

[Home](#)[Blog](#)[Talks](#)[Tutorials](#)[Podcast](#)[Reviews](#)[About Us](#)

# 0x0 Shellcoding Tutorial: Introduction to ASM

[Home](#) / [0x0 Shellcoding Tutorial: Introduction to ASM](#)

## 0x0 Shellcoding Tutorial: Introduction to ASM

```
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x08048080 <+0>:      mov     eax,0x4
    0x08048085 <+5>:      mov     ebx,0x1
    0x0804808a <+10>:     mov     ecx,0x80490a4
    0x0804808f <+15>:     mov     edx,0xc
    0x08048094 <+20>:     int     0x80
    0x08048096 <+22>:     mov     eax,0x1
    0x0804809b <+27>:     mov     ebx,0x5
    0x080480a0 <+32>:     int     0x80
End of assembler dump.
```

This blog post will be the start of a new series, covering the basic concepts of shellcoding from an introduction to assembly, all the way to writing your own shellcode for use in an exploit and beyond. I will be focusing on 32bit Linux assembly to start, and eventually branching into Windows assembly.

If you have never worked with assembly or a debugger before, that is okay, we will be introducing the concepts slowly, starting with basic manipulation of the registers, as well as creating a basic 'hello world' style program from scratch, compiling it into an executable file, and then debugging the program as it executes. As a comparison, a scripting language such as python can execute this task with a

single line of code(`python -c "print 'Hello world!'"`), compared to approximately 15 lines of assembly! If you plan to follow along in this post, the following setup is recommended:

1. VM platform (Virtualbox, Vmware, etc.)
2. 32bit Ubuntu VM (or base OS, if you are not using a virtual infrastructure)
3. GDB (GNU debugger), NASM (Netwide Assembler), and ld (GNU linker)

### Assembly Code Introduction:

Assembly language is a low-level programming language designed to facilitate communication with the microprocessor. The instructions are specific to the processor family, for example Intel, ARM, etc. The architecture also plays an important role in understanding assembly, as there are numerous differences between 32bit and 64bit. Today we will be focusing on Intel 32bit (IA-32) on Linux.

As it is not required for this tutorial, I will not go into extreme detail about CPU architecture or the detailed relationship between the system components, and will leave that for further research if you are interested.

The CPU registers we will be looking at today are EAX, EBX, ECX, and EDX. These registers have a general function as originally designed, but for our purposes, we can store any data we like in them for the time being. The standard usage for these registers are as follows:

EAX – “accumulator” normally used for arithmetic operations

EBX – Base register, which acts as a data pointer

ECX – “Counter” normally used to hold a loop index

EDX – Data register, which acts as an I/O pointer

We will avoid diving into the other registers until a later post.

### Manipulating Our Registers:

To begin, we will be creating a basic ‘hello world’ script in assembly utilizing the registers mentioned above. To do this, we will create

a new file called 'helloworld.asm' (this can be called anything you like), and edit the file in a text editor to create two new sections '.text' and '.data' as seen below:

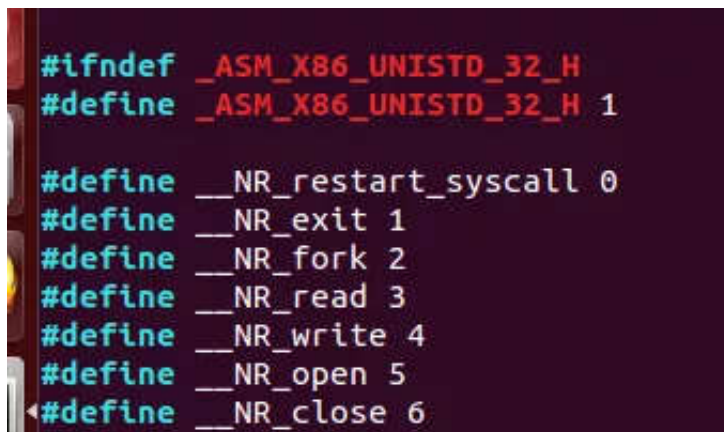
```
1 section .text
2 global _start      ;default entry point for linking
3
4 _start:             ; entry point for commands
5
6 section .data
```

The .data section is where we will be storing our string (this can be used for variables, etc), and the .text section is where we will create our entry point for ELF linking, our instructions for manipulating our registers to set up our syscall(s), as well as our instruction to the kernel to execute our system calls.

First, we will need to add our string to the .data section using define byte, or db:

msg: db "Hello World!"; the string, followed by a new line character

Our next step will be to determine what system calls we will be using for our assembly instructions. To take a look at the available syscalls, we will need to look at the 'unistd\_32.h' file, usually located '/usr/include/i386-linux-gnu/asm/' or some variation. We can open the file to view the available calls:

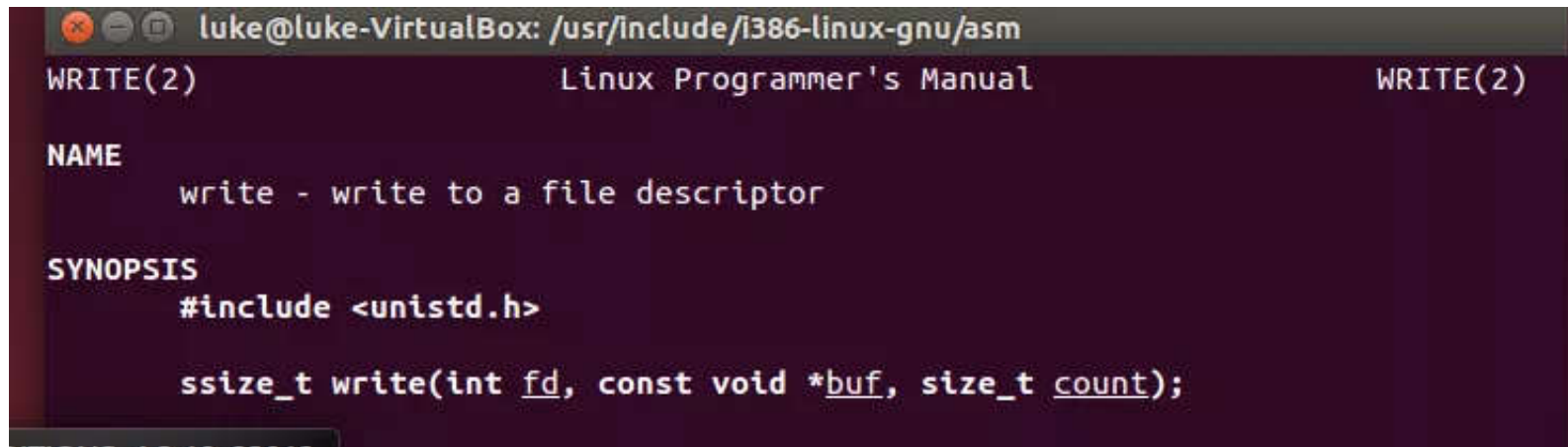


```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
```

Immediately we see two system calls that we can leverage, the exit function (#define \_\_NR\_exit 1) and the write function (#define

\_\_NR\_write 4). Take note of the two syscall numbers as we will use these later. We can see further detail about these syscalls by using the 'man 2' command (ex. man 2 write):



```
luke@luke-VirtualBox: /usr/include/i386-linux-gnu/asm
WRITE(2)                Linux Programmer's Manual                WRITE(2)

NAME
    write - write to a file descriptor

SYNOPSIS
    #include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);

RETURN VALUE
```

Looking at our man file, we see numerous fields that we need to utilize, 'int fd' or field descriptor, 'const void \*buf' or buffer, and 'size\_t count' or string size. In this example, our field descriptor indicates the location we will write to (0 for standard input, 1 for standard output, and 2 for standard error). Our buffer in this case, is going to be our 'Hello World!' string, and count is our buffer length. To recap, we have the following:

- syscall: 4 ; syscall number referencing our write command
- fd: 1 ; field descriptor indicating that our string will be written to stdout
- \*buf: msg ; hello world string we will create in the .data section
- count: 13 ; length of our buffer 12 for our buffer, plus a new line character

Now that we have identified the necessary information, we can begin to manipulate registers. To do this, we will be using the Intel structure of register manipulation with the 'mov' command:

mov

[destination],

We will repeat this with each of the four fields, sequentially with the EAX, EBX, ECX, and EDX registers, followed by an 'int 0x80' command to execute the system call:

```
1  section .text
2  global _start      ;default entry point for linking
3
4  _start:            ; entry point for commands
5
6      ; use the write syscall to print 'Hello world!' to stdout
7      mov eax, 4      ; move syscall 4(write) to the eax register
8      mov ebx, 1      ; move field descriptor for stdout to ebx
9      mov ecx, msg     ; move the memory address of our string to ecx
10     mov edx, 13      ; move the length of the string to edx
11     int 0x80         ; execute the syscall
12
13 section .data
14     msg: db "Hello world!", 0x0a ; the string, followed by a new line character
```

Now that we have our write syscall taken care of, we will need follow the same procedure to execute the program cleanly. To do this, we will be using the 'exit' syscall mentioned earlier, this time only requiring the 'int status' to be utilized. After following the steps for the exit call, your code should be similar to below:

```
1  section .text
2  global _start      ;default entry point for linking
3
4  _start:            ; entry point for commands
5
6      ; use the write syscall to print 'Hello world!' to stdout
7      mov eax, 4      ; move syscall 4(write) to the eax register
8      mov ebx, 1      ; move field descriptor for stdout to ebx
9      mov ecx, msg     ; move the memory address of our string to ecx
10     mov edx, 13      ; move the length of the string to edx
11     int 0x80         ; execute the syscall
12
```

```
13 ; use the exit syscall to exit the program with a status code of 0
14 mov eax, 1 ; mov syscall 1(exit) to the eax register)
15 mov ebx, 0 ; move status code to ebx
16 int 0x80 ; execute the syscall
17
18 section .data
19 msg: db "Hello world!", 0x0a ; the string, followed by a new line character
```

### Creating Our Executable:

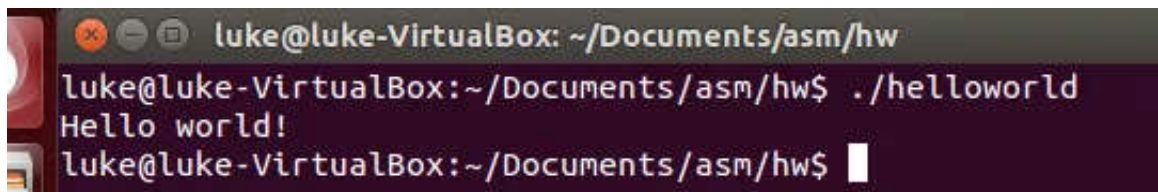
Now that our assembly file has been created, our next step will be to assemble it into an object file, and then use our linker to create our ELF executable. We will use the following NASM command to create our object file:

```
nasm -f elf32 -o <output object file> <input assembly file>
```

Now that we have a successful object file, we can link it with ld, and create our final executable. We will use the following command:

```
ld -o <output file> <input object file>
```

Assuming this occurs successfully, we should have a fully functional ELF executable. We can now execute our file and ensure proper execution:

A terminal window titled 'luke@luke-VirtualBox: ~/Documents/asm/hw'. The prompt is 'luke@luke-VirtualBox:~/Documents/asm/hw\$'. The user enters './helloworld' and the output is 'Hello world!'. The prompt returns to 'luke@luke-VirtualBox:~/Documents/asm/hw\$'.

Success! You have created your first functional program with assembly. In our next tutorial, we will be introducing you to debugging with GDB and further inspecting our hello world file to understand the effect each step of our program has as it happens.

Share this:



---

Related

## Python the Almighty

January 10, 2014

In "blog"

## 0x0 Python Tutorial: Getting Started

July 30, 2014

In "blog"

## 0x0 Python Tutorials: Getting Started pt2

August 22, 2014

In "blog"

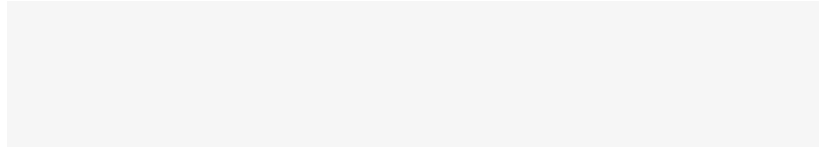
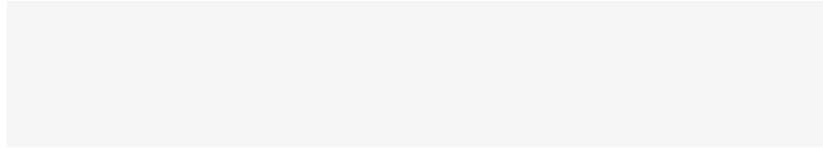
Share This Story, Choose Your Platform!



## Related Posts

---





Copyright 2012-2016 Primal Security | All Rights Reserved | Powered by Coffee

