

Forum:[Hack Forums](#)

Thread: [Assembly Basics + Shellcode Basics](#)

Zezil - March 7, 2015, 11:06 am

Assembly

Preface: I made this for a friend initially and decided to share this with you guys. Hopefully it will help some people looking to get started in writing their own payloads/understanding the core of programming.

Lets begin with types of assembly. There is asm and at&t. They both function the same yet use different syntax.

Lets take a look at the mov command. What this does is it copies a value to a certain location. `nasm: mov eax, 1 ;( is a comment in assembly)`. But anyway, this copies the value 1 into eax. eax is represented by 0x000000, so when 1 is copied, the register then becomes 0x0000001. `at&t: movl 1, %eax` ;As you can see, the eax register is marked by a % sign to indicate it is a register, and the position is switched.

I don't want to confuse you, so lets take a detour into what registers are. We will be doing 32 bit assembly, though there is also 64 bit assembly which follows this same structure, though there are just larger registers. As I learned, that `rsp/rbp` in your C program was just the 64 bit form of those registers.

Anyway registers are sections of the processor that store data and perform functions. All registers are unique, and as such you can never simply store a value in one register blindly.

Heres a list of registers.

EAX (32 bit register) -- AX (16 bit register) -- AH(8 bit register for top 8 bits of AX)/AL(8 bit register for bottom 8 bits of AX)
EBX (32 bit register) -- BX (16 bit register) -- BH(8 bit register for top 8 bits of BX)/BL(8 bit register for bottom 8 bits of BX)
ECX (32 bit register) -- CX (16 bit register) -- CH(8 bit register for top 8 bits of CX)/CL(8 bit register for bottom 8 bits of CX)
EDX (32 bit register) -- DX (16 bit register) -- DH(8 bit register for top 8 bits of DX)/DL(8 bit register for bottom 8 bits of DX)
ESI (32 bit register) This has a function but I'm not sure what, I believe it's something for flags, which we can look at later)
EDI (32 bit register) This also has a function but I'm not sure what, I believe it's something for flags, which we can look at later)
ESP (32 bit register used to point to values on the stack)
EBP (32 bit register used to point to the bottom of the stack, allocating stack space for a program)

We will focus on EAX,EBX,ECX,and EDX for now.

Understand that if you store lets say the value 1 in AL, and you store 2 in EAX, you may very well be overwriting the AL register.

Remember, AX(AH/AL) are all just sub-registers of the big 32 bit EAX register. AX writes the bottom 2 bytes of EAX (16 bits), and AL writes to the bottom byte (or low order byte) of AX.

This is very very useful when working with shellcode.

Say you are performing a system call and you use 1 as a parameter. If you do `mov eax, 1` you have as stated earlier `0x00000001`.

This would create a large amount of null bytes in your shellcode. `/x00/x00/x00/x01/`. Null bytes terminate strings, and as your shellcode is a string, it will be terminated prematurely. To avoid this, who do `mov al, 1`. Now `al` stores `0x01`, and as `al` is a part of `eax`, `eax` also stores `0x01`! This leaves our shellcode as `/x01/` instead of the previously mentioned.

Now that you understand registers, lets move on to the next thing. That is the stack. First of all, you need to understand what the stack is.

The stack is a section of memory allocated to keep track of function calls/etc. In this function variables are stored and other things.

The stack grows from high memory addresses, to low addresses. (`0xffffffff` to `0x00000000`). You can push/pop values from the stack. Lets take a look at how this works. So a program begins to run, the `EBP` is stored as the bottom of the stack, allocating stack space for the program running.

Some values are stored such as the program name/parameters/etc. All sorts of shit. But we can ignore that. Now, when you enter a function, a return is placed on the stack before the function is called. This occurs for the reason of returning control to the initial program when a function is over, otherwise the code would continue running as though the program were still in a certain function. A function being of course, `echo('test')` for example.

So, when the function is finished and it echoes our 'test', our return is called, which holds a memory location. Then, our program jumps to this return location and the program continues to run!

Now that you have a basic understanding of the stack, we need to look at data storage on the stack. Intel processors use the little-endian system, which stores low order bytes first. As you remember from registers, say we have `0xff00`. `0xff` would be high order, and `0x00` would be the low order. For some reason, due to the little-endian structure, we are required to reverse our hex values when storing them to the stack. So, to store `0xff00`, we reverse the bytes and make it `0x00ff`.

Pretty wierd, but just follow that rule and things will be fine.

Now push and pop, something that is very useful and important. Lets start with the string `0x68657921` ("Hey!" in ascii). Remember how a string is terminated with a null? Thats important here. Now lets get into storing this string on the stack.

```
xor eax, eax
push eax
push 0x21796568
mov eax, esp
```

This stores the string "Hey!" on the stack and sets `eax` to that value. Lets break this down though. So, `xor eax, eax` is a logical function used to zero out registers most often. All you need to remember is that if you do `xor register, register` that register becomes equal to 0.

Then we push eax. Right now our stack looks like this.

|0x000000|(rest of stack)|

Right now also, the ESP which points to the value last pushed/popped. So at the moment, ESP holds the memory location for our beloved 0x000000.

Next we push Hey!, but the bytes are reversed due to our little-endian system.

So now our stack is this |0x68657921|0x000000|(rest of stack)| after we push Hey!

Now, ESP points to our Hey! ascii value in hex. When this is read, our string is read, and then terminated with our null, which was eax pushed to the stack after

being xored. Understand how ascii works, you have hex values for characters and then you terminate a string with a null byte (0x00) in this case though, having multiple null bytes on the register doesn't really

matter because the string is terminated anyway after that first null byte is read.

Then, we move the location esp points to to eax, which then basically stores our string from the stack!

This allows you to store large values/arrays of bytes to a certain register by having that register point to a memory location on the stack.

Bam, your program is complete! When you do the pop command, as a side note, esp moves to the provided value on the stack and points to the popped value, it doesn't actually leave the stack.

For example, pop eax simply moves the pointer to eax on the stack.

Whew, now that this is over, lets get into system calls.

The command int 0x80 is a kernel call that then reads register values to perform a system function/etc.

```
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
push eax
push byte 0x0a
push 0x6767696e
push 0x21737265
mov dl, 9
mov ecx, esp
mov bl, 1
mov al, 4
int 0x80
```

Whew, lets break this down. We clear all registers first, and then push a string to the stack as we previously did.

```
"push eax
push byte 0xa
push 0x6767696e
push 0x21737265"
```

Also note, I push the single byte 0xa to prevent null bytes from terminating the string. This is a hex

value for a new line char.

Then we store that value in ecx like previously done. Then, however, we store the value 1 in bl, which is just the parameter used to print

this string, required by the sys_write system call. Then we move the value 9 to our 8 bit low order edx register (this is the string length).

Then we move the value 4 to our 8 bit low order eax register. This is the value for the system call sys_write. Then int 0x80 is issued, which runs through our stored values to execute the system call.

Now eax holds the system call id (in this case 4 for sys_write)

ebx holds the parameter required to print the string (ebx holds the first param)

ecx holds the string we are printing (ecx holds the second param)

edx holds the string length (edx holds the third param)

This is the general structure of a system call. Also note that this is a linux system call.

Now that you have the basics of assembly, you can pursue more of the language and learn about other commands and ways to interact with memory.

Shellcode

This section will be very short, and will focus on creating shellcode.

As we mentioned earlier, should shellcode contain any null bytes, the shellcode will be prematurely terminated. As such, we need to make sure no null bytes are found in our shellcode.

Lets take this program as example

```
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
push eax
push 0x0a
push 0x6767696e
push 0x21737265
mov edx, 9
mov ecx, esp
mov ebx, 1
mov eax, 4
int 0x80
```

Shellcode is basically a program represented in hex. You can even make a C program, compile it, and create shellcode directly from C without working with assembly at all, however this would be very inefficient. If you wish to experiment yourself, you will see that often unnecessary lines of code are added, increasing the size of the shellcode.

Now, to get the hex of our assembly program, we will be using nasm to create a list file containing the hex values of each line of code.

In this case, our shellcode is...

\x31\xC0\x31\xDB\x31\xC9\x31\xD2\x50\x68\x0A\x00\x00\x00\x68\x6E\x69\x67\x67\x68\x65\x72\x73\x21\xBA\x09\x00\x00\x00\x89\xE1\xBB\x01\x00\x00\x00\xB8\x04\x00\x00\x00\xCD\x80\

Notice all of the null bytes? Lets eliminate them!

```
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
push eax
push byte 0x0a
push 0x6767696e
push 0x21737265
mov dl, 9
mov ecx, esp
mov bl, 1
mov al, 4
int 0x80
```

\x31\xC0\x31\xDB\x31\xC9\x31\xD2\x50\x6A\x0A\x68\x6E\x69\x67\x67\x68\x65\x72\x73\x21\xB2\x09\x89\xE1\xB3\x01\xB0\x04\xCD\x80\

Wow! Lets go over how we fixed that. Now, our problem was that we were storing single bytes in the 32 bit registers. This created, in the case of eax, a register containing the value 0x0000004. This then translated to the many null bytes we found in our code. We fixed this by simply using the 8 bit low order registers. In the case of ecx though, it did not matter as we were storing esp (a 32 bit memory location value) in ecx. Another problem was in our new line byte (0x0a). The problem with this was that we were not pushing a single byte, but rather ended up pushing 0x000000a! We simply fix this by specifying that we are pushing a single byte. And viola, we have our shellcode!

That is the basics of it, anything else you need in particular you should be able to figure out on your own now that you hopefully understand the basics!

-Written by Zezil.

Hope this helped you!

xVipX - March 14, 2015, 10:19 am

thanx you for shear this & sory for my bad english