

Forum:[Hack Forums](#)

Thread: [\[Tutorial\]Buffer Overflow Part 2 : Introduction to assembly](#)

7 **R3X** - August 9, 2017, 11:53 am

Stack Buffer Overflow

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 1: Introduction to Stack](#)

Last time I wrote about stack and it's basic operations, namely, Push and Pop. Continuing our tutorial of Buffer Overflow tutorial, this time we're going to get introduced to Assembly language. Please note that there's a lot of theory we need to cover before we could actually dive into the practical aspects. This part of the tutorial is going to be entirely theoretical. Hopefully we'll create some awesome programs in the next part.

Assembly Language

As we all know that all the computers run on some or the other kind of processor. Each kind of processor has its own set of instruction that it user to handle different operations, be it as simple as calculating an addition or be it as complex as running a very complex artificial intelligence system. These set of instructions are what we call the Machine Language Instructions. A machine language instruction usually has the following syntax:

[label] mnemonic [operands] [;comment]

The parts of an instruction inside brackets in the above syntax are optional. A few examples of machine instructions are:

```
INC %rax
JMP tomato
MOVI $10, %eax
int 0x80
```

Registers

A register file is an array of processor registers in a CPU or a central processing unit. Talking of a 64 bit environment, we have 16 registers each of which can hold up to 64 bits or 8 bytes of data. The registers are mainly divided into three categories:

1. General Purpose Registers
2. Control Registers
3. Segment Registers

General purpose registers are further divided into 3 categories:

1. Data Registers
2. Pointer Registers
3. Index Registers
4. Additional Registers (Only for 64-bit Architecture)

General Purpose Registers

Lets talk about general purpose registers. General purpose registers (GPRs), as the name

suggests, are not used for storing any specific type of information. Instead operands as well as addresses are stored at the time of program execution. As an assembly language programmer, you might use these registers to store any kind of data but in past each of these registers had its own specific use. Not just in the past, but those usage are still followed by almost all the modern compilers. We'll look at these uses as we progress along this tutorial. For now, let's take a look at each type of GPRs:

Data Registers

There are four data registers namely, RAX, RBX, RCX and RDX. Each of these registers can hold 64 bits or 8 bytes of data. You can access the full 8 bytes (Bit 0-63) by mentioning their name as RAX, RBX, RCX and RDX. In case you have a need of only 4 bytes, you can access the least significant 4 bytes (bit 0-31) of these registers by using the name EAX, EBX, ECX and EDX. You can also access the Least significant byte (bit 0-7) by the names AL, BL, CL and DL. Also you can access the byte next to least significant byte using AX, BX, CX and DX. Confused? Don't worry, the following diagram would solve all your problems.

The AX register, also known as accumulator register, is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

The BX register, also known as the base register, is used in indexed addressing.

The CX register, also known as the count register, is used to store the loop count in iterative operations.

The DX register, also known as the data register, is also used in input/output operations as AX register. You can also use it with AX register along with DX for multiply and divide operations involving large values.

Pointer Registers

The three pointer registers, RIP, RBP and RSP are very important to understand as the success of most buffer overflow attack depends on them. Let's take a look at their uses:

Instruction Pointer IP – The IP register stores the offset address of the next instruction to be executed. Instruction pointer in association with the Code Segment (CS) register as CS:IP gives the complete address of the current instruction in the code segment.

Base Pointer BP – The BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in Stack Segment (SS) register is combined with the offset in Base Pointer (BP) to get the location of the parameter.

Stack Pointer SP – The SP register provides the offset value within the program stack. Stack Pointer (SP) in association with the Stack Segment (SS) register SS:SP refers to be current position of data or address within the program stack.

You don't need to worry, if you don't understand them now. We'll practically look at them and how their interaction when we'll dive deep into BOF. For now, just try to remember their uses and I encourage you to go to Wikipedia and read more about them as it'd help you understand what exactly goes on behind the curtain when a program executes on your computer.

Index Registers

The Index registers, RSI and RDI, are used for indexed addressing and sometimes used in addition and subtraction. Their uses are mentioned below:

Source Index (SI) – It is used as source index for array/string operations.

Destination Index (DI) – It is used as destination index for array/string operations.

Additional Registers

In addition to increasing the size of the general-purpose registers, the number of general-purpose registers is increased from eight in x86 to 16 in x64. These additional registers are r8, r9, r10, r11, r12, r13, r14 and r15. These additional registers also have sub-registers and can be accessed with their correct names.

Quote: A picture is worth a thousand words.

So instead of naming each of these sub registers, here's a pic of sub registers that explains it all.

Now that we've seen all the general purpose registers along with their uses, it's time to put them all together in one pic. The list of all the general purpose registers and their sub registers is shown in the pic below. Here we go:

Control Registers

The instruction pointer register (RIP) and the flags register combined are called the control registers. We've already looked at Instruction pointer (RIP) and I've dedicated a separate section for flag register which I'll write after explaining the registers, so nothing much to see here. Let's keep rolling.

Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments in an assembly program:

Code Segment (CS) – It contains all the instructions to be executed. This segment is sometimes also known as the text segment. A Code Segment register or CS register stores the starting address of the code segment.

Data Segment (DS) – It contains data, constants and work areas. A Data Segment register or DS register stores the starting address of the data segment.

Stack Segment (SS) – It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure that we've already discussed in the previous section of this BOF tutorial. The Stack Segment register or SS register stores the starting address of the stack.

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 Black Hat ? Assembly Language Continued...

Flags

On the 64 bit architecture, there are a total of 64 bits on the Flag register. Most of them are reserved. We're not going to discuss each and every bit as its not of much significance for BOF. We're going to look at the most commonly used flags in assembly language programming. If you want totake a look at each bit of this register, I'd suggest you to visit the wikipedia page here(https://en.wikipedia.org/wiki/FLAGS_register). Lets start with the most common bits:

Overflow Flag (OF) – It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.

Direction Flag (DF) – It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

Interrupt Flag (IF) – It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.

Trap Flag (TF) – It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

Sign Flag (SF) – It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

Zero Flag (ZF) – It indicates the result of an arithmetic or comparison operation. A non-zero result clears the zero flag to 0, and a zero result sets it to 1.

Auxiliary Carry Flag (AF) – It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

Parity Flag (PF) – It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.

Carry Flag (CF) – It contains the carry of 0 or 1 from a high-order bit leftmost after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

Interrupts

An interrupt causes the CPU to pause its current execution, store the state of the registers on the stack, then process a defined subroutine. When this subroutine completes, the interrupt finishes, the registers are restored from the stack, and the previous execution state resumes. Here's a list of available values in the interrupt handler tables.

We won't deal with all of these interrupts. In fact we'll only deal with int 0x80. I'll explain this interrupt in detail in a later part of this tutorial.

This is it for this time. I didn't expect it to get this long but we need to set up a firm foundation for BOF. I'll continue to write this tutorial on Buffer Overflow as soon as I get 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 Black Hat ?

andmuchmore - August 10, 2017, 3:31 pm

Very comprehensive tutorial! Keep up the good work mate! Can't wait to read the next one!

7_R3X - August 11, 2017, 5:41 am

(Aug 10, 2017 03:31 PM)andmuchmore Wrote: [\[CITE\]](#) Very comprehensive tutorial! Keep up the good work mate! Can't wait to read the next one!

Thanks for the encouraging comments! Working on the next part. ?

Freshnuts - August 11, 2017, 7:21 pm

Yeah, man this is great. I appreciate the effort put in, and it's easy to read and refer to. I got it saved.

Prawnster - August 11, 2017, 7:24 pm

Nice share mate, keep up the good work.

7_R3X - August 12, 2017, 3:34 am

(Aug 11, 2017 07:21 PM)Freshnuts Wrote: [\[CITE\]](#) Yeah, man this is great. I appreciate the effort put in, and it's easy to read and refer to. I got it saved.

(Aug 11, 2017 07:24 PM)Mazl Wrote: [\[CITE\]](#) Nice share mate, keep up the good work.

Thanks for the appreciating comments. ?

Unfortunately, I'm stuck with some work here and it might take a little longer than expected to post the next tutorial. Lets see if I could do it by coming Tuesday.

cryptoshell - August 13, 2017, 12:32 am

thanks for the share

MindOfLogic - August 13, 2017, 2:34 pm

You should make a video series on this. It's much harder to follow along with just text.

7_R3X - August 13, 2017, 10:59 pm

(Aug 13, 2017 02:34 PM)MindOfLogic Wrote: [\[CITE\]](#) You should make a video series on this. It's much harder to follow along with just text.

I'm not sure about other people but I find it really easy to follow along text instead of videos. ?

But thanks for the suggestion though. I'll definitely think about it. ?

Elweth - August 23, 2017, 2:37 am

HF these days lack guides like yours. Keep up the good work!

Your wikipedia link to flags are invalid btw.

https://en.wikipedia.org/wiki/FLAGS_register

