SysV AMD64 ABI Example

```
int auth( const char * user) {
    size_t i;
    char buf[16];
    strncpy(buf, user, sizeof (buf));

auth:
    push rbp                       ; save previous frame pointer
    mov rbp, rsp                   ; set new frame pointer
    sub rsp, 0x30                  ; allocate space for locals (i, buf)
    movabs rdx, 0x10               ; move sizeof(buf) to rdx
    lea rax, [rbp-0x20]            ; get the address of buf on the stack
    mov qword [rbp-0x08], rdi      ; move user pointer into stack
    mov rsi, qword [rbp-0x08]      ; move user pointer back into rsi
    mov rdi, rax                   ; move buf into rdi
    call strncpy                   ; call strncpy(rdi, rsi, rdx)
    ...
```

Writing in Assembly

```
> yasm -- version
yasm 1.3.0
```

- Ok, enough review, let's write a program
- In keeping with the slides, we'll use an Intel syntax assembler called yasm
  - nasm is equivalent for this course
  - Feel free to use gas if you can't stand Intel syntax
- Let's write the simplest possible program
  - Immediately exit with status code 0

Hello World in Assembly

```
bits 64              ; we are writing a 64-bit program
section .text        ; we will place this in the .text (code) section

extern _exit         ; we are referencing an external function (exit)
                     ; libc functions are prefixed with '_'

global _start        ; declare _start as global symbol
                     ; this preserves a symbol table entry
_start:              ; _start is the default ELF entry point
  mov rdi, 0x00 ; zero out rdi, our first argument
  call _exit         ; call exit(rdi=0)
  int3               ; raise a breakpoint trap if we get here
                     ; (we should never get here)
```

Assembling and Linking

```
> yasm -f elf64 -o exit.o exit.asm
> ld -o exit exit.o -lc
> /lib/ld-2.18.so ./exit
> echo $?
0
```

1. We first assemble the program to an object file exit.o
2. We link an ELF exe against libc
3. We run it using a given runtime loader
   - You might need to specify a different path
   - Or, you might not need to specify it on your system
4. It returns 0!

Disassembly

- **Disassembling is the process of recovering assembly from machine code**
  - Not to be confused with decompilation!
  - Requires knowledge of binary format and ISA
- **Distinction between linear sweep and recursive descent disassembly**
  - Linear sweep begins at an address and continues sequentially until the buffer is exhausted
  - Recursive descent disassembly begins at an address and follows program control flow, discovering all reachable code

Disassembly with objdump

```
> objdump -d -M intel exit
exit:     file format elf64-x86-64

...

Disassembly of section .text:

0000000000400230 <_start>:
  400230:  48 c7 c7 00 00 00 00  mov    rdi,0x0
  400237:  e8 e4 ff ff ff        call   400220 <_exit@plt>
  40023c:  cc                    int3
```

Dumping ELF Objects

```
> objdump -x exit
exit:     file format elf64-x86-64
[ … ]
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400230

Program Header:
[ … ]
Dynamic Section:
  NEEDED                libc.so.6
[ … ]
Sections:
Idx Name Size VMA LMA File off  Algn
[ … ]
SYMBOL TABLE:
[ ... ]
0000000000000000       F *UND* 0000000000000000              _exit@@GLIBC_2.2.5
[ … ]
```

Invoking Syscalls

- **We can, of course, bypass libc and directly ask the kernel for services**
  - Need to know the syscall number (index into syscall table in the kernel)
- **On Linux/x86_64, we use the *`syscall`* instruction to transfer control to the kernel**
  - On linux/x86, traditionally INT 0x80 (interrupt 128)
- **Let's write a program to print a message and exit cleanly**

Invoking Syscalls

```
bits 64                          ; as before
section .text

global _start

_start:
    mov rdx, msg_len             ; len(msg) to rdx
    mov rsi, msg                 ; msg to rsi
    mov rdi, 1                   ; fd 1 (stdout) to rdi
    mov rax, 1                   ; write is syscall 1
    syscall                      ; call write(rdi, rsi, rdx)

    mov rdi, 0                   ; status code 0 to rdi
    mov rax, 60                  ; exit is syscall 60
    syscall                      ; call exit(rdi)

section .data                    ; the program's .data section
msg:        db 'aha',0x0a        ; the message as a byte array
msg_len:    equ $-msg            ; len is current addr - msg
```

Invoking Syscalls

```
> yasm -f elf64 -o hello.o hello.asm
> ld -o hello hello.o
> ./hello
aha
> echo $?
0
```

- You should see something similar to the above
  - Notice we didn't link against libc – no need
  - And, we do not need to invoke the runtime loader

Debugging Programs

Sometimes your program (or exploit) is buggy

- Well, mine are

An interactive debugger is an invaluable tool in these cases

- Start, stop execution
- Set breakpoints, watchpoints
- Directly inspect memory and CPU state
- Modify program state

Debugging binary programs? gdb!

GDB Quick Reference Card

GDB QUICK REFERENCE GDB Version 4

Essential Commands

`gdb` *program* [*core*]	debug *program* [using coredump *core*]
`b` [*file*`:`]*function*	set breakpoint at *function* [in *file*]
`run` [*arglist*]	start your program [with *arglist*]
`bt`	backtrace: display program stack
`p` *expr*	display the value of an expression
`c`	continue running your program
`n`	next line, stepping over function calls
`s`	next line, stepping into function calls

Starting GDB

`gdb`	start GDB, with no debugging files
`gdb` *program*	begin debugging *program*
`gdb` *program core*	debug coredump *core* produced by *program*
`gdb --help`	describe command line options

Stopping GDB

`quit`	exit GDB; also `q` or `EOF` (eg `C-d`)
`INTERRUPT`	(eg `C-c`) terminate current command, or

Break

```
break [fi
b [file:]l
break [fi
break +
break -
break *
break
break ..
cond n [
tbreak .
rbreak r
watch ex
catch ev
info bre
info wat
clear
clear [fi
clear [f
```

Get it! from the web!

Initializing GDB

```
> cat .gdbinit
set disassembly-flavor intel
disp/i $rip
```

Initial setup

- Set default disassembly syntax
- Display current instruction at each prompt

Also useful for

- Setting breakpoints
- Scripting execution of program to known bad state
- etc.

Let's debug our hello world program from before

Starting GDB

```
> gdb hello
GNU gdb (Debian 7.7.1+dfsg-3) 7.7.1
[ ... ]
(gdb) b _start
Breakpoint 1 at 0x4000b0
```

- **We load the program in gdb and set a breakpoint at the entry point (_start)**
- **Breakpoints insert (by default) a software interrupt (`int3`) at the given address**
- **When `int3` executes, control transfers to gdb**
  - Replaces the original instruction
  - After original instruction is executed, the `int3` is restored

Running the Program

```
(gdb) r
Starting program: [...]/hello

Breakpoint 1, 0x00000000004000b0 in _start ()
2: x/i $rip
=> 0x4000b0 <_start>:    movabs rdx,0x4
(gdb)
```

- We run the program and immediately hit our breakpoint
- Our display command prints the current instruction

Single-Stepping

```
(gdb) si
0x00000000004000ba in _start ()
2: x/i $rip
=> 0x4000ba <_start+10>:    mov    rsi,0x6000e4
(gdb)
0x00000000004000c1 in _start ()
2: x/i $rip
=> 0x4000c1 <_start+17>:    mov    rdi,0x1
(gdb)
0x00000000004000c8 in _start ()
2: x/i $rip
=> 0x4000c8 <_start+24>:    mov    rax,0x1
(gdb)
0x00000000004000cf in _start ()
2: x/i $rip
=> 0x4000cf <_start+31>:    syscall
```

- We can single-step the program by issuing `si`

Inspecting State

```
(gdb) p $rax
$1 = 1
(gdb) p $rsi
$2 = 6291684
(gdb) p/x $rsi
$3 = 0x6000e4
(gdb) x/s $rsi
0x6000e4:   "aha\n"
```

- p (print) prints register contents
- x (examine) dereferences addresses
- Formatting suffixes control how values are interpreted
  - /x for hex
  - /s for null-terminated strings

More Inspection

```
(gdb) x/8xb $rsp
0x7fffffffe790: 0x01    0x00    0x00    0x00    0x00    0x00
0x00    0x00
(gdb) x/8xg $rsp
0x7fffffffe790: 0x0000000000000001  0x00007fffffffea5f
0x7fffffffe7a0: 0x0000000000000000  0x00007fffffffea9f
0x7fffffffe7b0: 0x00007fffffffeaaa  0x00007fffffffeace
0x7fffffffe7c0: 0x00007fffffffeadf  0x00007fffffffeaf2
```

- We can print the stack by referencing $rsp
- Repetition suffix (above, 8)
- Additional width suffix
  - b for bytes
  - w for dwords
  - g for qwords

Dumping CPU State

```
(gdb) info registers
rax            0x1   1
rbx            0x0   0
rcx            0x0   0
rdx            0x4   4
rsi            0x6000e4 6291684
rdi            0x1   1
rbp            0x0   0x0
rsp            0x7fffffffe790   0x7fffffffe790
r8             0x0   0
[ … ]
r15            0x0   0
rip            0x4000cf 0x4000cf <_start+31>
eflags         0x202    [ IF ]
cs             0x33 51
ss             0x2b 43
[ … ]
```

- We can dump the entire (visible) CPU state

Mutating State

```
(gdb) set $r10=0x31337
(gdb) p/x $r10
$1 = 0x31337
(gdb) x/s $rsi
0x6000e4: "aha\n"
(gdb) set *0x6000e4=0x0a485542
(gdb) x/s $rsi
0x6000e4: "BUH\n"
```

- We can set values both in registers and in memory

Continuing Execution

```
(gdb) c
Continuing.
BUH
[Inferior 1 (process 7228) exited normally]
```

- And we can continue execution
  - Here, we exit since we don’t hit our breakpoint again
  - And, we print out our modified string

Summary

You should now be able to

1. Follow and understand simple assembly programs
2. Create your own programs
3. Debug them using gdb

We are only scratching the surface, but it's enough to get started