

# CS356: Shellcode and Countermeasures

W. Michael Petullo

University of Wisconsin–La Crosse

As of September 29, 2021





# Shellcode and Countermeasures: Exercise

- ❶ Get `https://www.flyn.org/courses/cs356-2021-fall/schedule/stack`.
- ❷ Get `https://www.flyn.org/courses/cs356-2021-fall/schedule/stack.c`.
- ❸ Draw a picture of the stack when this program is executing its `foo` function; assume the instruction pointer is set to one of the instructions related to the program calling `printf`.
- ❹ Indicate each stack byte's value.
- ❺ Place a label to the right of each stack word indicating the purpose of its bytes.
- ❻ Place a label to the left of ranges of bytes indicating
  - where you would write shellcode for a stack-smashing attack,
  - where you would overwrite the return address and with what value you would overwrite it, and
  - anything else in particular you would need to overwrite.

# Exploitation: AMD64 Call Stack

```
void foo(long c1, long c2, long c3, long c4,
         long c5, long c6, long c7) {
    long c8 = 8;
    bar(c7, c8); ← RIP
}

foo(1, 2, 3, 4, 5, 6, 7);
```

RDI: 1   RSI: 2   RDX: 3   RCX: 4   R8: 5   R9: 6

RSP	Low addr.	0x0000000000000000	Padding
		0x0000000000000008	Local c8
		0x00007fffffff00000000	Saved RBP
		0x0000000000004011bd	Return addr.
	High addr.	0x0000000000000007	Seventh arg.
		...	

- ▶ Stack grows from high to low addresses (down)
- ▶ First six args. in registers RDI, RSI, RDX, RCX, R8, and R9
- ▶ Subsequent args. on stack
- ▶ Return value on stack
- ▶ Local variables on stack (i.e., array bytes go towards return address)



# Return Fun: Shellcode Demonstration

Cause `service-shellcode` (sanitized) to terminate with a exit code of 42.

Important: GDB disables a feature called ASLR to aid in reproducibility. Your “live” code will need to make use of the leaked buffer and bp.

- 1 Determine stack layout using `gdb`.
- 2 Write shellcode in assembly (examples and `/usr/include/asm/unistd_64.h`).
- 3 Compile with `gcc -nostartfiles -nostdlib -o shellcode shellcode.S`, and view with `objdump`; gather shellcode bytes.
- 4 Run `./shellcode` to test.
- 5 Build: `shellcode | spacer | bp | &buffer` with `echo "\x48..." >exploit`

Important: your shellcode must make use of a string (input file path). Refer to the project instructions for why this causes a problem and to find a solution.



Questions about homework?

What if you don't know the address of a buffer on the stack?





# Shellcode and Countermeasures: NOP Sled

What if you don't know the address of a buffer on the stack? Guess!

```
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 SHELL CODE...
```

Jumping here is good enough for government work!





# Exploitation: Countermeasure: Stack Canaries (1998)

- 1 Generate a random value  $r$  when loading program.
- 2 Place  $r$  on stack at beginning of each function.
- 3 Check that  $r$  is unchanged before returning; crash if changed.

RSP	0x7fffffff d1e0	0x002144454b434148	buf bytes 7-0
	0x7fffffff d1e8	$r$	Stack canary
RBP	0x7fffffff d1f0	0x00007fffffff daf0	Saved RBP
	0x7fffffff d1f8	0x00000000004017d5	Return addr.
		...	

To compile with stack canaries: `gcc -fstack-protector -o foo foo.c`

```

1   mov     -0x8(%rbp),%rax
2   sub     %fs:0x28,%rax
3   je     0x401264 <read_a_bit+254>
4   callq  0x401030 <__stack_chk_fail@plt>

```





# Exploitation: Defeat: Stack Canaries (1998)

- 1 Generate a random value  $r$  when loading program.
- 2 Place  $r$  on stack at beginning of each function.
- 3 Check that  $r$  is unchanged before returning; crash if changed.

RSP	0x7fffffff d1e0	0x002144454b434148	buf bytes 7–0
	0x7fffffff d1e8	$r$	Stack canary
RBP	0x7fffffff d1f0	0x00007fffffff daf0	Saved RBP
	0x7fffffff d1f8	0x00000000004017d5	Return addr.
		...	

## Defeat:

- 1 Find a bug that leaks the canary (e.g., see Aquinas fsv).
- 2 Write back  $r$  during course of buffer overflow (see Aquinas canary).



# Shellcode and Countermeasures: Read Primitive

```
1 #include <stdio.h>
2 #include <stdlib.h>

4 int main(void)
5 {
6     printf(getenv("EDITOR"));
7 }
```

# Exploitation: Countermeasure: AMD64 NX Bit (2000)



- ➊ Add a no-execute bit to the kernel data structures that describe memory.
- ➋ Modify the processor to enforce the no-execute bit.
- ➌ Mark stacks and other regions as no-execute.

Kills 1990s-style shellcode.

To compile without no-execute bit protection (it is on by default):

```
gcc -zexecstack -o foo foo.c
```

# Exploitation: Countermeasure: AMD64 NX Bit (2000)



- ➊ Add a no-execute bit to the kernel data structures that describe memory.
- ➋ Modify the processor to enforce the no-execute bit.
- ➌ Mark stacks and other regions as no-execute.

Defeat:  
Use return-to-libc.

Defeat:

Use return-oriented programming (see Aquinas rop).

- ➊ Find useful bits of code that already exist in the program.
- ➋ Build a wild stack that drives the processor through the useful bits of code.

Useful code includes things like this:

- ▶ `pop rdi; ret`
- ▶ `pop rsi; ret`

Why?



# Exploitation: Countermeasure: ASLR (2001–2005)

- ① Generate a random value  $r$  when loading program.
- ② Layout program in memory at offset  $r$ .

Makes it difficult to guess addresses required by attacks, such as the address of a buffer or the address of a function you want to return into.

Globally disable ASLR (as root):

```
echo 0 >/proc/sys/kernel/randomize_va_space
```

Debuggers often disable ASLR:

```
[user@host]$ gdb foo
(gdb) show disable-randomization
Disabling randomization of debuggee's virtual address space on.
(gdb) set disable-randomization off
```



- ① Generate a random value  $r$  when loading program.
- ② Layout program in memory at offset  $r$ .

## Defeat:

- ① Find a bug that leaks the offset or a relative address.
- ② Dynamically use offset or relative addresses in exploit.



Graded Homework Aquinas: smash, syscall, and shellcode

Reading Read 0x321–0x359

<https://www.flyn.org/courses/cs356/schedule>