Assembly Crash Course

System Calls

Yan Shoshitaishvili Arizona State University

Having Effects

exit();

How do we interact with the outside world?

Even something as simple as quitting the program?

System Calls

Remember system calls? It's an instruction that makes a *call* into the Operating System. syscall triggers the system call specified by the value in rax. arguments in rdi, rsi, rdx, r10, r8, and r9 return value in rax

Reading 100 bytes from stdin to the stack:

```
n = read(0, buf, 100);
mov rdi, 0 # the stdin file descriptor
mov rsi, rsp # read the data onto the stack
mov rdx, 100 # the number of bytes to read
mov rax, 0 # system call number of read()
syscall # do the system call
```

read returns the number of bytes read via rax, so we can easily write them out:

write(1, buf, n);

```
mov rdi, 1 # the stdout file descriptor
mov rsi, rsp # write the data from the stack
mov rdx, rax # the number of bytes to write (same as what we read in)
mov rax, 1 # system call number of write()
syscall # do the system call
```

System Calls

System calls have very well-defined interfaces that very rarely change.

There are over 300 system calls in Linux. Here are some examples:

int open(const char *pathname, int flags) - returns a file new file descriptor of the open file (also shows up in
/proc/self/fd!)

ssize_t read(int fd, void *buf, size_t count) - reads data from the file descriptor
ssize_t write(int fd, void *buf, size_t count) - writes data to the file descriptor
pid_t fork() - forks off an identical child process. Returns 0 if you're the child and the PID of the child if you're the
parent.

int execve(const char *filename, char **argv, char **envp) - replaces your process.
pid t wait(int *wstatus) - wait child termination, return its PID, write its status into *wstatus.

Look familiar?

"String" Arguments

Some system calls take "string" arguments (for example, file paths).

A string is a bunch of contiguous bytes in memory, followed by a **0** byte.

Let's build a file path for open() on the stack:

```
mov BYTE PTR [rsp+0], '/' # write the ASCII value of / onto the stack

mov BYTE PTR [rsp+1], 'f'

mov BYTE PTR [rsp+2], 'l'

mov BYTE PTR [rsp+3], 'a'

mov BYTE PTR [rsp+4], 'g'

mov BYTE PTR [rsp+5], 0 # write the 0 byte that will terminate our string
```

Now, we can open() the /flag file!

```
mov rdi, rsp # read the data onto the stack
mov rsi, 0 # open the file read only (more on this later)
mov rax, 2 # system call number of open()
syscall # do the system call
```

open() returns the file descriptor number in rax

The argument <u>flags</u> must include one of the following <u>access</u> <u>modes</u>: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

Some system calls require archaic "constants".

Example: open() has a flags argument to determine how the file will be opened.

We can figure out the values of these arguments in C!

```
#include <stdio.h>
#include <fcntl.h>
int main() {
  printf("O_RDONLY is: %d\n", O_RDONLY);
}
```

```
yans@ramoth ~/pwn $ ./print_rdonly
0_RDONLY is: 0
yans@ramoth ~/pwn $
```

Quitting The Program

Finally, we can quit!

```
mov rdi, 42 # our program's return code (e.g., for bash scripts)
mov rax, 60 # system call number of exit()
syscall # do the system call
```

Goodbye, world!