

Forum:[Hack Forums](#)

Thread: [\[Tutorial\]Buffer Overflow Part 3 : Learning to Code Assembly](#)

7 **R3X** - August 14, 2017, 3:39 am

Stack Buffer Overflow

Learning to Code Assembly

First off, thanks for the awesome comments you left on the last part of this tutorial. I was almost on the verge of giving up on these tutorials when I woke up in the morning and saw those encouraging comments. It takes quite a lot of time to research all the material, read them all, test them, organize them in correct flow, add my own experience in it and then finally to write it all down for you to read it. It's those comments and your appreciation that keeps me ticking. That being said, let's get to today's lesson.

Before you start reading this part of the tutorial, I'd highly encourage you to go and take a look at [\[Tutorial\]Buffer Overflow Part 2 : Introduction to assembly](#)

Last time I wrote about the assembly language and its key components that one should know before writing their own assembly language codes. This part of the tutorial is going to be awesome as we're going to create some of our own assembly codes. For this purpose I'm going to use GNU assembler (aka GAS) and GNU linker. These softwares come pre-installed with most of the Linux distros so there is a fine chance that you won't need to install them again but just in case you don't have them installed, these packages go by the name of "as" and "ld" respectively. So go ahead and have them installed on your machines. Now without a further ado, let's get started.

Hold on! Not so fast mate!! We still have a little bit of theory to learn. Let's cover them first.

Sections of Assembly

There are three most common sections of assembly that we'll look at, namely:

.data Section

.bss Section

.text Section

.data Section

Data section (aka data segment) is a portion of a program that contains initialized static variables, that is, global variables and static local variables. The size of this segment is determined by the size of the values in the program's source code, and does not change at run time.

.bss Section

BSS stands for "Block Started by Symbol". BSS section (aka bss segment) contains all the statically allocated spaces. For example a 1000 byte buffer.

.text Section

.text tells the assembler that the upcoming section is considered assembly language instructions or simply, we write our assembly code/instructions in this section.

Datatypes

`.ascii str` : This stores `str` in memory, but without a null terminator.

`.asciz str` : This stores `str` in memory, but with a null terminator. The "z" refers to zero, which is the ASCII code for the null character. This is how C-style strings are stored.

`.byte b1, ..., bn` : Stores `n` bytes contiguously in memory. It's comparable to the "unsigned char" in C programming language.

`.short h1, h2,..., hn` : Stores `n` 16-bit integer contiguously in memory. It's comparable to the "short" in C programming language.

`.int w1, w2,..., wn` : Stores `n` 32-bit integer contiguously in memory. It's comparable to the "int" in C programming language. You can also use `".word"` instead of `".int"`.

`.float f1, f2,..., fn` : Stores `n` Single precision floating point numbers. This one is similar to "float" in C programming language.

`.double d1, d2,..., dn` : Stores `n` Double precision floating point numbers. This one is similar to "double" in C programming language.

`.comm buffer, <size>` : This line is used in `.bss` section. It declares common memory area of `<size>` bytes.

`.lcomm buffer, <size>` : This line is used in `.bss` section. It declares local common memory area of `<size>` bytes.

Enough of this theory!!! I know, right? Let's create our first assembly language code. It's a tradition in the world of programming that the first program of any programmer is a traditional "Hello World" program. So let's go along our tradition.

Simple "Hello World" Code

Code `.data`

`HelloWorldString:`

`.ascii "Hello World\n"`

`.text`

`.globl _start`

`_start:`

`mov $4, %rax`

`mov $1, %rbx`

`mov $HelloWorldString, %rcx`

`mov $12, %rdx`

`int $0x80`

`mov $1, %rax`

`mov $0, %rbx`

`int $0x80`

You can buy me a cup of coffee so I could stay up at night and write for you.

My Blockchain wallet : 17mKvXY27BdS2FnpAS91FbvX2XuGos4BNW

Let me know if you've donated.

Have a great day ahead. Keep Hacking ? Learning to Code Assembly Continued...

Now that we've written the code, let's see how to run it and also try to understand how it works. The program should be saved as a file with a ".s" extension. Suppose that we've saved the file with a name "hello_world.s". This file is simple an ASCII file and our processor cannot do anything with it. We need to assemble it to generate our object file. To do this, type in:

```
Code as -o hello_world.o hello_world.s
```

This will assemble the source file "hello_world.s" and create its corresponding object file "hello_world.o".

Now we need to link the file using GNU linker. To do this, type in:

```
Code ld -o hello_world hello_world.o
```

This will create an executable file named "hello_world". Now run this executable. The following is the screenshot of my execution.

Now let's see how does this program actually work.

Line 1 in our code declares a .data segment in our code. Line 2 declares a label named HelloWorldString and the next line initializes an ASCII string with a value "Hello World\n". Basically, the idea is to declare an ASCII string with some pre-defined value. You can compare this to the following C programming language instruction:

```
Code char HelloWorldString[] = "Hello World\n";
```

This is simply an analogy and there are quite some differences between the assembly instructions and its C counterpart that I provided. For example the C string "Hello World\n" is a null terminated string while the assembly string is not (Note the datatype used is .ascii and not .asciz). But we're not gonna care about that many details for now.

Moving on, we have our text section in the program and the first line says ".globl _start". ".globl" is an assembly language directive (or simply a reserved keyword) which tells that the label "_start" is accessible from outside the program. Why did we do that, you ask? That's a nice question. Think of a C++ program and ask yourself why the main() function is always public? It's because the main() function is the point of beginning for our program. When we pass a command to execute our code, it doesn't get executed all by itself. Someone has to call its point of beginning and that "someone" is our operating system. Similarly, in an assembly language program, the label "_start" is the point of beginning.

Next, we move an integer value 4 to register rax, value 1 to rbx, address of string HelloWorldString to rcx, value 12 to rdx and simply called an interrupt 0x80. Simple, right?

NO!!!

There's a lot to understand in it. Let's get it piece by piece, starting off by what does interrupt do. I also promised you in my last part of this tutorial that I'd dedicate a separate section for interrupt 0x80, so here we go.

Int 0x80

The following diagram might look familiar to you if you've gone through the last part of this tutorial. It says that the interrupt 128 or hexadecimal value 80 (0x80, "0x" is hexadecimal just like in C programming language) is an exception for system calls. Well what is a system call? Basically, these are simple functions that are pre-defined in the kernel of an operating system. An application program makes a system call to get the operating system to perform a service for it, like reading from a file, writing to console, etc.. fork(), write() and read() are a few examples of Linux system calls.

Every system call in Linux has an assigned value (an integer) and to make a system call in assembly, we need to put this value into the AX register before putting the interrupt 0x80. As you can clearly see that we've moved "4" in RAX, a question you might ask is which system call does the value "4" corresponds to? And the answer is written in "/usr/include/x86_64-linux-gnu/asm/unistd_32.h" file. It's the "write" system call. Now let's take a look at what does this "write" system call expects in order to work properly. A quick DuckDuckGo search(Yes I hate google too) for "write system call in Linux" leads me to the [Man page of write syscall\(\)](#). We can see that it requires 3 parameters namely, a file descriptor of the stream to which it needs to write, Address of the buffer that holds what should be written and finally the number of bytes that should be written.

But how do we pass these parameters in an assembly language program? Well, we need to store them in the BX, CX and DX registers respectively. Talking of its first parameter, a file descriptor of the stream stream, we know that linux defines the three streams namely Standard Input Stream(STDIN), Standard Output Stream (STDOUT) and Standard Error Stream (STDERR) with values 0, 1 and 2 respectively. You might also want to recall that you redirect any error message in bash script to bit-bucket known by /dev/null if you don't want them to appear on the screen.

Code 2>/dev/null

This works because "2" represents the STDERR. Anyway, we need to output our string on STDOUT, i.e. our console so we moved "1" into RBX. Then, the address of HelloWorldString is moved to RCX and finally the size of string "Hello World\n" i.e. "12" is moved to RDX(Hello = 5, Space between Hello and World = 1, World = 5 and Newline character = 1, 5 + 1 + 5 + 1 = 12). Now that we have all the arguments and the syscall value in place and ready to call the interrupt, we fire int 0x80.

I'll leave it to you to figure out what does the next interrupt does. Just follow the same method that we used to figure out our write syscall.

NOTE : You might be wondering that there is a file named "unistd_64.h" in the same directory as of the file "unistd_32.h" and since we're working in a 64-bit environment why is it that the convention in unistd_32.h has been followed instead of unistd_64h. The reason is that "int 0x80" is used to perform system call in a 32-bit environment while it has its counterpart named "syscall" for performing system calls in a 64-bit environment. When using "syscall" instead of "int 0x80", the convention in the file "unistd_64.h" would be followed.

Here's a link for you to practice, test and sharpen your assembly language skills:

[Practice some GAS coding](#)

This is it for this time. If you have any questions so far, please leave them below in the comments

and I'll try to answer them all. I encourage you to put some comments, either good or bad. Let me know if you liked the way I write, it'll encourage me to write further. If you didn't like the way I write, let me know about it too. One learns the best from his own mistakes. Your valuable comments are heartily welcome.

You can buy me a cup of coffee so I could stay up at night and write for you.

My Blockchain wallet : 17mKvXY27BdS2FnpAS91FbvX2XuGos4BNW

Let me know if you've donated.

Have a great day ahead. Keep Hacking ?

andmuchmore - August 14, 2017, 4:59 pm

You have done it again dude.

I am glad you are going through all the theory of buffers and giving people a fundamental understanding of a programs structure and how memory is allocated and assigned before getting into overflows!

Keep up the good work!

7_R3X - August 14, 2017, 11:13 pm

(Aug 14, 2017 04:59 PM)andmuchmore Wrote: [\[CITE\]](#) You have done it again dude.

I am glad you are going through all the theory of buffers and giving people a fundamental understanding of a programs structure and how memory is allocated and assigned before getting into overflows!

Keep up the good work!

Thanks for the appreciation!! ?

KooKlique - August 24, 2017, 6:30 am

I don't know anything about assembly, I've been programming in vb, and python for quite a few years now.. so when I see something related to programming in any language, I usually take the time to breeze thru it and see if it's something I might be interested in. The language is definitely completely different than vb, or python.. but that also intrigues me. I may take the time later and look at your other tutorial/s and see if this is something I might want to learn. Well, that's for taking the time to write these, excellent material.