The Vulnerable Space

Tuesday, 16 June 2015

(N)ASM101 - 0x01 - Calling Win32 API Functions

Hello and welcome to my first post. A few weeks ago I've acquired an interest in building my own small functions with ASM. I've been an amateur exploit developer for a number of years and used off-the-shelf shellcode quite a lot but I've never gotten my hands dirty with it.

I've decided to start blogging about what I learn thanks to a recent experience. I always believed that teaching was somewhat of a waste of time, and that the time spent teaching could be spent learning new things. This was proven to be wrong when, in a workshop I was delivering (BSides London 2015), a guy pointed out that there are better ways of achieving what I was explaining and that got me thinking. Teaching is not the one way process I had always thought it to be ...

Anyways .. enough about my life and let's get crackin' ..

Why NASM ?

When it comes to assembly language programming, NASM (Netwide ASM) and MASM (Microsoft Macro Assembler) are considered to be 2 of the top, if not the top, standard assemblers. There is little to no difference in the way ASM is written in both, as they both use Intel syntax, and as one would expected MASM is easier to use in a Windows environment.

For example, MASM comes with STDIN and STDOUT Windows console functions to read user input and to output strings to the console. Reading and Writing are functions that are used over and over again and hence Microsoft has decided to abstract the Windows functions that handle these operations to make them more accessible. Personally, I think it's very convenient but I don't like taking shortcuts. The purpose here is to learn ASM and since we're already dealing with the lowest level of programming language might aswell go all the way.

NASM is an assembler/disassembler for the Intel x86 architecture used for both Windows and Linux. It can output several binary formats such as Portable Executable and ELF.

Some other minor differences are:

- NASM is case-sensitive
- NASM requires square brackets for memory references
- Macros and Directives work completely different

If anyone is still not convinced by this explanation and would like to use MASM, they are welcome to do so. At the end of the day, the concepts remain the same.

Equipping ourselves

The 2 programs that will be required to generate executables from ASM are:

- NASM
- A Linker (eg: Golink)

NASM converts an ASM file into a Common Object File Format (COFF) file, whilst GoLink packages it with the required DLLs to produce the final PE (exe) file.

Laying down some theory

Before we dive in we will go over some basic theoretical concepts, namely the Registers and the Stack and how it's utilized when calling Windows functions. Those of you who are familiar with these concepts can jump to the next section.

The Registers

Registers are small locations used by the processor to perform operations. An x86 processor has 16 of these registers some of which are listed in the table below:

Register	Name	Main Purpose
EAX	Accumulator Register	Arithmetic Operations
EBX	Base Register	Base Pointer for Memory Access
ECX	Counter Register	Loop Counter and Shifts
EDX	Data Register	An extension to EAX
ESI	Source Index	Source Pointer for String Manipulation (Reading)
EDI	Destination Index	Destination Pointer for String Manipulation (Writing)
EBP	Base Pointer	Points to the base address of the Stack
ESP	Stack Pointer	Points to the top of the Stack
EIP	Instruction Pointer	Points to the next instruction to be executed
EFLAGS	FLAGS Register	Holds several Flags e.g.: - ZF (Zero Flag): 1 if result = 0, else 0 - CF (Carry Flag):1 if result = -ve, else 0 - OF (Overflow Flag): 1 if result overflowed, else 0

Twitter
@GradiusX

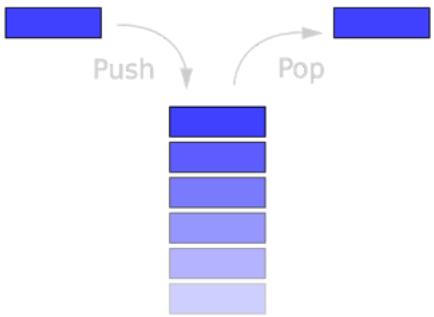
Blog Archive
▼ 2015 (5)
► September (1)
► July (2)
▼ June (2)
(N)ASM101 - 0x01 - Calling Win32 API Functions
Welcome

The E in front of each register signifies that the register contains 32 bits. In the case of 64-bit registers, this is replaced by an R, so EAX (32-bit) would become RAX (64-bit).

Some register can be further divided into 8- and 16-bit registers in the following manner:

The Stack

A stack is a last-in first-out data structure which supports 2 operations: PUSH and POP. A PUSH instruction will put something on top of the stack whereas a POP instruction will remove an item from the top of the stack.



These operations implicitly modify the ESP register. When a PUSH is executed, ESP is decremented and data is written at the memory address ESP is pointed at. In a POP operation, data is removed from the stack and stored in some other location; ESP is incremented.

The stack is used by x86 architectures as extra storage since the amount of data that can be stored in registers is severely limited. To better understand the role of the stack during execution we'll look at calling the MessageBox Win32 API function and take a close look at what happens when we do so. The following tells us that the MessageBox function is expecting 4 parameters:

```
int WINAPI MessageBox(
    _In_opt_ HWND    hWnd,
    _In_opt_ LPCTSTR lpText,
    _In_opt_ LPCTSTR lpCaption,
    _In_     UINT    uType
);
```

This is where the stack comes in. Win32 API functions use the STDCALL calling convention. This means that the CPU expects the parameters required by the function to be on the stack when the CALL instruction is encountered.

The stack has another very important role. Implicitly a CALL instruction performs 2 operations:

- The address right after the CALL instruction is pushed on the stack
- The execution flow (EIP register) is transferred to the function that is called. In the case of the MessageBox, this is somewhere inside User32.dll.

When the called function finishes executing, the saved address is popped back to EIP (RET instruction) and the execution of the main program continues.

To summarize, 2 main functions of the stack:

- Used to pass parameters to the called functions (callees)
- Used to save the state of the caller function before the execution is transferred to the callee.

Calling the MessageBox Function

With all the theory behind us it's time to present some code:

```
1  extern _MessageBoxA@16
2
3  global _main
4
5  section .data
6      msg  db "Vulnerable Space",0
7      ttl  db "Welcome",0
8
9  section .text
10 _main:
11     push  dword 0x00       ; MB_OK = 0
12     push  dword ttl        ; "Welcome"
13     push  dword msg        ; "Vulnerable Space"
14     push  dword 0          ; handle to owner window
15     call  _MessageBoxA@16  ; in user32.dll
```

It might look convoluted at first but let's take it line by line:

extern _MessageBoxA@16

Tells our program that the MessageBoxA function requires importing from an external library (User32.dll). Later we'll see how this is done using GoLink.

global _main

global is the other end of extern; for a function to be importable by external programs using extern, it has to be explicitly defined as being global. _main is the name of our main function.

section .data

This section contains initialized static variables (i.e. global variables) and static local variables.

msg db "Vulnerable Space",0

Declares a variable called msg (db : declare byte) and initializes it to "Vulnerable Space" with a NULL byte at the end. Strings in C have to end with 0x00.

ttl db "Welcome",0

Same as above : "Welcome/00".

section .text

Contains the actual machine instructions that make our program.

_main

The name of the main function our program will run.

push dword ????

The 4 push instructions place the 4 dword (double word = 4 bytes = 32 bits) parameters required by the message box function on the stack. As described earlier this is necessary when using the STDCALL calling convention. The first parameter specifies the type of message box (in this case it's an OK message box) while the last parameter tells the message box that it has no owner window . The other 2 are self-explanatory. More information can be found here.

call _MessageBoxA@16

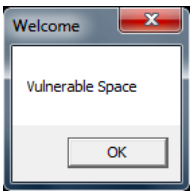
This performs the call to the message box function. The "_" (underscore) prefix is used to call the ASM version of the MessageBoxA C++ function whereas the @16 postfix signifies that 16 bytes of arguments are passed on the stack (4 dwords).

We now use NASM and GoLink to generate the executable, and run it:

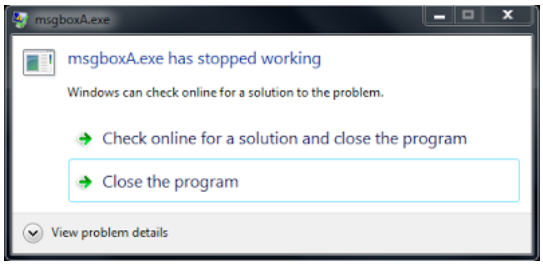
```
                                Command Prompt

C:\>nasm -fwin32 msgboxA.asm

C:\>GoLink /entry _main msgboxA.obj user32.dll

GoLink.Exe Version 1.0.1.0 - Copyright Jeremy Gordon 2002-2014 - JG@JGnet.co.uk
Output file: msgboxA.exe
Format: Win32   Size: 2,048 bytes

C:\>msgboxA.exe
```

The -fwin32 switch is used to generate Win32 format Object Files. For GoLink we specify the entry point function with the /entry switch, the COFF file generated by NASM, and the libraries required.

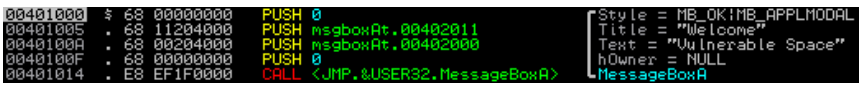
Congrats on your first ASM program! The result is as expected, an OK message box with "Vulnerable Space" as the caption and a "Welcome" title:



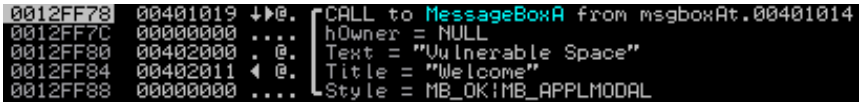
A few seconds later we get an unexpected(?) crash:



Before we polish our program, it's interesting to look at the resultant PE under a debugger. Here we can see the exact same instructions we have specified in the .text section. Conveniently, Immunity debugger maps them to the strings we have initialized them to in the .data section.



This is the state of the stack after all the parameters have been pushed onto it and just before the MessageBox function starts executing:



The last thing we'll look into before ending the tutorial is how to get rid of the crash. This happens because, after the MessageBox function ends its execution, we do not specify what the program should do. The function that will solve this issue is another Win32 API function called ExitProcess and resides in Kernel32.dll. More information on this function can be found here.

The final code looks like this:

```
;nasm -fwin32 msgboxA.asm
;GoLink /entry _main msgboxA.obj user32.dll kernel32.dll
;msgboxA.exe

extern _ExitProcess@4
extern _MessageBoxA@16

global _main

section .data
    msg  db "Vulnerable Space",0
    ttl  db "Welcome",0

section .text
_main:
    push  dword 0x00       ; MB_OK = 0
    push  dword ttl        ; "Welcome"
    push  dword msg        ; "Vulnerable Space"
    push  dword 0          ; handle to owner window
    call  _MessageBoxA@16  ; in user32.dll

    push  0                ; no error
    call  _ExitProcess@4   ; in kernel32.dll
```

Because of _ExitProcess@4, we now need to add "kernel32.dll" when using GoLink. Also notice that we need not clean the stack between function calls. This is a very convenient advantage of the STDCALL calling convention (used by Win32 API functions) in which the callee is expected to clear the stack rather than the caller program.

Conclusion

Hope you've all enjoyed it and feel proud of your first ASM program. You really should! Understanding ASM is no easy feat. As mentioned in the intro, I would love to hear your feedback; any improvements, clarifications, missing detail or suggestions.

In the next post we'll look at some more ASM instructions and try to build our own function .. またね

Posted by Francesco Mifsud at 13:02 G+1 Recommend this on Google

1 comment:

ggyy 12 September 2015 at 23:17

Thank the Tutorial!!! Nice job!!!!

Reply

Enter your comment...

Comment as: Google Account

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)