Nytro Security

Personal security research blog

Writing shellcodes for Windows x64

ON 30 JUNE 2019 / BY NYTROSECURITY

Long time ago I wrote three detailed blog posts about how to write shellcodes for Windows (x86 – 32 bits). The articles are beginner friendly and contain a lot of details. First part explains what is a shellcode and which are its limitations, second part explains PEB (Process Environment Block), PE (Portable Executable) file format and the basics of ASM (Assembler) and the third part shows how a Windows shellcode can be actually implemented.

This blog post is the port of the previous articles on Windows 64 bits (x64) and it will not cover all the details explained in the previous blog posts, so who is not familiar with all the concepts of shellcode development on Windows must see them before going further.

Of course, the differences between x86 and x64 shellcode development on Windows, including ASM, will be covered here. However, since I already write some details about Windows 64 bits on the Stack Based Buffer Overflows on x64 (Windows) blog post, I will just copy and paste them here.

As in the previous blog posts, we will create a simple shellcode that swaps the mouse buttons using SwapMouseButton function exported by user32.dll and grecefully close the proccess using ExitProcess function exported by kernel32.dll.

ASM for x64

There are multiple differences in Assembly that need to be understood in order to proceed. Here we will talk about the most important changes between x86 and x64 related to what we are going to do.

Please note that this article is for educational purposes only. It has to be simple, meaning that, of course, there are a lot of optimizations that can be done for the resulted shellcode to be smaller and faster.

First of all, the registers are now the following:

- The general purpose registers are the following: RAX, RBX, RCX, RDX, RSI, RDI, RBP and RSP. They are now 64 bit (8 bytes) instead of 32 bits (4 bytes).
- The EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP represent the last 4 bytes of the previously mentioned registers. They hold 32 bits of data.
- There are a few new registers: R8, R9, R10, R11, R12, R13, R14, R15, also holding 64 bits.
- It is possible to use R8d, R9d etc. in order to access the last 4 bytes, as you can do it with EAX, EBX etc.
- Pushing and poping data on the stack will use 64 bits instead of 32 bits

Calling convention

Another important difference is the way functions are called, the calling convention.

Here are the most important things we need to know:

- First 4 parameters are not placed on the stack. First 4 parameters are specified in the RCX, RDX, R8 and R9 registers.
- If there are more than 4 parameters, the other parameters are placed on the stack, from left to right.
- Similar to x86, the return value will be available in the RAX register.
- The function caller will allocate stack space for the arguments used in registers (called "shadow space" or "home space"). Even if when a function is called the parameters are placed in registers, if the called function needs to modify the registers, it will need some space to store them, and this space will be the stack. The function caller will have to allocate this space before the function call and to deallocate it after the function call. The function caller should allocate at least 32 bytes (for the 4 registers), even if they are not all used.
- The stack has to be 16 bytes aligned before any call instruction. Some functions might allocate 40 (0x28) bytes on the stack (32 bytes for the 4 registers and 8 bytes to align the stack from previous usage – the

return RIP address pushed on the stack) for this purpose. You can find more details here.

- Some registers are volatile and other are nonvolatile. This means that if we set some values into a register and call some function (e.g. Windows API) the volatile register will probably change while nonvolatile register will preserve their values.

More details about calling convention on Windows can be found here.

Function calling example

Let’s take a simple example in order to understand those things. Below is a function that does a simple addition, and it is called from main.

```
#include "stdafx.h"

int Add(long x, int y)
{
    int z = x + y;
    return z;
}

int main()
{
    Add(3, 4);
    return 0;
}
```

Here is a possible output, after removing all optimizations and security features.

Main function:

```
sub rsp,28
mov edx,4
mov ecx,3
call <consolex64.Add>
xor eax,eax
add rsp,28
ret
```

We can see the following:

1. **sub rsp,28** – This will allocate 0x28 (40) bytes on the stack, as we discussed: 32 bytes for the register arguments and 8 bytes for alignment.
2. **mov edx,4** – This will place in EDX register the second parameter. Since the number is small, there is no need to use RDX, the result is the same.
3. **mov ecx,3** – The value of the first argument is place in ECX register.
4. **call <consolex64.Add>** – Call the "Add" function.
5. **xor eax,eax** – Set EAX (or RAX) to 0, as it will be the return value of main.
6. **add rsp,28** – Clears the allocated stack space.
7. **ret** – Return from main.

Add function:

```
mov dword ptr ss:[rsp+10],edx
mov dword ptr ss:[rsp+8],ecx
sub rsp,18
mov eax,dword ptr ss:[rsp+28]
mov ecx,dword ptr ss:[rsp+20]
add ecx,eax
mov eax,ecx
mov dword ptr ss:[rsp],eax
mov eax,dword ptr ss:[rsp]
add rsp,18
ret
```

Let’s see how this function works:

1. **mov dword ptr ss:[rsp+10],edx** – As we know, the arguments are passed in ECX and EDX registers. But what if the function needs to use those registers (however, please note that some registers must be preserved by a function call, these registers are the following: RBX, RBP, RDI, RSI, R12, R13, R14 and R15)? In this case, the function will use the “shadow space” (“home space”) allocated by the function caller.

With this instruction, the function saves on the shadow space the second argument (the value 4), from EDX register.

2. **mov dword ptr ss:[rsp+8],ecx** – Similar to the previous instruction, this one will save on the stack the first argument (value 3) from the ECX register
3. **sub rsp,18** – Allocate 0x18 (or 24) bytes on the stack. This function does not call other function, so it is not needed to allocate at least 32 bytes. Also, since it does not call other functions, it is not required to align the stack to 16 bytes. I am not sure why it allocates 24 bytes, it looks like the "local variables area" on the stack has to be aligned to 16 bytes and the other 8 bytes might be used for the stack alignment (as previously mentioned).
4. **mov eax,dword ptr ss:[rsp+28]** – Will place in EAX register the value of the second parameter (value 4).
5. **mov ecx,dword ptr ss:[rsp+20]** – Will place in ECX register the value of the first parameter (value 3).
6. **add ecx,eax** – Will add to ECX the value of the EAX register, so ECX will become 7.
7. **mov eax,ecx** – Will save the same value (the sum) into EAX register.
8. **mov dword ptr ss:[rsp],eax** and **mov eax,dword ptr ss:[rsp]** look like they are some effects of the removed optimizations, they don't do anything useful.
9. **add rsp,18** – Cleanup the allocated stack space.
10. **ret** – Return from the function

Writing ASM on Windows x64

There are multiple ways to write assembler on Windows x64. I will use NASM and the linker provided by Microsoft Visual Studio Community.

I will use the x64.asm file to write the assembler code, the NASM will output x64.obj and the linker will create x64.exe. To keep this process simple, I created a simple Windows Batch script:

```
del x64.obj
del x64.exe
nasm -f win64 x64.asm -o x64.obj
link /ENTRY:main /MACHINE:X64 /NODEFAULTLIB /SUBSYSTEM:CONSOLE x64.o
```

You can run it using “x64 Native Tools Command Prompt for VS 2019” where “link” is available directly. Just not forget to add NASM binaries directory to the PATH environment variable.

To test the shellcode I open the resulted binary in x64bdg and go through the code step by step. This way, we can be sure everything is OK.

Before starting with the actual shellcode, we can start with the following:

```
BITS 64
SECTION .text
global main
main:

sub   RSP, 0x28                  ; 40 bytes of shadow space
and   RSP, 0FFFFFFFFFFFFFFF0h    ; Align the stack to a multiple of 1
```

This will specify a 64 bit code, with a "main" function in the ".text" (code) section. The code will also allocate some stack space and align the stack to a multiple of 16 bytes.

Find kernel32.dll base address

As we know, the first step in the shellcode development process for Windows is to find the base address of kernel32.dll, the memory address where it is loaded. This will help us to find its useful exported functions: GetProcAddress and LoadLibraryA which we can use to achive our goals.

We will start finding the TEB (Thread Environment Block), the structure that contains thread information in usermode and we can find it using GS register, ar **gs:[0x00]**. This structure also contains a pointer to the PEB (Process Envrionment Block) at offset **0x60**.

The PEB contains the "Loader" (Ldr) at offset **0x18** which contains the "InMemoryOrder" list of modules at offset **0x20**. As we did for x86, first module will be the executable, the second one ntdll.dll and the third one kernel32.dll which we want to find. This means we will go through a linked list (LIST_ENTRY structure which contains to LIST_ENTRY* pointers, Flink and Blink, 8 bytes each on x64).

After we find the third module, kernel32.dll, we just need to go to offset **0x20** to get its base address and we can start doing our stuff.

Below is how we can get the base address of kernel32.dll using PEB and store it in the RBX register:

```
; Parse PEB and find kernel32

xor rcx, rcx             ; RCX = 0
mov rax, [gs:rcx + 0x60] ; RAX = PEB
mov rax, [rax + 0x18]    ; RAX = PEB->Ldr
mov rsi, [rax + 0x20]    ; RSI = PEB->Ldr.InMemOrder
lodsq                    ; RAX = Second module
xchg rax, rsi            ; RAX = RSI, RSI = RAX
lodsq                    ; RAX = Third(kernel32)
mov rbx, [rax + 0x20]    ; RBX = Base address
```

Find the address of GetProcAddress function

It is really similar to find the address of GetProcAddress function, the only difference would be the offset of export table which is 0x88 instead of 0x78.

The steps are the same:

1. Go to the PE header (offset 0x3c)

2. Go to Export table (offset 0x88)
3. Go to the names table (offset 0x20)
4. Get the function name
5. Check if it starts with “GetProcA”
6. Go to the ordinals table (offset 0x24)
7. Get function number
8. Go to the address table (offset 0x1c)
9. Get the function address

Below is the code that can help us find the address of GetProcAddress:

```
; Parse kernel32 PE

xor r8, r8                ; Clear r8
mov r8d, [rbx + 0x3c]     ; R8D = DOS->e_lfanew offset
mov rdx, r8               ; RDX = DOS->e_lfanew
add rdx, rbx              ; RDX = PE Header
mov r8d, [rdx + 0x88]     ; R8D = Offset export table
add r8, rbx               ; R8 = Export table
xor rsi, rsi              ; Clear RSI
mov esi, [r8 + 0x20]      ; RSI = Offset namestable
add rsi, rbx              ; RSI = Names table
xor rcx, rcx              ; RCX = 0
mov r9, 0x41636f7250746547 ; GetProcA
```

```
; Loop through exported functions and find GetProcAddress

Get_Function:

inc rcx                     ; Increment the ordinal
xor rax, rax                ; RAX = 0
mov eax, [rsi + rcx * 4]    ; Get name offset
add rax, rbx                ; Get function name
cmp QWORD [rax], r9         ; GetProcA ?
jnz Get_Function
xor rsi, rsi                ; RSI = 0
mov esi, [r8 + 0x24]        ; ESI = Offset ordinals
add rsi, rbx                ; RSI = Ordinals table
mov cx, [rsi + rcx * 2]     ; Number of function
xor rsi, rsi                ; RSI = 0
mov esi, [r8 + 0x1c]        ; Offset address table
add rsi, rbx                ; ESI = Address table
xor rdx, rdx                ; RDX = 0
mov edx, [rsi + rcx * 4]    ; EDX = Pointer(offset)
add rdx, rbx                ; RDX = GetProcAddress
mov rdi, rdx                ; Save GetProcAddress in RDI
```

Please note that this has to be done carefully. Some structures from the PE file are not 8 bytes, while we need in the end 8 bytes pointers. This is why in the code above there are registers such as ESI or CX used.

Find the address of LoadLibraryA

Since we have the address of GetProcAddress and the base address of kernel32.dll, we can use them to call GetProcAddress(kernel32.dll, “LoadLibraryA”) and find the address of LoadLibraryA function.

However, it is something important we need to be careful about: we will use the stack to place our strings (e.g. “LoadLibraryA”) and this might break the stack alignment, so we need to make sure it is 16 bytes alligned. Also, we must not forget about the stack space that we need to allocate for a function call, because the function we call might use it. So, we need to place our string on the stack and just after this to allocate space for the function we call (e.g. GetProcAddress).

Finding the address of LoadLibraryA is pretty straightforward:

```
; Use GetProcAddress to find the address of LoadLibrary

mov rcx, 0x41797261           ; aryA
push rcx                      ; Push on the stack
mov rcx, 0x7262694c64616f4c   ; LoadLibr
push rcx                      ; Push on stack
mov rdx, rsp                  ; LoadLibraryA
mov rcx, rbx                  ; kernel32.dll base address
sub rsp, 0x30                 ; Allocate stack space for function cal
call rdi                      ; Call GetProcAddress
add rsp, 0x30                 ; Cleanup allocated stack space
add rsp, 0x10                 ; Clean space for LoadLibrary string
mov rsi, rax                  ; LoadLibrary saved in RSI
```

We put the “LoadLibraryA” string on the stack, setup RCX and RDX registers, allocate space on the stack for the function call, call GetProcAddress and cleanup the stack. As a result, we will store the LoadLibraryA address in the RSI register.

Load user32.dll using LoadLibraryA

Since we have the address of LoadLibraryA function, it is pretty simple to call LoadLibraryA(“user32.dll”) to load user32.dll and find out its base address which will be returned by LoadLibraryA.

```
mov rcx, 0x6c6c                ; ll
push rcx                       ; Push on the stack
mov rcx, 0x642e323372657375    ; user32.d
push rcx                       ; Push on stack
mov rcx, rsp                   ; user32.dll
sub rsp, 0x30                  ; Allocate stack space for function ca
call rsi                       ; Call LoadLibraryA
add rsp, 0x30                  ; Cleanup allocated stack space
add rsp, 0x10                  ; Clean space for user32.dll string
mov r15, rax                   ; Base address of user32.dll in R15
```

The function will return the base address of the user32.dll module into RAX and we will save it in the R15 register.

Find the address of SwapMouseButton function

We have the address of GetProcAddress, the base address of user32.dll and we know the function is called “SwapMouseButton”. So we just need to call GetProcAddress(user32.dll, “SwapMouseButton”);

Please note that when we allocate space on stack for the function call, we do not allocate anymore 0x30 (48) bytes, we allocate only 0x28 (40) bytes. This is because to place our string (“SwapMouseButton”) on the stack we use 3 PUSH instructions, so we get 0x18 (24) bytes of data, which is not a multiple of 16. So we use 0x28 instead of 0x30 to align the stack to 16 bytes.

```
; Call GetProcAddress(user32.dll, "SwapMouseButton")

xor rcx, rcx                  ; RCX = 0
push rcx                      ; Push 0 on stack
mov rcx, 0x6e6f7474754265     ; eButton
push rcx                      ; Push on the stack
mov rcx, 0x73756f4d70617753   ; SwapMous
push rcx                      ; Push on stack
mov rdx, rsp                  ; SwapMouseButton
mov rcx, r15                  ; User32.dll base address
sub rsp, 0x28                 ; Allocate stack space for function ca
call rdi                      ; Call GetProcAddress
add rsp, 0x28                 ; Cleanup allocated stack space
add rsp, 0x18                 ; Clean space for SwapMouseButton stri
mov r15, rax                  ; SwapMouseButton in R15
```

GetProcAddress will return in RAX the address of SwapMouseButton function and we will save it into R15 register.

Call SwapMouseButton

Well, we have its address, it should be pretty easy to call it. We do not have any issue as we previously cleaned up and we do not need to alter the stack in this function call. So we just set the RCX register to 1 (meaning true) and call it.

```
; Call SwapMouseButton(true)

mov rcx, 1      ; true
call r15        ; SwapMouseButton(true)
```

Find the address of ExitProcess function

As we already did before, we use GetProcAddress to find the address of ExitProcess function exported by the kernel32.dll. We still have the kernel32.dll base address in RBX (which is a nonvolatile register and this is why it is used) so it is simple:

```
; Call GetProcAddress(kernel32.dll, "ExitProcess")

xor rcx, rcx                 ; RCX = 0
mov rcx, 0x737365            ; ess
push rcx                     ; Push on the stack
mov rcx, 0x636f725074697845  ; ExitProc
push rcx                     ; Push on stack
mov rdx, rsp                 ; ExitProcess
mov rcx, rbx                 ; Kernel32.dll base address
sub rsp, 0x30                ; Allocate stack space for function cal
call rdi                     ; Call GetProcAddress
add rsp, 0x30                ; Cleanup allocated stack space
add rsp, 0x10                ; Clean space for ExitProcess string
mov r15, rax                 ; ExitProcess in R15
```

We save the address of ExitProcess function in R15 register.

ExitProcess

Since we do not want to let the process to crash, we can “gracefully” exit by calling the ExitProcess function. We have the address, the stack is aligned, we have just to call it.

```
; Call ExitProcess(0)

mov rcx, 0      ; Exit code 0
call r15        ; ExitProcess(0)
```

Conclusion

There are many articles about Windows shellcode development on x64, such as this one or this one, but I just wanted to tell the story my way, following the previously written articles.

The shellcode is far away from being optimized and it also contains NULL bytes. However, both of these limitations can be improved.

Shellcode development is fun and swithing from x86 to x64 is needed, because x86 will not be used too much in the future.

Or course, I will add support for Windows x64 in Shellcode Compiler.

If you have any question, please add a comment or contact me.

Share this:

Facebook Twitter LinkedIn Reddit Email Print

PREVIOUS

Network scanning with nmap

4 thoughts on "Writing shellcodes for Windows x64"

Pingback: Writing shellcodes for Windows x64 | | Lowmiller Consulting Group Blog

Pingback: Writing shellcodes for Windows x64 - ZRaven Consulting

Pingback: Writing shellcodes for Windows x64 – Trusted News Feeds 2.0

Pingback: Writing shellcodes for Windows x64 – Nytro Security – The Library 6.0

Leave a Reply

Enter your comment here...

Search …

RECENT POSTS

Writing shellcodes for Windows x64

Network scanning with nmap

Understanding Java deserialization

NetRipper at BlackHat Asia Arsenal 2018

Hooking Chrome’s SSL functions

Stack Based Buffer Overflows on x64 (Windows)

Stack Based Buffer Overflows on x86 (Windows) – Part II

Stack Based Buffer Overflows on x86 (Windows) – Part I

Hello, world!

BLOG STATS

31,794 hits

RSS

RSS - Posts