CS356: Shellcode

W. Michael Petullo

University of Wisconsin–La Crosse

As of April 5, 2022

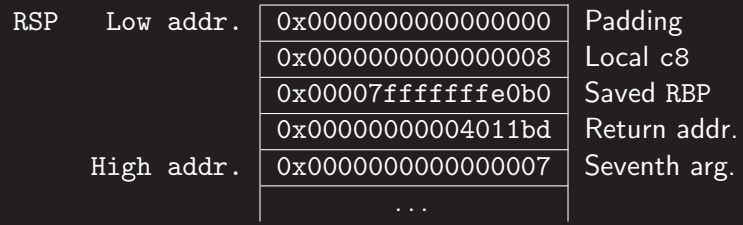
Questions about homework?

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



- 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)

Shellcode: Exercise

1. Get `https://www.flyn.org/courses/cs356-2022-fall/schedule/stack`.
2. Get `https://www.flyn.org/courses/cs356-2022-fall/schedule/stack.c`.
3. 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`.
4. Indicate each stack byte's value.
5. Place a label to the right of each stack word indicating the purpose of its bytes.
6. 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.

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: a full solution to the shellcode project needs to use a string input file path. Refer to the project instructions for why this causes a problem and to find a solution.

Board work: %rbp trick

Shellcode: NOP Sled

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

Shellcode: 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\x90SHELL CODE...
```

Jumping somewhere around here is good enough for government work!

SW Security 2: 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	0x7fffffffd1e0	0x002144454b434148	buf bytes 7–0
	0x7fffffffd1e8	r	Stack canary
RBP	0x7fffffffd1f0	0x00007fffffffdaf0	Saved RBP
	0x7fffffffd1f8	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>
```

Software Security 2: 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	0x7fffffffd1e0	0x002144454b434148	buf bytes 7–0
	0x7fffffffd1e8	r	Stack canary
RBP	0x7fffffffd1f0	0x00007fffffffdaf0	Saved RBP
	0x7fffffffd1f8	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: Read Primitive

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

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

SW 2: Countermeasure: AMD64 NX Bit (2000)

1. Add a no-execute bit to the kernel data structures that describe memory.
2. Modify the processor to enforce the no-execute bit.
3. 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
```

SW Security 2: Countermeasure: AMD64 NX Bit (2000)

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

Defeat:
Use return-oriented programming (see Aquinas rop).

1. Find useful bits of code that already exist in the program.
2. 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?

Defeat:
Use return-to-libc.
How to set parameter registers on AMD64?

Software Security 2: 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
```

Software Security 2: Countermeasure: ASLR (2001–2005)

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.

Assignments

Please refer to the course website for reading and homework assignments.

`http://www.flyn.org/courses/cs356-2022-fall/schedule`