Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

http://pwn.college
http://groups.google.com/group/cse-466
https://goo.gl/qgZF62

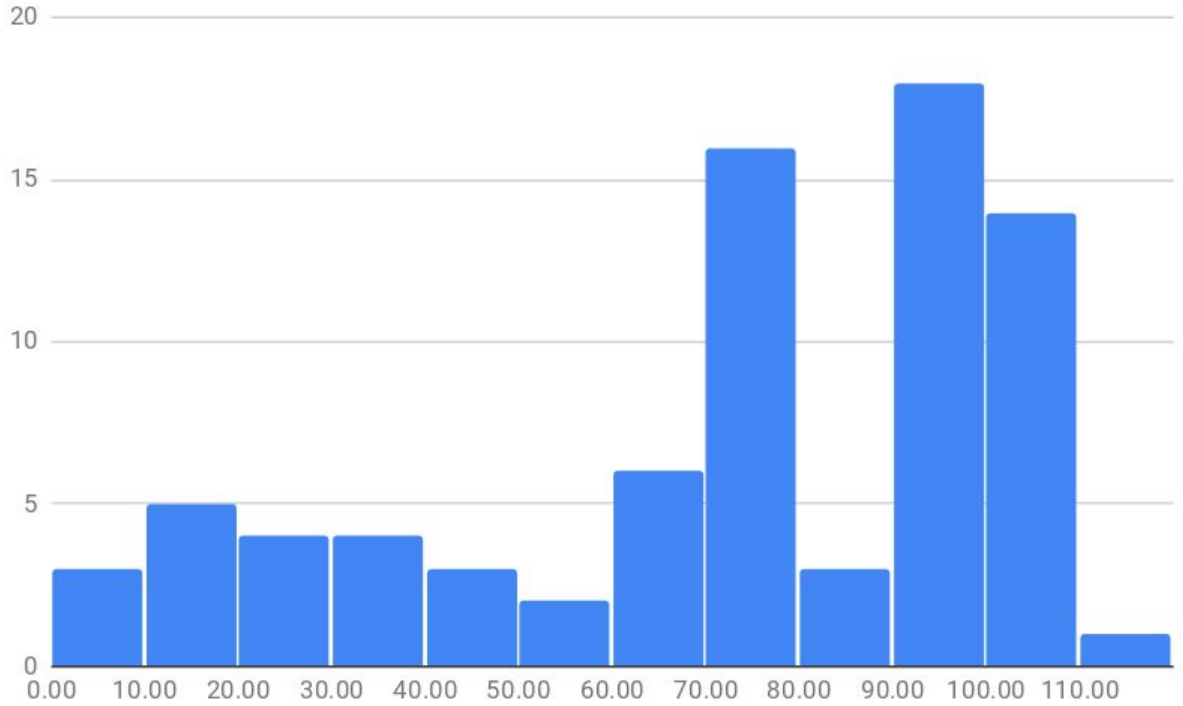
Homework 5 Retrospective

Statistics

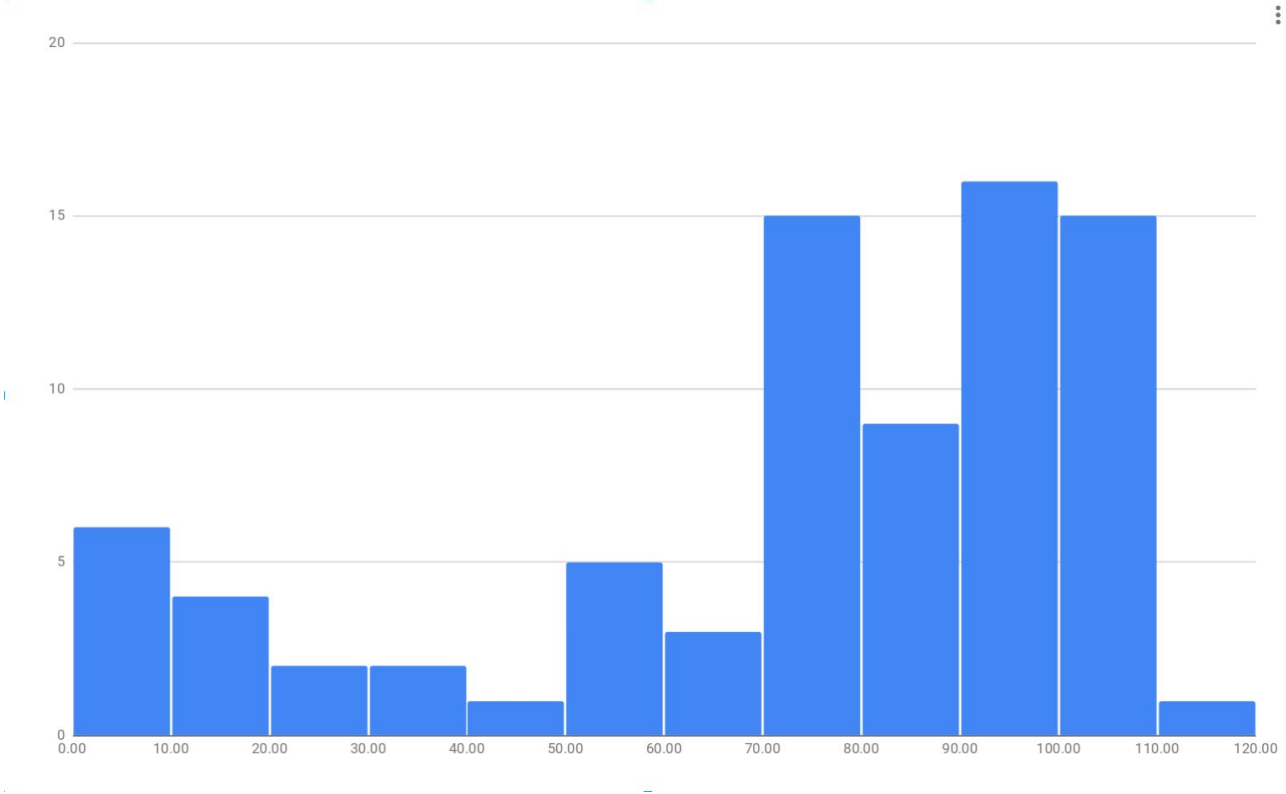
Total in course: 77 graded students (+1 auditing)
For hw:

26 attempted
21 A+
22 raised their course grade
12 seemingly waiting to withdraw

Student grade distribution



Student grade distribution



Discussion

How were the walkthroughs?

Did ssh significantly help with things?

CSE 466 Week 6

Shellcode Injection

The story so far...

In HW4 and HW5, we changed program behavior by overwriting certain data:

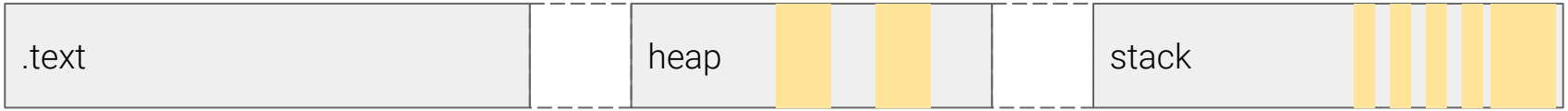
- In "amiga", "android", and babypwn lvl1, we overwrote the "win" variable.
- In others, we triggered the "win" function.

What if there was no handy "win" variable or function?

Memory Corruption

Programs start up with potentially user-influenced data already present:

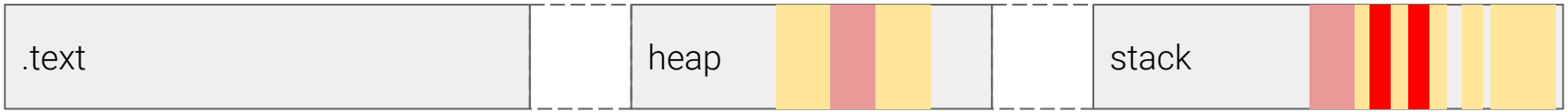
During execution, user data spreads through the program:



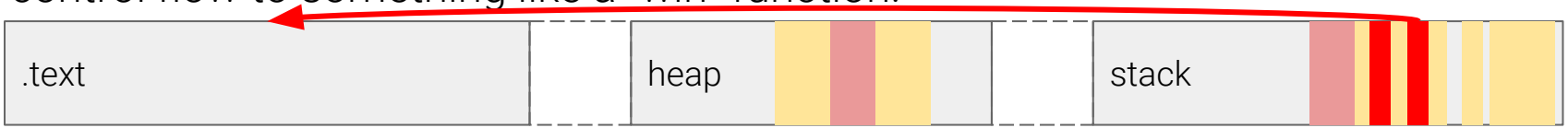
User data shouldn't directly control program execution. It is normally "non-control" data. However, it is stored *together* with "control" data.

Memory Corruption

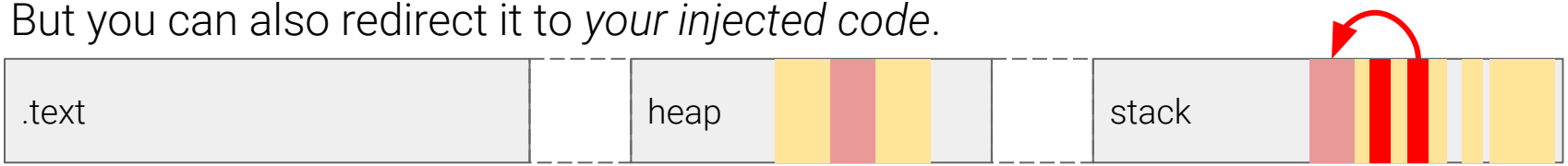
Memory corruption occurs when user-controlled data manages to spread into data that *shouldn't* be user controlled (i.e., through a buffer overflow).



If you overwrite *control data* (i.e., a return address), you can use this to redirect control flow to something like a "win" function.



But you can also redirect it to *your injected code*.



Memory corruption: ?

In the presence of memory corruption vulnerabilities, what can we corrupt?

1. Memory that doesn't influence anything. (Boring)
2. Memory that is used in a **value** to influence mathematical operations, conditional jumps, etc (such as the win variable).
3. Memory that is used as a **read pointer** (or offset), allowing us to force the program to access arbitrary memory.
4. Memory that is used as a **write pointer** (or offset), allowing us to force the program to overwrite arbitrary memory.
5. Memory that is used as a **code pointer** (or offset), allowing us to redirect program execution!

Typically, you use one or more vulnerabilities to achieve multiple of these effects.

Memory corruption: ✓

The takeaway is that once we corrupt memory, we can do some subset of:

1. Modify non-control data to our benefit ("android", "amiga", babypwn lvl1).
2. Modify control data to redirect execution to other parts of the program ("win" function).
3. Modify non-control data to inject our own code, and modify control data to redirect execution to it!

Today, we focus on #3.

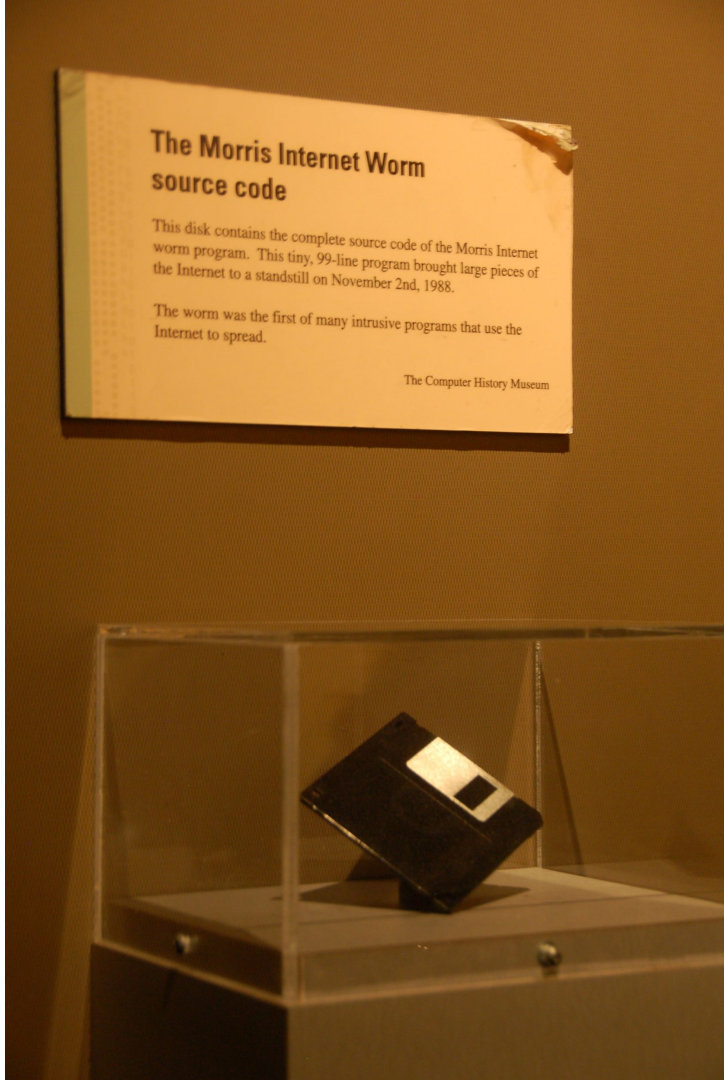
Code Injection

Code injection has a recorded history as old as buffer overflows, starting from the Morris worm.

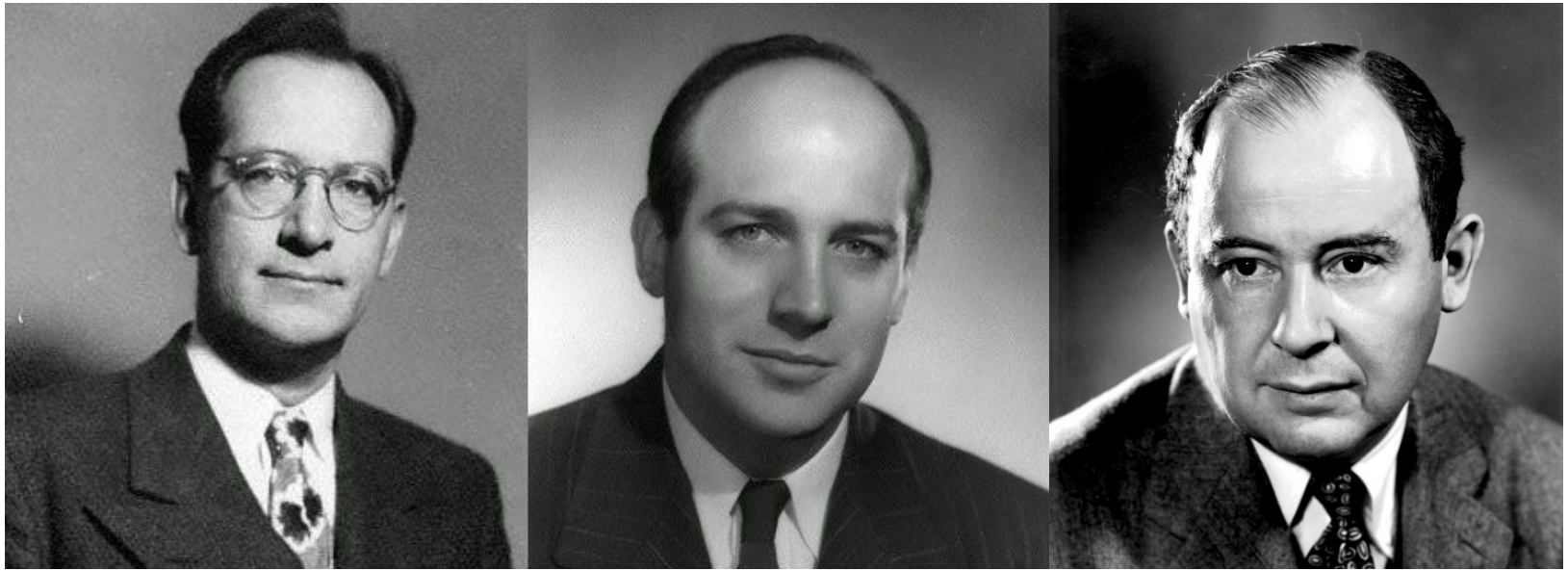
- Overflowed stack buffer in the **`fingerd`** service.
- No handy "win" function, so Morris injected his own!

(translated to x86_64):

```
mov rax, 59
mov rdi, "/bin/sh"
mov rsi, 0
mov rdx, 0
syscall
```



Why is this possible?



John Mauchly (Physicist), John Presper Eckert (Electrical Engineer), John Von Neumann (Mathematician)

John von Neumann, First Draft of a Report on the EDVAC, 1945.

Von Neumann Architecture vs Harvard Architecture

A Von Neumann architecture sees (and stores) code as data.

A Harvard architecture stores data and code separately.

Almost all general-purpose architectures (x86, ARM, MIPS, PPC, SPARC, etc) are Von Neumann.

Harvard architectures pop up in embedded use-cases (AVR, PIC).

Discussion: problems with viewing code as data?

Why "shell"code?

Usually, the goal of an exploit is to achieve arbitrary command execution.

The easiest way to do this is to launch a shell (i.e., "/bin/sh").

Thus: "shellcode"

Shellcode Injection!

Step 0: how to inject?

Step 1: where to inject?

Step 2: what to inject?

Step 3: inject!

Step 4: interpret results.

Shellcode Injection!

Step 0: how to inject?

Step 1: where to inject?

Step 2: what to inject?

Step 3: inject!

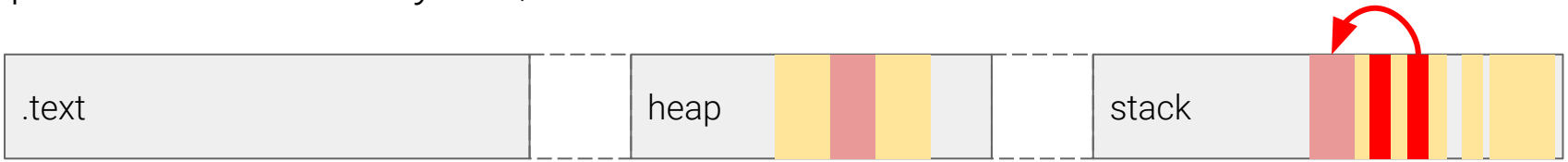
Step 4: interpret results.

Where can we inject shellcode?

Traditionally: anywhere!

In a "vintage" Von Neumann architecture, you can execute any memory!

The Morris worm deposited its payload on the stack, and similar patterns persisted for *over 20 years*, until ~2010.



This is a *stack-based inter-frame buffer overflow*, the most impactful memory corruption vulnerability in history.

The "No-eXecute" bit

Finally, computer architectures wised up!

Modern architectures support memory permissions:

- **PROT_READ** allows the process to read memory
- **PROT_WRITE** allows the process to write memory
- **PROT_EXEC** allows the process to execute memory

Intuition: *normally*, all code is located in .text segments of the loaded ELF files. There is no need to execute code located on the stack or in the heap.

By default in modern systems, the stack and the heap are *not* executable.

YOUR SHELLCODE NEEDS TO EXECUTE.

Game over?

The "No-eXecute" bit

The rise of NX *has* made shellcoding rarer.

It is now... an ancient art!

Remaining Injection Points - de-protecting memory

Vector 1: abuse the program to mprotect() memory.

Memory can be made executable using the mprotect() system call. So we can:

1. Trick the program into mprotect(PROT_EXEC)ing our shellcode.
2. Jump to the shellcode.

How do we do #1?

- Most common way is *code reuse* through *Return Oriented Programming*, where we stitch pieces of a program together to achieve our goal. We will cover this next week.
- Other cases are situational, depending on what the program is designed to do.

Remaining Injection Points - JIT

Vector 2: target a JIT.

- Just in Time compilers need to generate (and frequently re-generate) code that is executed.
- Pages must be writable for code generation.
- Pages must be executable for execution.
- Pages must be writable for code *re-generation*.

The safe thing to do would be to:

- mmap(PROT_READ|PROT_WRITE)
- write the code
- mprotect(PROT_READ|PROT_EXEC)
- execute
- mprotect(PROT_READ|PROT_WRITE)
- update code
- etc...

Obviously, not all JIT engines are safe...

Remaining Injection Points - JIT

Most JIT engines don't bother to mprotect(), but always have the pages executable.

Discussion: why is this?

Injection:

1. Corrupt a write pointer to point to the JIT page.
2. Write shellcode to the JIT page.
3. Corrupt a code pointer (such as a return address) to redirect execution into your shellcode.

Remaining Injection Points - JIT

What if the JIT safely mprotect()s its pages?

Shellcode injection technique: JIT spraying.

- Make constants in the code that will be JITed:

```
var evil = "%90%90%90%90%90";
```

- The JIT engine will mprotect(PROT_WRITE), compile the code into memory, then mprotect(PROT_EXEC). Your constant is now present in executable memory.
- Corrupt a code pointer to redirect execution into the constant.

If you can make many many constants, you can even mitigate the effects of Address Space Layout Randomization.

Remaining Injection Points - JIT

JIT is used *everywhere*: browsers, Java, and most interpreted language runtimes (luajit, pypi, etc), so this vector is very relevant.

Shellcode Injection!

Step 0: how to inject?

Step 1: where to inject?

Step 2: what to inject?

Step 3: inject!

Step 4: interpret results.

What do we inject?

Typically, we want to `execve("/bin/sh", NULL, NULL)`.

For our purposes, we can also `sendfile(1, open("/flag", NULL), 0, 1000)`.

How do we get "/bin/sh" or "/flag" into memory?

- option 1: we can write them in after the shellcode, then figure out the address (if we don't know it)
- option 2: we can programmatically create the strings in memory

Demo time!

Building Shellcode

First, write your shellcode as assembly:

```
_start:
.intel_syntax noprefix
          /* push '/flag\x00' */
          push 0x00000067616c662f
          /* call open(rsp, NULL) */
          mov rax, 2
          mov rdi, rsp
          mov rsi, 0
          syscall
          /* call sendfile(1, fd, 0, 1000) */
          mov rdi, 1
          mov rsi, rax
          mov rdx, 0
          mov r10, 1000
          mov rax, 40
          syscall
```

Then, assemble and extract it:

```
as shellcode.s -o shellcode.o
objcopy --dump-section .text=shellcode shellcode.o
```

Now, your shellcode is in the "shellcode" file! This method works great with cross-compilers to create shellcode for other architectures.

Testing Shellcode

To test your shellcode, build a shellcode loader:

```
page = mmap(0x1337000, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, 0, 0);
read(0, page, 0x1000);
((void(*)())page)();
```

Then `cat shellcode | ./tester`

Use a cross-compiler and qemu to test shellcode for other architectures.

Building Shellcode - Complications 1

Depending on how the program works, you might have to send shellcode matching some specific format:

Common conditions:

- no NULL bytes (i.e., shellcode injected via strcpy())
- no newlines or spaces (scanf())
- only printable characters
- many other situations exist (and will be encountered in HW6).

Building Shellcode - Complications 2

Your shellcode might be mangled beyond recognition.

Common situations:

- your shellcode might be *sorted*!
- your shellcode might be *compressed* or *uncompressed*.
- your shellcode might be *encrypted* or *decrypted*.
- any other mangling could be applied...

Getting around issues

1. Use jumps to jump over introduced garbage code.
    a. very useful for sorted shellcode.
2. Understand what instructions you *can* inject.
    a. useful library: *capstone* (apt install python-capstone)
3. Use an *unpacking* shellcode.
    a. if the page where your shellcode is writable, it can modify itself!
    b. this lets you use the few instructions you can to write other instructions and execute them.
4. Use a *multi-stage* shellcode.
    a. stage 1: read(0, rip, 1000).
    b. this overwrites your shellcode with unfiltered data!
    c. stage 2: whatever you want!
    d. a good stage-1 shellcode is *very short*, letting you get around complex shellcode requirements
    e. downside: you don't always have access to inject more shellcode...

Shellcode Injection!

Step 0: how to inject?

Step 1: where to inject?

Step 2: what to inject?

Step 3: inject!

Step 4: interpret results.

Shellcode Injection!

Step 0: how to inject?

Step 1: where to inject?

Step 2: what to inject?

Step 3: inject!

Step 4: interpret results.

Reaping the rewards

Normally, your shellcode will just give you a shell (or the flag).

What if there is no way to output data (i.e., `close(1); close(2);`)?

What other ways can you use to communicate the flag?

Homework 6

You are going to become shellcoding pros!

The homework:

- nc cse466.pwn.college 23, then ssh when the port is provided.
- **3 points** per challenge.
- Has a mix of reverse-engineering and pwnable challenges.
- Many walkthroughs.

First three levels up now, remaining levels will be pushed tonight (walkthroughs are time-consuming to write).