Introduction to x86 Assembly in Linux using Syscalls

Travis Phillips

What is Assembly Language?

- One step up from machine byte code
- Uses mnemonic instructions to explain itself.
- Very simple and straightforward rules.
  - powerful and dangerous all at the same time...
- Firm rules but think of it like a game, once you learn the rules it's pretty straightforward.

Why Should I Care About It?

- Useful in security for shellcode and reversing.
- Very fast and lightweight code.
- Fun and interesting way to learn about processors and memory.

How Does Assembly Language Work?

- Write your code.
  - Use the text editor of choice
- Compile it.
  - I use NASM to compile
  - nasm -f elf [file.asm]
- Link it.
  - ld -m elf_i386 [file.o] -o [file]
- Run it.
  - If all goes well

Just Script the Build

```
$ ./buildnasm.sh helloworld.asm

 --==[ compile and link NASM script v1.0]==--

 [*] Compiling helloworld.asm with NASM...Done!
 [*] Linking the object file with ld...Done!
 [*] Removing Object file helloworld.o...Done!
 [*] Done Son!

$
```

Memory... It's Kinda Important

- Memory is a place to store data as you are processing it
- On x86 you have a very limited number of registers so you will be using it to juggle data around.
- Memory is available as RAM install in the computer.
- The CPU also has a bit of memory available as registers.

Memory Layout in RAM

- Memory you'll be using will generally be in two sections.
- The Heap
  - Dynamic memory designed to be allocated and freed by the program
- The Stack
  - Lower memory section that just “grows upwards“ meaning the address decreases as you add to it.
  - Works on a First In, Last Out Model (FILO)

Registers

- Located on CPU
- Very fast but small storage.
- Used to setup syscalls.
- Generally used for small math or pointers to data stored in memory.

Registers

- General Purpose Registers
  - EAX, EBX, ECX, EDX
- Stack and Base Pointers
  - ESP
  - EBP
- Index Registers
  - ESI, EDI
- instruction pointer
  - EIP

Registers

- General purpose Registers can be addressed in a few different ways.
  - By 32 bit E*X notation
  - By 16 bit *X notation
  - By 8 bit high and low of 16 bit using *[H|L]

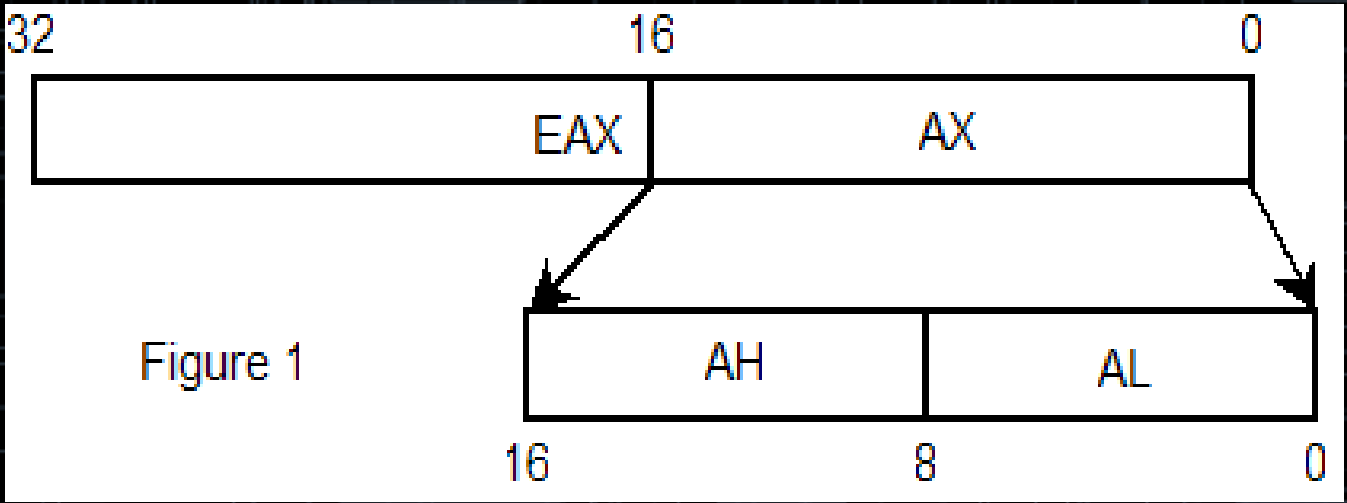

Figure 1

Instructions - MOV

- Purpose
  - Used to place data in one location to another.
- Syntax
  - MOV [DST], [SRC]
- Example
  - MOV EAX, 4
- This is Intel Syntax, AT&T Syntax would swap the Source and Destination.
- NASM uses Intel Syntax.

Instructions - XOR

- Purpose
  - Used to run an Exclusive Or (XOR) operation.
  - Useful for clearing registers.
  - The first parameter is the one it is run on.
- Syntax
  - XOR [Val1], [val2]
- Example
  - XOR EAX, EAX

Instructions - INT 0x80

- Purpose
  - Interrupt 80 – A friendly way for us to poke the kernel and tell it to execute a syscall we set up.
- Syntax
  - INT 0x80

Instructions – ADD and SUB

- Purpose
  - Provides us with a way to do simple addition and subtraction. The Val1 is the one that will collect the sum
- Syntax
  - ADD [Val1], [Val2]
  - SUB [Val1], [Val2]
- Example
  - ADD EAX, 5
  - SUB EAX, EBX

Instructions – INC and DEC

- Purpose
  - Provides us with a way to increment and Decrement a register. These are awesome because they are only 1 byte!
- Syntax
  - INC [Val1]
  - DEC [Val1]
- Example
  - INC EAX
  - DEC EAX

Labels

- Labels provide us with something of a friendly name for a section of the program
- These are used to create sections, variables, and jump/call points.
- To create a label just place text followed by a colon (:)
- Example:
  - The label on the line above is an example of how to create a label ;-)

Sections

- ELF Binaries have sections that we can create via labels.
- .data - For Storing Constants
- .bss - For variables that can be dynamic.
- .text - For some reason, this is where executable code goes...

Instructions - DB

- DB stands for Declare Byte
- Can be used to mark a string of bytes
- Used with labels, this can make a variable
- Used to allow the compiler to just place bytes at that location of the program

Jumps

- Jumps are used to make the program skip to another address. These take one value which is the address to go to. There are a few types
- JMP
- JGE, JG, JLE, JL
- JZ, JNZ, JS, JNS

Instructions - PUSH & POP

- Stack instructions
- PUSH sends data to the top of the stack
- POP pulls data from the top of the stack
- Think of it like a stack of cards or can of Pringles.

CALL and RET

- Like a jump, except it will push the next instruction address before it jumps to the stack.
- RET will pull that address of the stack and set it as the EIP.
- This can work for making functions in your code.

What are Syscalls

- Syscalls are functions available to us by the kernel.
- If you have your headers installed on x64_86 Debian, they can be found in they can be found in /usr/include/x86_64-linux-gnu/asm/unistd_32.h
- Also online at http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Table on The Web

- Also online at http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-

How to setup a syscall

- Syscall value goes into EAX
- parameters go into EBX, ECX, EDX, ESX, EDI in that order as needed
- Once every thing is setup the way you want it you use INT 0x80 to poke the kernel and tell it to run it.

How to setup a syscall

- So if we wanted to print Hello World...
- We would use the write syscall which is number 4, and load it into EAX
- Write takes 3 parameters
  - File descriptor to write to (1 is STDOUT)
  - Pointer to string to write
  - Number of bytes to print (Basically the length of the string)

So Our Registers Should Look Like

```
(gdb) info registers
eax            0x4        4
ecx            0x8048082  134512770
edx            0xe        14
ebx            0x1        1
```

```
(gdb) x/s 0x8048082
0x8048082:      "Hello, World!\n"
```

Example 0x01 – Hello World

```
global _start        ; global is used to export the _start label

section .text
_start:
    mov eax, 4     ; Syscall number for Write()

    mov ebx, 1     ; File Descriptor to write to
                   ; In this case: STDOUT is 1

    mov ecx, msg   ; String to write. A pointer to
                   ; the variable 'msg'

    mov edx, 14    ; The length of string to print
                   ; which is 14 characters

    int 0x80       ; Poke the kernel and tell it to run the
                   ; write() call we set up

    mov eax, 1     ; Syscall for Exit()
    mov ebx, 0     ; The Exit Code we want to provide.
    int 0x80       ; Poke kernel. This will end the program.

section .bbs
    msg: db "Hello, World!",0xa
```

Example 0x01 – Hello World

```
$ ./buildnasm.sh helloworld.asm

 --==[ compile and link NASM script v1.0]==--

 [*] Compiling helloworld.asm with NASM...Done!
 [*] Linking the object file with ld...Done!
 [*] Removing Object file helloworld.o...Done!
 [*] Done Son!

$ ./helloworld
Hello, World!
$
```

Value of Doing This?

- 4931 / 640 = 7.7
- Our program is *almost* 8 times smaller than its C coded counterpart.
- When size matters this is valuable!

```
$ cat helloinc.c
#include <stdio.h>

int main(){
        printf("Hello, World!\n");
}
$ ls -l helloworld helloinc
-rwxr-xr-x 1 owner owner 4931 Jul 25 20:44 helloinc
-rwxr-xr-x 1 owner owner  640 Nov 11 21:31 helloworld
$
```

Value of Doing This?

```
$ strace ./helloworld
execve("./helloworld", ["./helloworld"], [/* 37 vars */]) = 0
[ Process PID=6252 runs in 32 bit mode. ]
write(1, "Hello, World!\n", 14Hello, World!
)           = 14
_exit(0)                                  = ?
+++ exited with 0 +++
$
```

Value of Doing This?

```
$ strace ./helloinc
execve("./helloinc", ["./helloinc"], [/* 37 vars */]) = 0
[ Process PID=6301 runs in 32 bit mode. ]
brk(0)                                  = 0x89dc000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7710000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=198151, ...}) = 0
mmap2(NULL, 198151, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffffff76df000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\233\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1742588, ...}) = 0
mmap2(NULL, 1747580, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xffffffff7534000
mmap2(0xf76d9000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a5000) = 0xffffffff76d9000
mmap2(0xf76dc000, 10876, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffffff76dc000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7533000
set_thread_area({entry_number:-1, base_addr:0xf7533940, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_
 seg_not_present:0, useable:1}) = 0 (entry_number:12)
mprotect(0xf76d9000, 8192, PROT_READ)   = 0
mprotect(0xf7734000, 4096, PROT_READ)   = 0
munmap(0xf76df000, 198151)              = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff770f000
write(1, "Hello, World!\n", 14Hello, World!
)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
$
```

Also Really Easy to Debug!

Source Code

```
_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg
    mov edx, 14
    int 0x80
    mov eax, 1
    mov ebx, 0
    int 0x80
```

Objdump Disassembly

```
mov     $0x4,%eax
mov     $0x1,%ebx
mov     $0x8048082,%ecx
mov     $0xe,%edx
int     $0x80
mov     $0x1,%eax
mov     $0x0,%ebx
int     $0x80
```

Example 0x02 – Fork Bomb

- Fork Bombs will hang a system that doesn't limit the number of children process a process can create by endless forking itself. With syscalls and jmp, we can make a fork bomb.
- Fork requires no parameters but will return a value in EAX; The parent will see the PID of the child EAX, and the child will see zero EAX.
- But we don't even need that

Example 0x02 – Fork Bomb

```
global _start

section .text
_start:
        xor eax, eax
        mov eax, 2
        int 0x80
        jmp _start
```

Fork Bomb in 11 Bytes!

```
31 c0                   xor    %eax,%eax
b8 02 00 00 00          mov    $0x2,%eax
cd 80                   int    $0x80
eb f5                   jmp    8048060 <_start>
```

Shellcode Limitations – Null Bytes

- Shellcode is often passed as a string.
- Therefore NULL bytes generally aren't allowed due to the fact these terminate a string.
- This also can save us some space if we think about it!
- Both these examples had null bytes. Let's revisit them and remove them...

Removing 0x00 From Fork Bomb

- We had null bytes due to the MOV EAX, 2 instruction.
- Instead let's use MOV AL, 2 which only addresses the lower address and is 8 bits instead 32. This removes them and makes the Payload only 8 bytes!

```
31 c0                   xor    %eax,%eax
b0 02                   mov    $0x2,%al
cd 80                   int    $0x80
eb f8                   jmp    8048060 <_start>
```

Same Trick on Hello World

```
mov     $0x4,%al
mov     $0x1,%bl
mov     $0x8048072,%ecx
mov     $0xe,%dl
int     $0x80
mov     $0x1,%al
dec     %ebx
int     $0x80
```

Questions?

Slides, Code, and Build Script are available on

wiki.jaxhax.org