Code Injection Attacks

6.857 spring 2009
Lecture By
Jayant Krishnamurthy
L20, 4/22/2009

- ☐ Overview - why do them?
- ☐ Buffer Overflows (on the stack)
  - ☐ Program memory layout
  - ☐ Stack frames
  - ☐ A simple overflow
  - ☐ & Spawning Shells
  - ~~☐ Heap-based overflows and other~~ ☐ 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

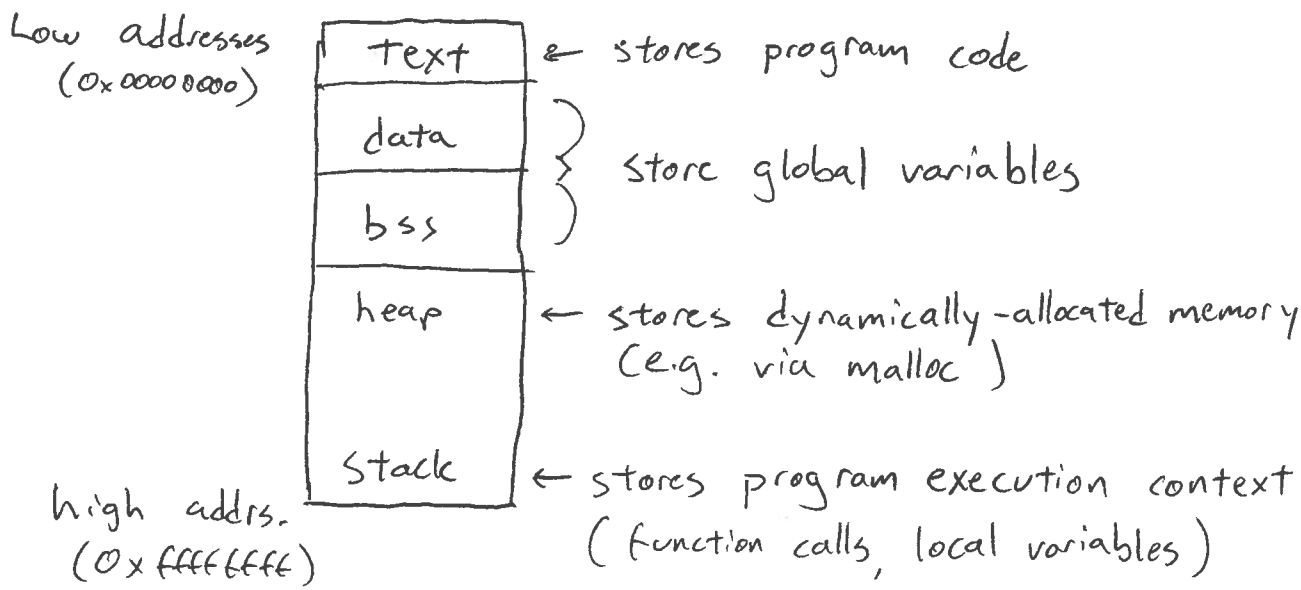
Why Code Injection:

1. Cause (potentially advantageous) incorrect behavior
2. Gain system priviledges (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



- Heap and stack are dynamic — their sizes change as the program runs.
- Heap grows <u>up</u> toward higher addresses, the stack grows <u>down</u> toward lower addresses
- Most common buffer overflow occurs on the stack

Stack Frames:

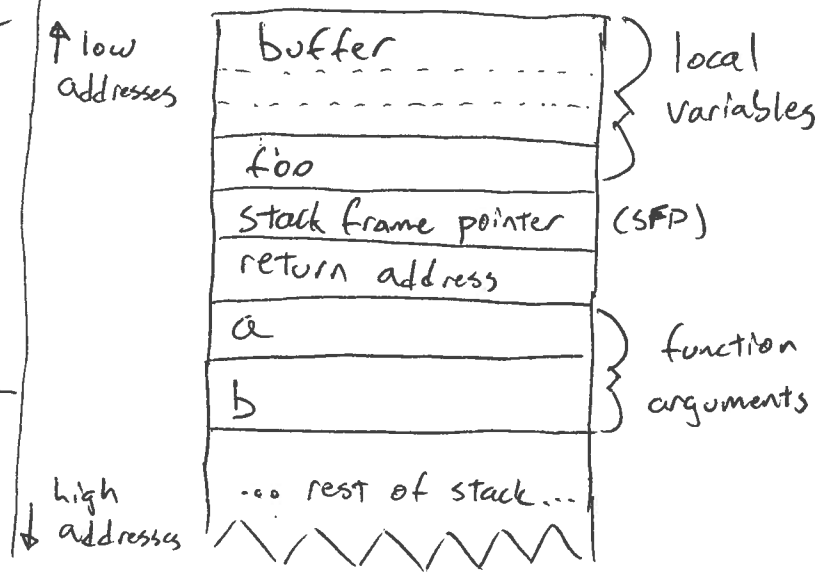
- whenever a function is called in a C program, a stack frame is created and added to the stack.

Example: C program:

```
void test(int a, int b) {
    char foo;
    char buffer[10];
}

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

The stack frame:



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")

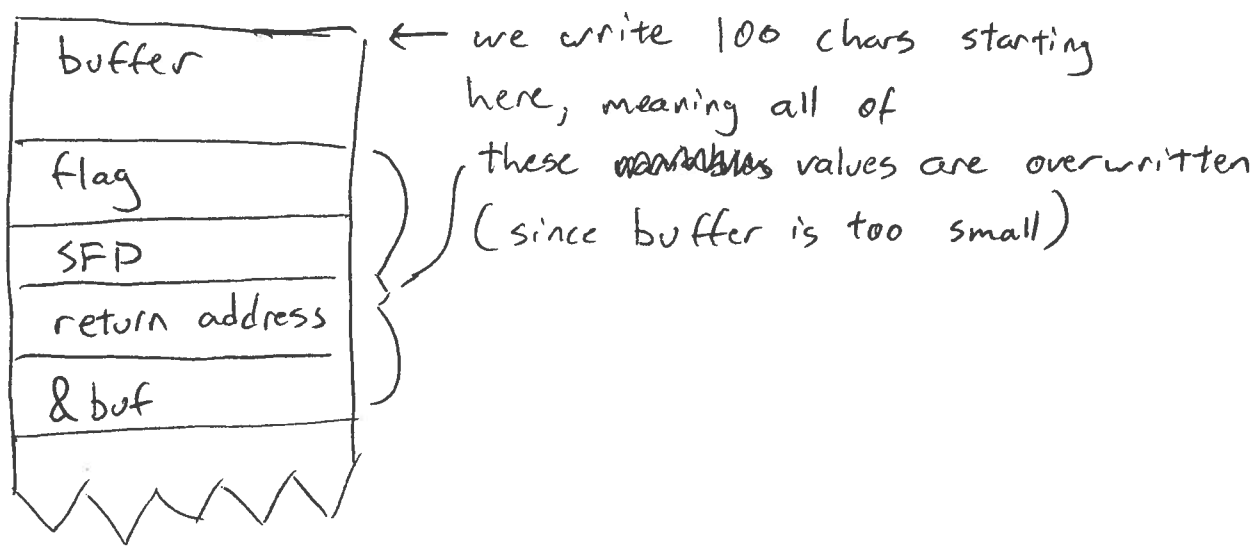
A vulnerable program:

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

void main() {
    char *buf = "AAA..." (100 "A"s)
    test(buf);
}
```

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

- Running this program causes a segmentation fault. Why? Look at the stack:



- 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.

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.

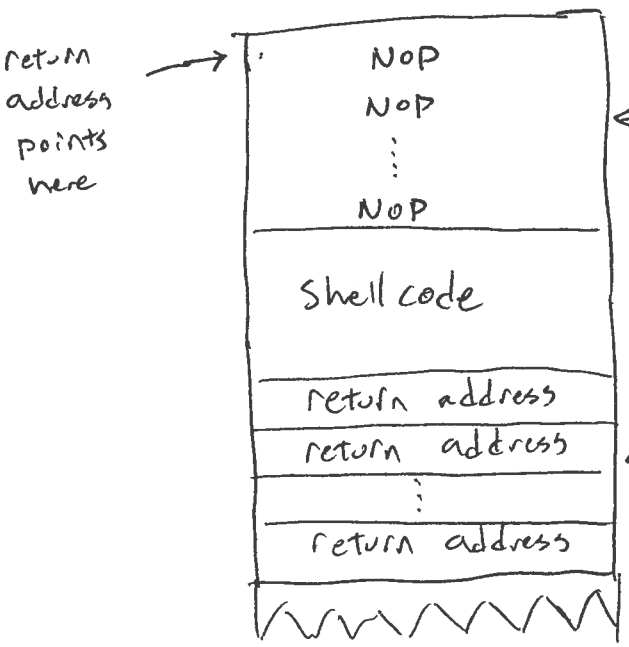
Let's say our overflow example declared buf as

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

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

~~Also~~ what should we input to the program to cause an overflow?

Crafting an input buffer:



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

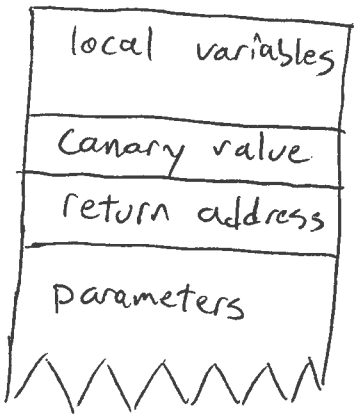
The return address is the start address of the buffer. Repeating the address several times lets us miss the exact position of the return address by a little bit.

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

Defense mechanisms:

- Address Space Randomization (ASR) - put the stack in a randomly chosen memory location so it's hard to guess the location of the buffer.
- Safe functions - use strncpy instead of strcpy, since strncpy ~~[illegible]~~ lets you specify the maximum number of characters to copy
- Non-Executable Stacks - prevent memory locations on the stack from being interpreted as code. (Requires hardware support)
- StackGuard - prevent the attacker from overwriting the return address by detecting changes and terminating the program.

Change stack frames to look like:



The canary value is chosen randomly when the program starts. Before the function returns, it checks to make sure that the canary is still the same. It is difficult (though not impossible) to overwrite the return address without changing the canary.

Note: Only using safe functions can prevent all buffer overflows. The other mechanisms mainly ~~target~~ prevent the standard stack-based overflow that we saw earlier.

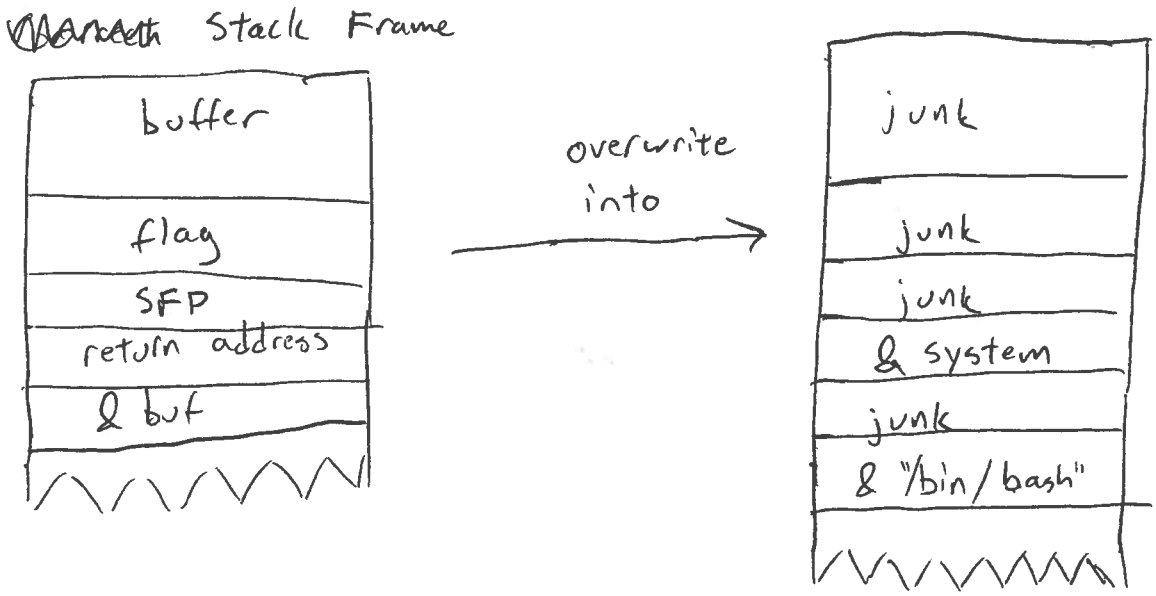
Heap Overflows: (or overflows in other program regions)

- 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~~ get a shell using the vulnerable program from before:



- The return address is now the address of the system function.
- The argument to system is the address ~~to~~ 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 priviledges of the executing program because system() drops priviledges.

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 ~~the~~ 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:

```
printf("%x", 16);
```
→ prints "10"

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

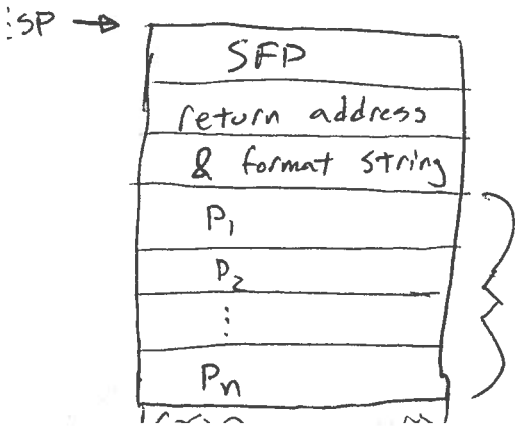
```
printf("%x, %s %n", foo, foo, &a);
```
→ prints the address of "abcd", then "abcd", then saves 13 in a (8 chars for the address, 1 space, 4 chars in "abcd").

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

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

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

Normal printf call stack:



The location of the ith parameter P_i is computed by adding to ESP, even if P_i wasn't provided in the call.

printf("%x"); → prints the hexadecimal value

- If the format string is also allocated on the stack, we can control the arguments to the escape sequences as well.

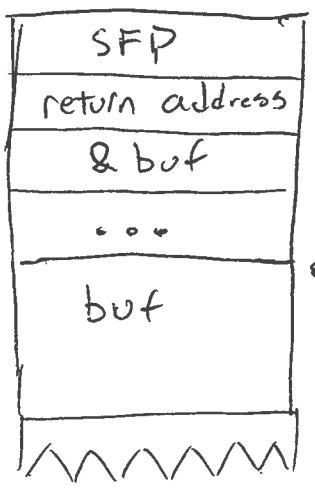
~~Example~~

A Vulnerable Program:

(ignore the potential buffer overflow...)

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

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



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 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~~ escape sequence ~~really~~ uses the first word of buffer as its argument.
- ~~Say~~ To write to <address>, our string looks like

$$\text{"<address>}\underbrace{\%x\ \%x\ \ldots\ \%x}_{k-1\ \text{"\%x"'s}}\ \%n\text{"}$$

- 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 %n'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...)

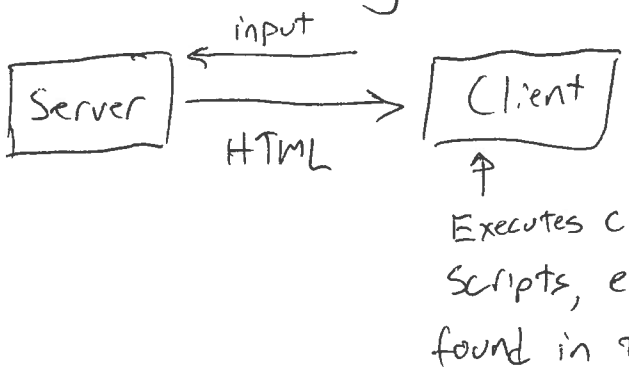
```
<html>
  <body> Hi
    <? echo $_GET["name"]; ?>
  </body>
</html>
```

Notes on next page →

script.php → Hi

script.php?name=bob → Hi bob

Cross-Site Scripting (XSS):



- XSS modifies the generated HTML to include extra scripts on the page seen by the client.
- These extra scripts can steal login information and cookies from the client and send them to an adversary.

A simple vulnerable program:

```
<html>
Hi, <? echo $_GET["name"]; ?>
</html>
```

GET parameters are specified in the url after the ? character

Sample Run

```
script.php?name=Jayant
-> <html>
     Hi, Jayant
   </html>
```

```
script.php?name=<script>alert("hello");</script>
-> <html>
     Hi, <script>alert("hello");</script>
   </html>
```

By setting the name parameter, I can include Javascript on the page.

- Useful for phishing attacks: the attacker crafts a URL, then sends it to other people. For example, if my bank had an XSS vulnerability, an attack could generate a URL that included a script to steal login information, then send the link out in fake emails from the bank. When clients click on the link and log in to their accounts, the adversary's script will run and send the client's login information to the attacker.

- XSS attacks can occur whenever user-controlled values are printed out as part of an HTML document.

Prevention:

- Escape all variables before printing them out
  - Replace < with < , etc.
- (Note that the problem here is analogous to the problem that enables SQL injection attacks.)

SQL Injection Attacks

```
SELECT * FROM users
WHERE username = '<username>'
and password = '<password>';
```

Program generates a SQL query by replacing <username> and <password> with specific values

Input:

username = admin

password = ' OR 't' = 't

```
=> SELECT * FROM users
   WHERE username = 'admin'
   AND password = '' OR 't' = 't';
```

user can log in as an admin without knowing the password!

More insidious:

password = '; DROP TABLE users;" ← Runs 2 SQL queries instead of 1, and the 2nd query deletes all users!

Prevention:

- Escape strings before putting them in SQL queries.
  - => Replaces ' with \', which is <u>not</u> interpreted as a '.
- Most languages have libraries for this.