism of the buffer overflows (hence "overflow")
**Uses of Buffer Overflows:**

1. Cause crashes (as we've seen)
2. Overwrite variables with new values
   (In the previous example, the value of flag was changed to 0x41414141)
3. Execute arbitrary code
   - IDEA: change return address to a new, valid value.
     A common location is the start of the buffer itself or an environment variable. Assembly code that is placed in the chosen location will be executed by the program.

The address of the buffer can be found using a debugger

**Shellcode:**

- Bytecode that opens up a shell. (Use the exec() system call to execute a shell process)
- Somewhat tricky to make - typically have to avoid null bytes since they terminate C strings
- Can be as small as 31 bytes
- Can be all ASCII printable characters.
Let's say our overflow example declared buf as

```c
char * buf = argv[1];
```

(meaning buf points to the 1st command line parameter).

Now what should we input to the program to cause an overflow?

**Crafting an input buffer:**

```
+-----------------------------+            +-----------------------------+
|    NOP                     |            |    NOP                     |
|    NOP                     |            |    NOP                     |
|    ...                    |            |
|    NOP                     |            |
+-----------------------------+            +-----------------------------+
| Shell code                 |            | Return address             |
| Return address             |            | Return address             |
| Return address             |            | Return address             |
+-----------------------------+            +-----------------------------+
```

NOPs (short for No Operation) do nothing. This "NOP sled" lets us miss the exact address of the buffer by a little bit.

The return address is the start address of the buffer. Repeating the address several times lets us miss the exact position of the return address by a little bit.

Ideally, when this is copied into buffer, we will overwrite the return address of the function call with the address of buffer. This will cause execution to jump to our custom code and spawn a shell.
Defense mechanisms:

- **ASLR** - prevent memory locations on the stack from being interpreted as code (requires hardware support)
  - **Stack Canaries**
    - Software emulation: Exec Shield (Linux), WAX (EDS), Software DEP (Win)
  - **Stack Guard** - prevent the attacker from overwriting the return address by detecting changes and terminating the program.

- **Address Space Randomization (ASLR)** - put the stack in a randomly chosen memory location so it's hard to guess the location of the buffer.
- **Safe functions** - use `strcpy` instead of `strncpy`, since `strncpy` lets you specify the maximum number of characters to copy
- **Non-Executable Stacks** - prevent memory locations on the stack from being interpreted as code 

Change stack frames to look like:

![Stack Frame Diagram]

- Use a type-safe language!

Note: Only using safe functions can prevent all buffer overflows. The other mechanisms mainly prevent the standard stack-based overflow that we saw earlier.
Heap Overflows: (or overflows in other program regions)
  * Possible, though harder to find since the heap layout is not as transparent as the stack layout
  * Can still execute arbitrary code by overwriting function pointers
  * Or just overwrite data...

Return-to-libc Attacks:
  * Libc is a standard library including functions like printf(), exec(), etc.
  * Basic idea: set up the stack to look like a function call to one (or more) functions in libc. It is possible to get a root shell by chaining several calls.
  * Exploit works on non-executable stacks.

A hypothetical attack to execute `system('/bin/sh')` and gain a shell using the vulnerable programs from before:

![Diagram of stack frame modification](image)
• The return address is now the address of the system function.

• The argument to system is the address of the string "/bin/bash" stored somewhere else in memory (e.g., in an environment variable).

• This will execute a shell, but it won't maintain the privileges of the executing program because system() drops privileges.

- Borrowed code chunks—jump into middle of function
- Return-oriented programming—jump into middle of byte sequence

**Format String Exploits!**

• Format strings are arguments to printf containing special characters escape sequences that begin with "%"

- If programmers call printf() incorrectly, we can cause all kinds of trouble: (We can write arbitrary memory locations)

**Notable Escape Sequences:**

• %x - print a value in hexadecimal

• %s - interpret the argument as a pointer then to a char buffer (a string). Print the string.

• %n - save the number of bytes written so far to the address location pointed to by the argument
Some printf Examples:

```c
char* foo = "abcd";
int a = 10;

printf("%x, %s %n", foo, foo, &a); \rightarrow prints the address of "abcd", then "abcd", then saves 13 in a (8 chars for the address, 1 space, 4 chars in "abcd").

printf([argu[1]]); \leftarrow the wrong way to print a string.
Note that escape characters in argu[1] will be interpreted by printf().

printf("%s", argu[1]); \leftarrow the right way to print a string.
```

- Format String Exploits occur when people use printf() to store incorrectly to print strings.
- By including escape characters in the string, (especially %on), we can write arbitrary addresses.
- The arguments for the escape sequences above are calculated by adding an offset to the stack pointer.

Normal printf call stack:

```
:SP \rightarrow
    SP
    Return address
    & Format string
    P_1
    P_2
    ... 
    P_n
```

The location of the i\text{th} parameter \( P_i \) is computed by adding to ESP, even if \( P_i \) wasn't provided in the call.

printf("%x"); \rightarrow prints some hexadecimal value
* If the format string is also allocated on the stack, we can control the arguments to the escape sequences as well.

**Example**

**A Vulnerable Program:** (ignore the buffer overflow...)

```c
void main (int argc, char **argv) {
    char buf[100];
    strcpy (buf, argv[1]);
    printf (buf);
}
```

* In the `printf()` call, the stack will look like:

```
SFP
return address
&buf
...
buf
```

Since `buf` is below the `printf` call, at some point `printf` will start using its contents as the arguments to the escape sequences. Relatively easy to find which escape sequence first reads its argument from `buf`.

* Can now use the `%n` sequence `&buf` and control its argument (the address to write) — can write to arbitrary memory locations, and set them to values of our choosing.
The exploit string:
* Say we figure out that the kth printf() argument really uses the first word of buffer as its argument.
* Say to write to <address>, our string looks like "<address> \%0x \%0x ... \%0x \0n"
  \( k-1 \) \"\%0x\"'s
* This writes something like 4 + 8(\( k-1 \)) to <address>,
  4 + 8(\( k-1 \)) is (probably) the length of the printed string.
* By using several \%0n's, we can write any value we want.

Cross-Site Scripting (XSS):
- Attack on websites to run some code on the client viewing the website
- Can be used to steal login information, cookies

Simple script (in PHP...)
```php
<html>
<body>
  Hi!
  <? echo $_GET["name"] ?>
</body>
</html>
```

script.php → Hi:
script.php ?name = bob → Hi, bob