Odzhan

DLL/PIC Injection on Windows from Wow64 process

Posted on November 19, 2015

Introduction

Injecting PIC (Position Independent Code) into a remote process is trivial enough for a programmer but if they try using CreateRemoteThread() API from Wow64 against a 64-bit process, it fails.

Transitioning from 32-bit to 64-bit was discussed by rgb/29a in his article Heaven's Gate: 64-bit code in 32-bit file around 2009.

ReWolf has also written extensively on this issue and published a helper library in C/ASM which enables x86 applications to read/write to x64 applications in addition to calling any NTDLL.DLL API using his X64Call() function which is probably the best solution for developers that want to solve the problem without hacking assembly.

There's lots of information about what I'm discussing here and there's a freely available open source solution if compiling 2 separate binaries isn't to your liking.

This is only going to document a DLL/PIC injection tool called 'pi' written for the purpose of testing win32/win64 shellcodes.

It was written because sometimes, you need to execute code from 32-bit application in 64-bit process.

Traditional method

The steps familiar to those struggling with the problem:

API	Description
OpenProcess	self explanatory, open the process we want to inject PIC into
VirtualAllocEx	allocate read/write/executable memory for our PIC
WriteProcessMemory	write PIC code
VirtualProtectEx	optionally change the memory to read/execute only
CreateRemoteThread	run the PIC code as a thread
WaitForSingleObject	optionally wait for the thread to exit (or crash!)

All is well until you hit CreateRemoteThread. Even if you use NtCreateThreadEx() it still won't work. (open to correction)

To circumvent the limitation, Wow64 process will transition to 64-bit as demonstrated by rgb/29a and ReWolf with their articles/code, resolve the API NtCreateThreadEx() and execute thread in remote process.

Wow64 detection

Various methods of detecting wow64/64-bit using assembly have been suggested over the last number of years.

Peter Ferrie published one that uses REX prefixes and there are undoubtedly many other methods published on the internet.

isWow64 also uses a REX prefix 0x48 which under 32-bit mode is “DEC EAX”

When this code executes in 32-bit mode, it will return TRUE, else FALSE for 64-bit mode. If calling from ASM, you can simply check ZF (zero flag) after it returns.

```
  global isWow64
  global _isWow64

  ; returns TRUE or FALSE
isWow64:
_isWow64:
    bits   32
    xor    eax, eax
    dec    eax
    neg    eax
    ret
```

32-bit mode: xor eax, eax makes eax zero. dec eax makes it -1. neg eax makes it 1. return 1 in eax

```
/* 0000 */ "\x31\xc0" /* xor eax, eax */
/* 0002 */ "\x48"     /* dec eax      */
/* 0003 */ "\xf7\xd8" /* neg eax      */
/* 0005 */ "\xc3"     /* ret          */
```

64-bit mode: xor eax, eax makes rax zero. neg rax does nothing. return zero in rax.

```
/* 0000 */ "\x31\xc0"     /* xor eax, eax */
/* 0002 */ "\x48\xf7\xd8" /* neg rax      */
/* 0005 */ "\xc3"         /* ret          */
```

The following by Peter Ferrie uses 0x41 REX prefix which is “inc ecx” in 32-bit mode.

```
bits 32
; if ZF is 1, we're 64-bit, else 32-bit
is64:
_is64:
    xor    ecx, ecx
    inc    ecx
    xchg   eax, ecx
    ret
```

32-bit mode

```
/* 0000 */ "\x31\xc9" /* xor ecx, ecx  */
/* 0002 */ "\x41"     /* inc ecx       */
/* 0003 */ "\x91"     /* xchg ecx, eax */
/* 0004 */ "\xc3"     /* ret           */
```

64-bit mode

```
/* 0000 */ "\x31\xc9" /* xor ecx, ecx  */
/* 0002 */ "\x41\x91" /* xchg r9d, eax */
/* 0004 */ "\xc3"     /* ret           */
```

Obtaining API address

Rather than search the NTDLL export table for LdrGetProcAddress and pass the string “NtCreateThreadEx”, we search for the hash of this string using old simple hash algorithm originally suggested by LSD-PL in their winasm

presentation, published in 2002

I found a nifty NASM macro originally written by Vecna/29a and converted to NASM syntax by Jibz, the author of apLib.

The getapi function obtains NTDLL from the 64-bit PEB (Process Environment Block) and then searches through the export table for required function.

```
%define ROL_N 5

%macro HASH 1.nolist
  %assign %%h 0
  %strlen %%len %1
  %assign %%i 1
  %rep %%len
    %substr %%c %1 %%i
    %assign %%h ((%%h + %%c) & 0FFFFFFFFh)
    %assign %%h ((%%h << ROL_N) & 0FFFFFFFFh) | (%%h >> (32-ROL_N))
    %assign %%i (%%i+1)
  %endrep
  %assign %%h ((%%h << ROL_N) & 0FFFFFFFFh) | (%%h >> (32-ROL_N))
  dd %%h
%endmacro

; mov eax, HASH "string"
%macro hmov 1.nolist
  db 0B8h
  HASH %1
%endmacro

getapi:
    bits   64
    push   rsi
    push   rdi
    push   rbx
    push   rcx

    mov    r8, rax
    push   60h
    pop    rsi
    mov    rax, qword [gs:rsi]
    mov    rax, [rax+18h]
    mov    r10, [rax+30h]
l_dll:
    mov    rbp, [r10+10h]
    test   rbp, rbp
    mov    eax, ebp
    jz     xit_getapi
    mov    r10, [r10]

    mov    eax, [rbp+3Ch]      ; IMAGE_DOS_HEADER.e_lfanew
    add    eax, 10h
    mov    eax, [rbp+rax+78h]
    lea    rsi, [rbp+rax+18h]  ; IMAGE_EXPORT_DIRECTORY.NumberOfNames
    lodsd
    xchg   eax, ecx
    jecxz  l_dll

    lodsd                  ; IMAGE_EXPORT_DIRECTORY.AddressOfFunctions

    ; EMET will break on the following instruction
    lea    r11, [rbp+rax]

    lodsd                  ; IMAGE_EXPORT_DIRECTORY.AddressOfNames
    lea    rdi, [rbp+rax]

    lodsd                  ; IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
    lea    rbx, [rbp+rax]
```

```
l_api:
    mov    esi, [rdi+4*rcx-4]
    add    rsi, rbp
    xor    eax, eax
    cdq
h_api:
    lodsb
    add    edx, eax
    rol    edx, ROL_N
    dec    eax
    jns    h_api

    cmp    edx, r8d

    loopne l_api
    jne    l_dll

    movzx  edx, word [rbx+2*rcx]
    mov    eax, [r11+4*rdx]
    add    rax, rbp
xit_getapi:
    pop    rcx
    pop    rbx
    pop    rdi
    pop    rsi
    ret
```

Switching to 64-bit mode

Again, this has been described/documented by rgb/29a and ReWolf very well and it's how I wrote the following function.

```
  bits 32
  ; switch to x64 mode
sw64:
    call   isWow64
    jz     ext64                ; we're already x64
    pop    eax                  ; get return address
    push   33h                  ; x64 selector
    push   eax                  ; return address
    retf                        ; go to x64 mode
ext64:
    ret
```

Switching back to x86 mode

This piece of code returns to 32-bit mode after we've created thread.

```
  bits 32
  ; switch to x86 mode
sw32:
    call   isWow64
    jnz    ext32                ; we're already x86
    pop    eax
    sub    esp, 8
    mov    dword[esp], eax
    mov    dword[esp+4], 23h    ; x86 selector
    retf
ext32:
    ret
```

CreateRemoteThread

The following is a wrapper for calling NtCreateThreadEx. It first switches to x64 mode, then calls the function before returning to 32-bit mode.

The reason to use a structure when loading arguments for NtCreateThreadEx() is that it's easier to align the stack.

The function needs somewhere to store handle of thread since we can't use global data, so that's why you see hThread at the bottom (or top depending on how you look at it)

The stack must be aligned by 16 bytes minus 8 before calling an API otherwise it'll cause problems and occasionally crash.

If you're wondering why aligned by 16 minus 8. The eight bytes will be occupied by return address once call is executed.

The HOME_SPACE structure is required for all API.

Have a look at [The history of calling conventions, part 5: amd64] for more information.

```
struc HOME_SPACE
  ._rcx resq 1
  ._rdx resq 1
  ._r8  resq 1
  ._r9  resq 1
  .size:
endstruc

struc ct_stk
  .hs: resb HOME_SPACE.size

  .lpStartAddress     resq 1
  .lpParameter        resq 1
  .CreateSuspended    resq 1
  .StackZeroBits      resq 1
  .SizeOfStackCommit  resq 1
  .SizeOfStackReserve resq 1
  .lpBytesBuffer      resq 1

  .hThread            resq 1
  .size:
endstruc

%ifndef BIN
    global CreateRemoteThread64
    global _CreateRemoteThread64
%endif
CreateRemoteThread64:
_CreateRemoteThread64:
    bits 32
    push   ebx
    push   esi
    push   edi
    push   ebp

    call   sw64                 ; switch to x64 mode
    test   eax, eax             ; we're already in x64 mode and will only work with
    jz     exit_create

    bits   64
    mov    rsi, rsp
    and    rsp, -16
    sub    rsp, ((ct_stk.size & -16) + 16) - 8

    hmov   "NtCreateThreadEx"
    call   getapi
    mov    rbx, rax

    xor    r8, r8
    xor    rax, rax

```

```
52      mov     [rsp+ct_stk.lpBytesBuffer     ], rax ; NULL
53      mov     [rsp+ct_stk.SizeOfStackReserve], rax ; NULL
54      mov     [rsp+ct_stk.SizeOfStackCommit ], rax ; NULL
55      mov     [rsp+ct_stk.StackZeroBits     ], rax ; NULL
56
57      mov     [rsp+ct_stk.CreateSuspended   ], rax
58
59      mov     eax, [rsi+9*4]            ; lpParameter
60      mov     [rsp+ct_stk.lpParameter       ], rax
61
62      mov     eax, [rsi+8*4]            ; lpStartAddress
63      mov     [rsp+ct_stk.lpStartAddress    ], rax
64
65      mov     r9d, [rsi+5*4]            ; hProcess
66      mov     edx, 10000000h            ; GENERIC_ALL
67      mov     ecx, [rsi+12*4]           ; &hThread
68      call    rbx
69
70      mov     rsp, rsi                  ; restore stack value
71
72      call    sw32                      ; switch back to x86 mode
73
74  exit_create:
75      bits    32
76      pop     ebp
77      pop     edi
78      pop     esi
79      pop     ebx
80      ret
```

Calling from C

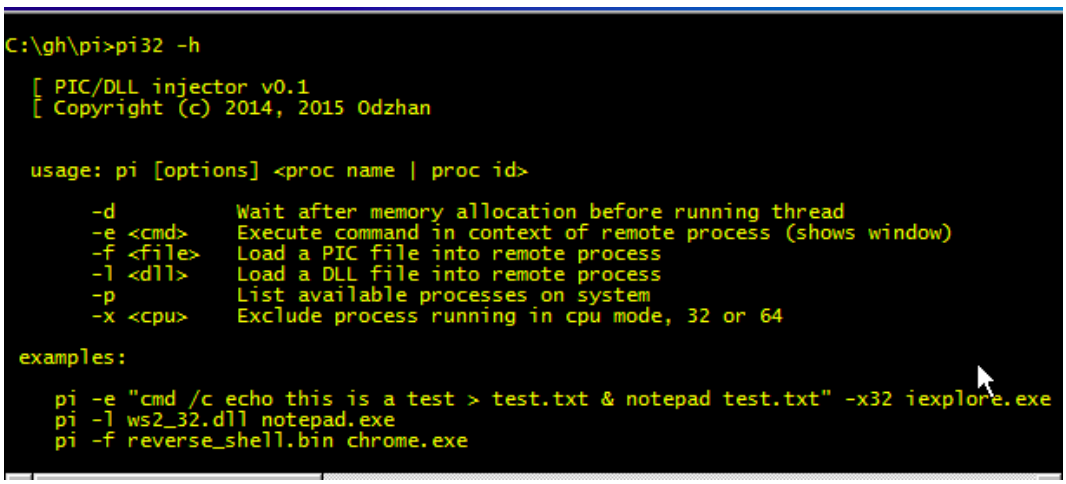
Initially, the CreateRemoteThread64() function was linked with pi.c but I decided to assemble as binary and convert to a C string to keep it simple.

Write/Executable memory is allocated by VirtualAlloc before the string is copied over and executed.

All steps required to inject into remote process using the API described are the same as before with just one exception…

If pi is running as 32-bit and remote process is 64-bit, we call CreateRemoteThread64() else CreateRemoteThread().

usage



The tool still needs testing/developing but I've uploaded source/binaries to github if you're interested.

Future work

Shellcodes for 32/64-bit windows are difficult to write and the process of testing can be frustrating. An ideal addition to a tool like this would be a debugger for both 32 and 64-bit processes.

It doesn't necessarily have to be interactive, just capable of handling exceptions, disassembling the address/code where the exception occurred before cleaning up, leaving the target application intact; that would save a lot of time.

The problem is you can't debug a 64-bit process from 32-bit, at least not directly through the Windows user API.

Share this:

Related

Asmcodes: Platform Independent PIC for Loading DLL and Executing Commands
In "assembly"

Asmcodes: MD4
In "assembly"

About Odzhan

I like optimizing assembly code and have an interest in computer security.

View all posts by Odzhan →

This entry was posted in assembly, programming, security and tagged assembly, dll injection, heavens gate, pic, wow64, x64, x86. Bookmark the permalink.

2 Responses to *DLL/PIC Injection on Windows from Wow64 process*

Peter Ferrie *says:*

November 21, 2015 at 10:10 pm

That "dec eax" instruction that you have is a REX prefix in 64-bit mode. There is no one-byte dec instruction in 64-bit mode.

windbg can debug code that transitions between 32-bit and 64-bit code from user-mode.

Like

Reply

Odzhan *says:*

November 22, 2015 at 12:41 am

Hi Peter. Is it possible to debug a 64-bit process from wow64 though? I couldn't get it to work.

"dec eax/neg eax" becomes "neg rax" in 64-bit mode. I should be bit more clearer about this part.

★ Like

Reply

Odzhan