CS356: Return Fun

W. Michael Petullo

University of Wisconsin–La Crosse

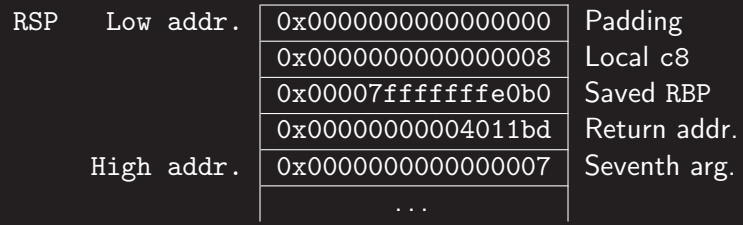
As of April 5, 2022

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

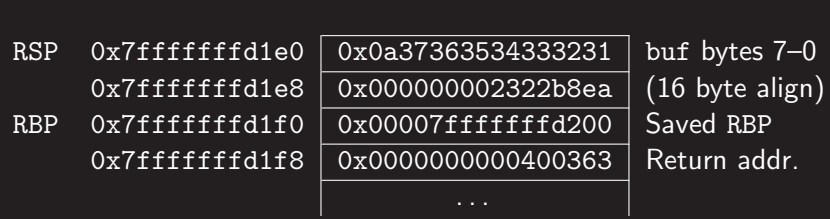


- Stack grows from high to low addresses (down)
- First six args. in registers RDI, RSI, RDX, RCX, R8, and R9
- Subsequent args. on stack
- Return value on stack
- Local variables on stack (i.e., array bytes go towards return address)

Review Aquinas' smash

Exploitation: service-smash.c's authenticate

```
        char buf[8];
...
        for (;;) {
                c = getchar();
                if (EOF == c) { break; }
                buf[i++] = c;
                if ('\n' == c) { break; }
        }
...     ⟵ RIP, after user entered "1234567"
```

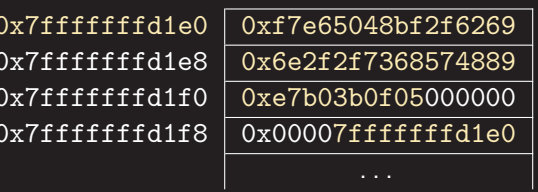


Review Aquinas' shellcode

Exploitation: Injecting Arbitrary Code, 1990s style

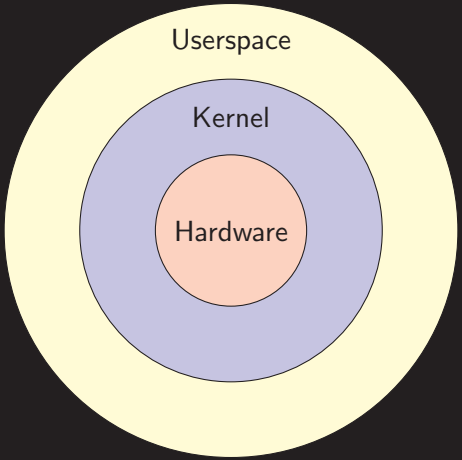
What if the program you attack does not have a `reveal` function? What if you want to execute code not already in the program?

1. Identify the address of `buffer` (sometimes this takes another bug).
2. Write instructions into `buffer` before overwriting return address.
3. Set return address to return into `buffer`.

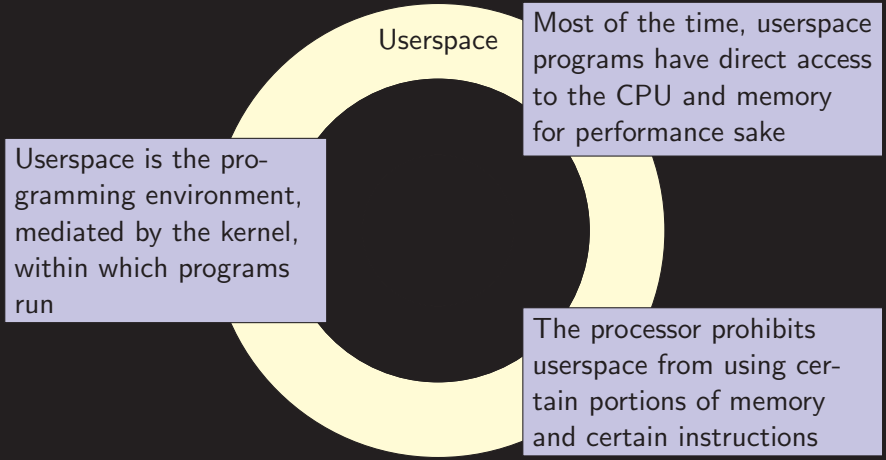


The instructions written are referred to as `shellcode` because a common aim is to execute a shell.

Return Fun: Just Enough OSes

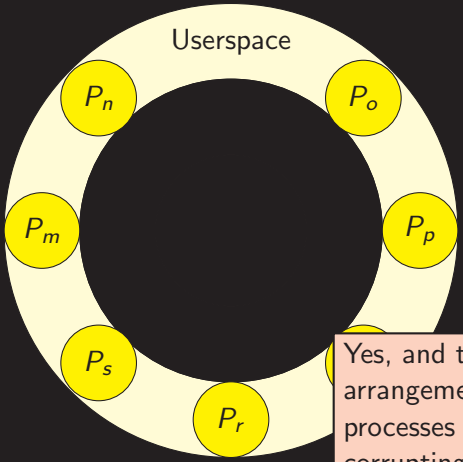


Return Fun: Just Enough OSes



Return Fun: Just Enough OSes

So easy even a Ph.D. can do it!

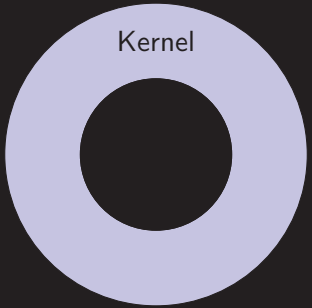


So, each process appears to have its own address space and direct access to most instructions? Wow!

Yes, and the arrangement prevents processes from corrupting each other or the system.

Return Fun: Just Enough OSes

That portion of software that runs in privileged mode



Collection of data structures used to manage processes as they make use of resources

Collection of syscalls that permit access to I/O

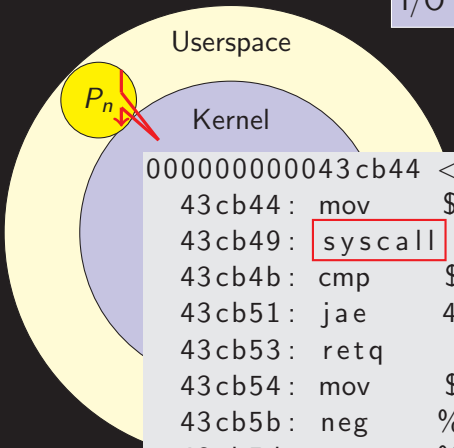
Collection of algorithms used to manage processes as they make use of resources

Return Fun: Just Enough OSes

Collection of syscalls that permit access to I/O

```
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    alarm(1);
}
```



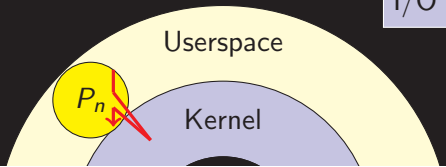

```
000000000043cb44 <alarm>:
  43cb44:  mov     $0x25,%eax
  43cb49:  syscall
  43cb4b:  cmp     $0xff...ff001,%rax
  43cb51:  jae     43cb54 <alarm+0x14>
  43cb53:  retq
  43cb54:  mov     $0xff...ffc0,%rcx
  43cb5b:  neg     %eax
  43cb5d:  mov     %eax,%fs:(%rcx)
  43cb60:  or      $0xff...ff,%rax
  43cb64:  retq
```

Return Fun: Just Enough OSes

Collection of syscalls that permit access to I/O

```
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
```



Intel 64 and ia-32 Architectures Software developer's Manual: Volume 2. 2016. p. 4-668

SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

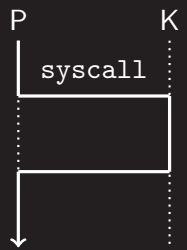
Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures

Return Fun: Just Enough OSes

1. Prepare arguments per convention
2. Set system call identifier (`/usr/include/asm/unistd_64.h`)
3. Invoke `syscall` instruction
   - Processor stores PC
   - Processor loads syscall handler into PC
   - Execution context set to privileged
   - Continue fetch-execute cycle:
4. Save additional process state
5. Check parameters (e.g., is `buf[0--`n] legal?)
6. Check permissions (e.g., may process access file descriptor n?)
7. Satisfy system call
8. Restore process state and set return value per convention
9. Return to unprivileged context and restore PC

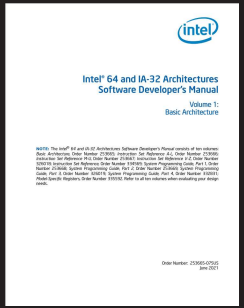


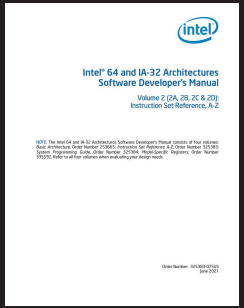
Exploitation: AMD64 Assembly Primer (Aquinas syscall)

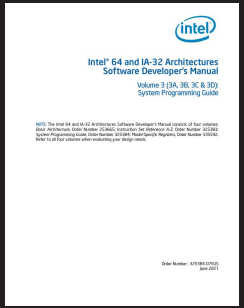
Fragment	Meaning
`hello:`	Create a string with contents
`.string "A string\n"`	`"A string\n"` and label with hello.
`.globl _start`	Declare the entry point label `_start`
`_start:`	as global, and define (like C's `main`).
`mov $1, %rax`	Move the literal value 1 (`write`) into register `rax`.
`mov $1, %rdi`	Move the literal value 1 (`stdout`) into register `rdi`.
`mov $hello, %rsi`	Move the address of the string `hello` into register `rsi`.
`mov $9, %rdx`	Move length of `hello` into register `rdx`.
`syscall`	Invoke the system call instruction.

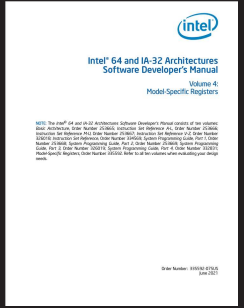
To compile example.S: `gcc -nostartfiles -nostdlib -o example example.S`

Return Fun: Intel Manuals









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.

Assignments

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

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