Code Injection (Process Injection)

https://t.me/learningnets

Code Injection

- The technique of injecting malicious code into a target process's memory and executing the malicious code within the context of the target process is called code injection (or process injection).

- An attacker typically chooses a legitimate process (such as ***explorer.exe*** or ***svchost.exe***) as the target process. Once the malicious code is injected into the target process, it can then perform malicious actions

- After injecting the code into the memory of the target process, the malware component responsible for injecting code can either continue to persist on the system, thereby injecting code into the target process every time the system reboots, or it can delete itself from the filesystem, keeping the malicious code in-memory only.

https://t.me/learningnets

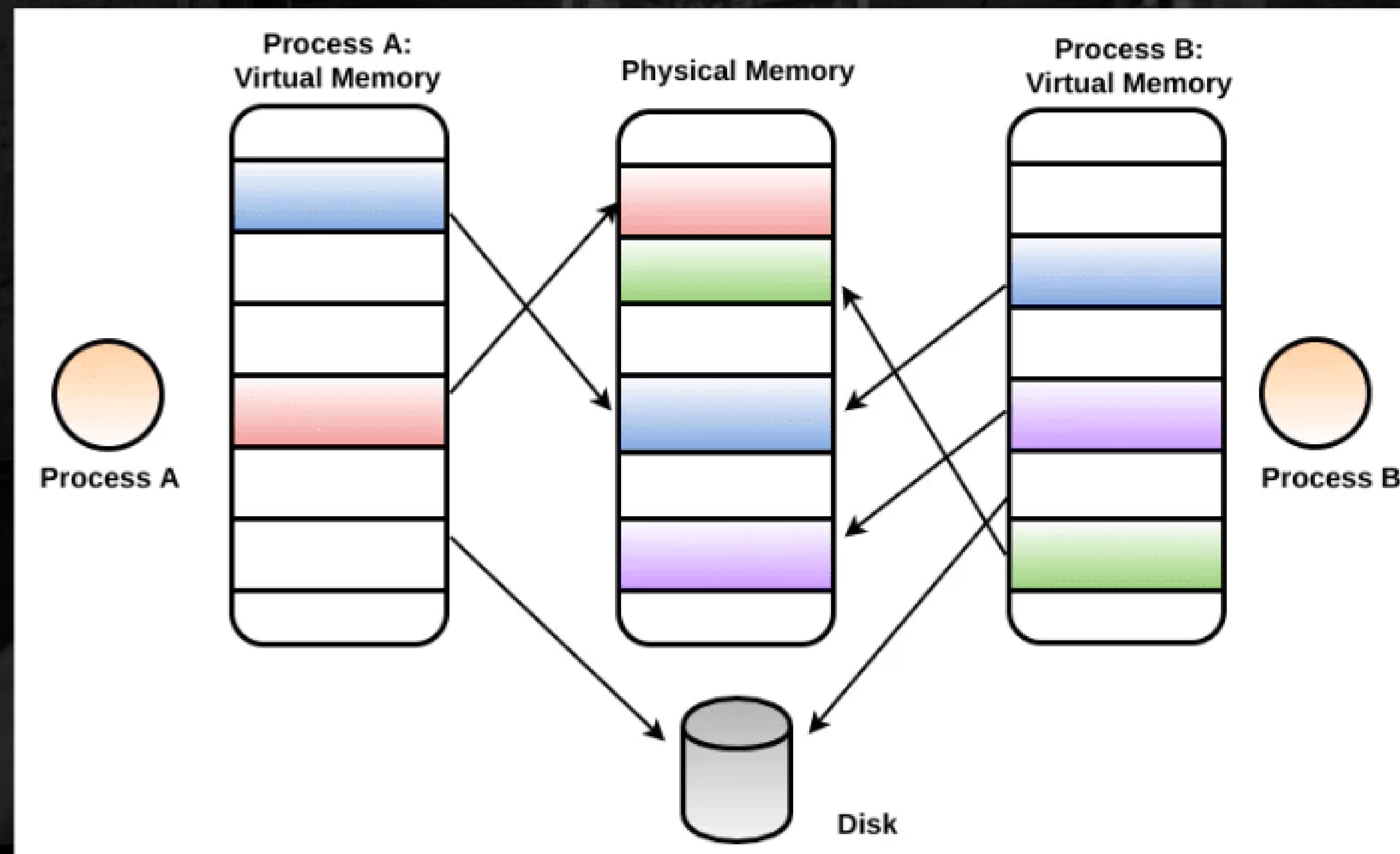
The injected code could be a module such as an executable, DLL, or even shellcode.

Once the code is injected into the remote process, an adversary can do the following things:

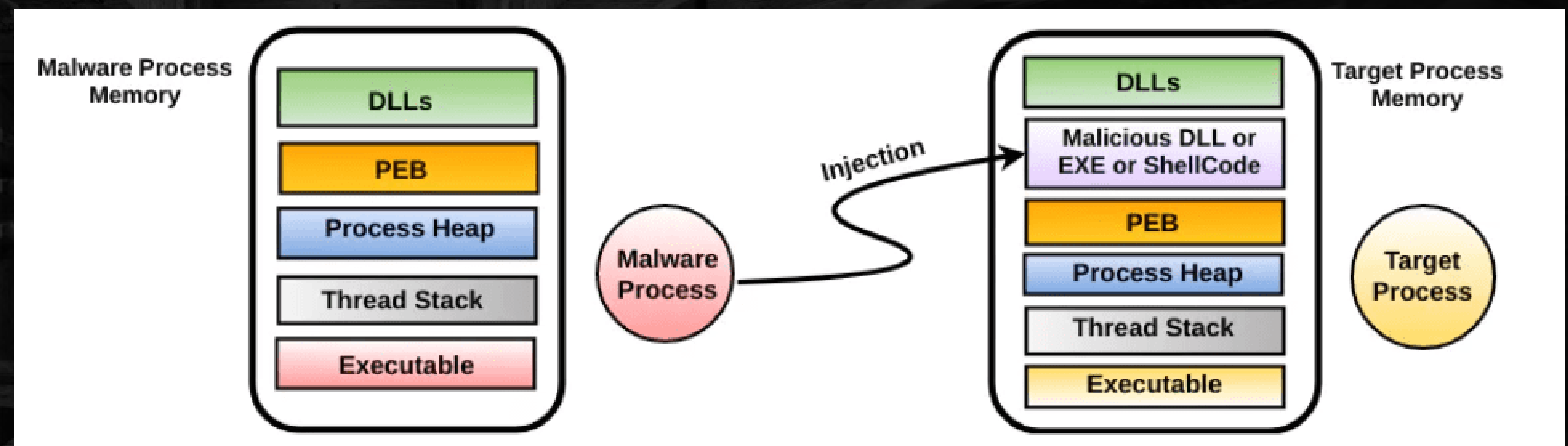
- Force the remote process to execute the injected code to perform malicious actions (such as downloading additional files or stealing keystrokes).
- Inject a malicious module (such as a DLL) and redirect the API call made by the remote process to a malicious function in the injected module. The malicious function can then intercept the input parameters of the API call, and also filter the output parameters.
- Injecting code into an already running process allows an adversary to achieve persistence.
- Injecting code into trusted processes allows an attacker to bypass security products (such as whitelisting software) and hide from the user.

Virtual Memory

Windows operating system provides each new process that is created, with its own private memory address space (called the ***process memory***). The process memory is a part of virtual memory; virtual memory is not real memory, but an illusion created by the operating system's memory manager.



The following screenshot should give you a high-level overview of code injection techniques in the user-space:



https://t.me/learningnets

Remote Executable/Shellcode Injection

- The malicious code is injected into the target process memory directly, without dropping the component on the disk.
- The injected malicious code is forced to execute by creating a remote thread via ***CreateRemoteThread()***, and the start of the thread is made to point to the code/function within the injected block of code.
- The advantage of this method is that the malware process does not have to drop the malicious DLL on the disk; it can extract the code to inject from the resource section of the binary, or get it over the network and perform code injection directly.

Demo 11 - Remote Executable Injection

https://t.me/learningnets

Remote DLL Injection

https://t.me/learningnets

Remote DLL Injection

In this injection technique, the malware process creates a thread in the target process and the thread is made to call ***LoadLibrary()*** by passing a malicious DLL path as the argument. Since the thread gets created in the target process, the target process loads the malicious DLL into its address space. Once the target process loads the malicious DLL, the operating system automatically calls the DLL's ***DllMain()*** function, thus executing the malicious code.

Step 1:

The malware process identifies the target process and opens a handle to the target process. In the following screenshot, the malware calls ***OpenProcess()*** by passing the pid of ***explorer.exe*** (***0x624***, which is ***1572***) as the third parameter. The return value of ***OpenProcess()*** is the handle to the ***explorer.exe*** process:

```
    0040143D  53                  push ebx
    0040143E  6A 00               push 0
    00401440  68 FF FF 1F 00      push 1FFFFF
EIP 00401445  FF 15 10 A0 40 00   call dword ptr ds:[<&OpenProcess>]
    0040144B  8B F8               mov edi,eax
    0040144D  85 FF               test edi,edi
    0040144F  75 1E               jne nps.40146F

EAX  0000003C   '<'
EBX  00000624   L'Ƥ'

Default (stdcall)
1: [esp] 001FFFFF
2: [esp+4] 00000000
3: [esp+8] 00000624
```

Step 2:

Malware process allocates memory in the target process using the ***VirutualAllocEx()*** API. The ***1st*** argument (***0x30***) is the handle to ***explorer.exe*** (the target process). The ***3rd*** argument, ***0x27*** (***39***), represents the number of bytes to be allocated in the target process, and the ***5th*** argument (***0x4***) is a constant value that represents the memory protection of ***PAGE_READWRITE***. The return value of ***VirtualAllocEx()*** is the address of the allocated memory in explorer.exe

```
    0040148C   6A 04             push 4
    0040148E   68 00 10 00 00    push 1000
    00401493   56                push esi
    00401494   6A 00             push 0
    00401496   57                push edi
EIP 00401497   FF 15 58 A0 40 00 call dword ptr ds:[<&VirtualAllocEx>]
    0040149D   8B D8             mov ebx,eax
    0040149F   85 DB             test ebx,ebx
    004014A1   75 1E             jne nps.4014C1

EAX   0000005F   '_'
EBX   00000624   L'Ĥ'

Default (stdcall)
1: [esp] 00000030
2: [esp+4] 00000000
3: [esp+8] 00000027
4: [esp+C] 00001000
5: [esp+10] 00000004
```

Step 3:

Malware uses WriteProcessMemory() to copy the DLL pathname to the allocated memory in the target process. The *2nd* argument, *0x01E30000*, is the address of the allocated memory in the target process, and the *3rd* argument is the full path to the DLL that will be written to the allocated memory address *0x01E30000* in *explorer.exe*

```
56                  push esi
FF B5 DC FB FF FF   push dword ptr ss:[ebp-424]
53                  push ebx
57                  push edi
FF 15 4C A0 40 00   call dword ptr ds:[<&WriteProcessMemory>]
85 C0               test eax,eax
75 2A               jne nps.401505
```

```
EAX 0012F674
EBX 01E30000
Default (stdcall)
1: [esp] 00000030
2: [esp+4] 01E30000
3: [esp+8] 0012FABC "C:\\Users\\test\\AppData\\Roaming\\adpr.dll"
4: [esp+C] 00000027
```

Step 4:

Malware determines the address of ***LoadLibrary()*** in ***kernel32.dll***; to do that, it calls the ***GetModuleHandleA()*** API and passes ***kernel32.dll*** as the argument, which will return the base address of ***Kernel32.dll***. Once it gets the base address of ***kernel32.dll***, it determines the address of ***LoadLibrary()*** by calling ***GetProcessAddress()***

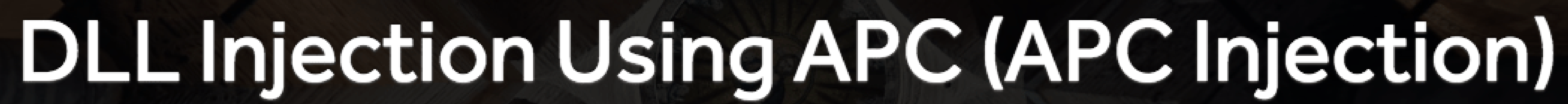
Step 5:

Malware calls ***CreateRemoteThread()*** which creates a thread in the target process. The ***1st*** argument, ***0x30***, to ***CreateRemoteThread()*** is the handle to the ***explorer.exe*** process, in which the thread will be created. The ***4th*** argument is the address in the target process memory where the thread will start executing, which is the address of ***LoadLibrary()***, and the ***5th*** argument is the address in the target process memory that contains the full path to the DLL.

```
004015D9   50                 push eax
004015DA   50                 push eax
004015DB   53                 push ebx
004015DC   56                 push esi
004015DD   50                 push eax
004015DE   50                 push eax
004015DF   57                 push edi
EIP 004015E0   FF 15 30 A0 40 00  call dword ptr ds:[<&CreateRemoteThread>]
004015E6   8B F0              mov esi,eax
```

```
EAX  00000000
EBX  01E30000

Default (stdcall)
1: [esp]    00000030
2: [esp+4]  00000000
3: [esp+8]  00000000
4: [esp+C]  7791DE15 <kernel32.LoadLibraryA>
5: [esp+10] 01E30000
6: [esp+14] 00000000
```



https://t.me/learningnets

DLL Injection Using APC (APC Injection)

- The APC injection technique is similar to remote DLL injection, but instead of using ***CreateRemoteThread()***, a malware makes use of ***Asynchronous Procedure Calls (APCs)*** to force the thread of a target process to load the malicious DLL.
- An APC is a function that executes asynchronously in the context of a particular thread. Each thread contains a queue of APCs that will be executed when the target thread enters an alertable state.
- A thread enters an alertable state if it calls one of the following functions:

```
SleepEx(),
SignalObjectAndWait()
MsgWaitForMultipleObjectsEx()
WaitForMultipleObjectsEx()
WaitForSingleObjectEx()
```

- The way APC injection technique works is, a malware process identifies the thread in the target process (the process in which the code will be injected) that is in an alertable state, or likely to go into an alertable state.

- It then places the custom code in that thread's APC queue by using the ***QueueUserAPC()*** function.

- The idea of queueing the custom code is that, when the thread enters the alertable state, the custom code gets picked up from the APC queue, and it gets executed by the thread of the target process.

Steps In APC Injection

This technique starts with the same four steps as ***Remote DLL injection:***

Step1: It opens a handle to the target process

Step2: Allocates memory in the target process

Step 3: Copies the malicious DLL pathname into the allocated memory

Step 4: determines the address of ***Loadlibrary()***

Step 5:

It opens a handle to the thread of the target process using the ***OpenThread()*** API. In the following example, The ***3rd*** argument, ***0xBEC(3052)***, is the thread ID (TID) of the ***iexplore.exe*** process. The return value of ***OpenThread()*** is the handle to the thread of ***iexplore.exe***:

```
         0128827E   57                  push edi
         0128827F   6A 00               push 0
         01288281   68 FF 03 1F 00      push 1F03FF
EIP ---> 01288286   FF 15 00 D4 4A 01   call dword ptr ds:[<&OpenThread>]
         0128828C   A3 68 09 55 01      mov dword ptr ds:[1550968],eax
         01288291   85 C0               test eax,eax

EAX  00000001
EBX  00000000

Default (stdcall)
1: [esp]   001F03FF
2: [esp+4] 00000000
3: [esp+8] 00000BEC  <--- Thread Id of iexplore.exe
```

Step 6:

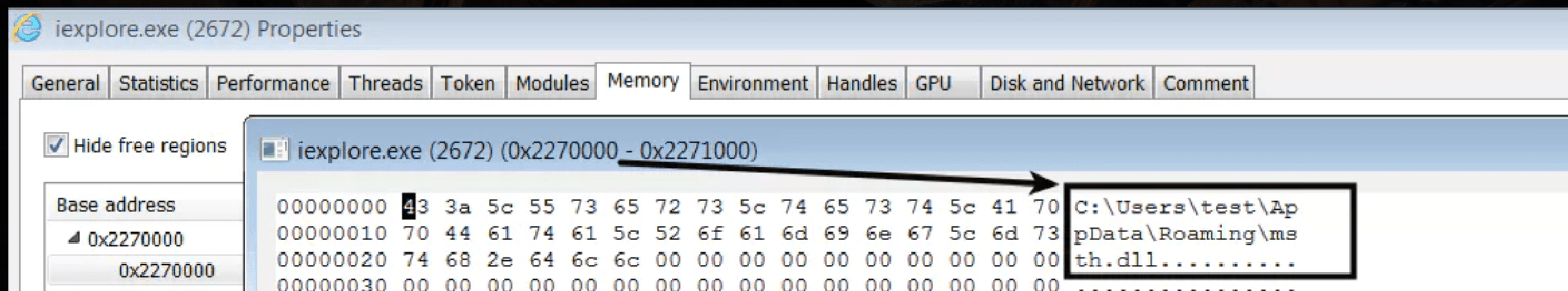
The malware calls ***QueueUserAPC()*** to queue the APC function. In the following screenshot, the ***1st argument*** is the pointer to the APC function that the malware wants the target thread to execute. In this case, the APC function is the ***LoadLibrary()*** whose address was determined previously. The ***2nd argument***, ***0x22c***, is the handle to the target thread of ***iexplore.exe***. The ***3rd argument***, ***0x2270000***, is the address in the target process (iexplore.exe) memory containing the full path to the malicious DLL

```
     01288A5F   FF 75 FC           push dword ptr ss:[ebp-4]
     01288A62   FF 75 E8           push dword ptr ss:[ebp-18]
     01288A65   FF 75 F4           push dword ptr ss:[ebp-C]
EIP  01288A68   FF 15 10 D4 4A 01  call dword ptr ds:[<&QueueUserAPC>]
     01288A6E   85 C0              test eax,eax
     01288A70   75 2C              jne rmdi.1288A9E
     01288A72   FF 15 88 D3 4A 01  call dword ptr ds:[<&GetLastError>]
```

```
EAX   00000015
EBX   00000000

Default (stdcall)
1: [esp] 7791DE15 <kernel32.LoadLibraryA>
2: [esp+4] 0000022C
3: [esp+8] 02270000
```

The following screenshot shows the content of the address 0x2270000 in Internet Explorer's process memory (this was passed as the ***3rd argument*** to ***QueueUserAPC()***); this address contains the full path to the DLL that was previously written by the malware:



At this point the injection is complete and when the thread of the target process enters alertable state, the thread executes ***LoadLibrary()*** from the ***APC queue*** by passing the full path to the DLL as the argument to ***LoadLibrary()***, as a result the malicious DLL gets loaded into target process address space which in turn invokes the ***DLLMain()*** function containing the malicious code

DLL Injection Using SetWindowsHookEx()

https://t.me/learningnets

DLL Injection Using SetWindowsHookEx()

- The ***SetWindowsHookEx()*** API can also be used to load a DLL into a target process address space and execute malicious code.

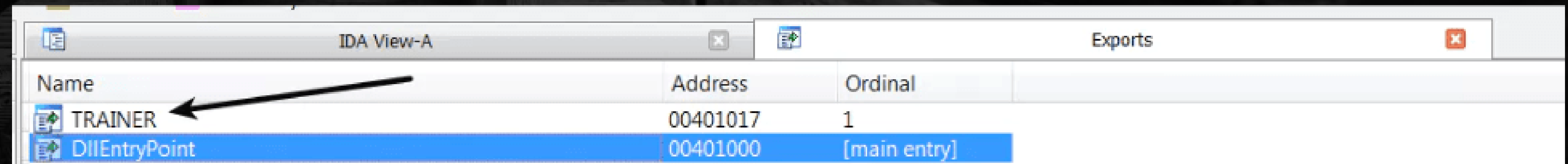
- A malware first loads the malicious DLL into its own address space. It then installs a hook procedure (a function exported by the malicious DLL) for a particular event (such as a keyboard or mouse event), and it associates the event with the thread of the target process (or all of the threads in the current desktop).

- When a particular event is triggered, for which the hook is installed, the thread of the target process will invoke the hook procedure. To invoke a hook procedure defined in the DLL, it must load the DLL (containing the hook procedure) into the address space of the target process.

Example

The following describes the steps performed by the malware sample (***Trojan Padador***) :

Step 1: The malware executable drops a DLL named ***tckdll.dll*** on the disk. The DLL contains an ***entrypoint*** function, and an export function named ***TRAINER***. The DLL entry point function does not do much, whereas the ***TRAINER*** function contains the malicious code.



Step 2:

Malware loads the DLL (***tckdll.dll***) into its own address space using the ***LoadLibrary()*** API, but no malicious code is executed at this point. The return value of ***LoadLibrary()*** is the handle to the loaded module (***tckdll.dll***). It then determines the address of the ***TRAINER*** function by using ***GetProcAddress()***:

```
00401047    push    offset LibFileName ; "tckdll.dll"
0040104C    call    LoadLibraryA
00401051    mov     hmod, eax
00401056    push    offset ProcName ; "TRAINER"
0040105B    push    eax             ; hModule
0040105C    call    GetProcAddress
```

Loads tckdll.dll into it's own address space

Determimes the address of TRAINER function

Step 3:

Malware uses the handle to tckdll.dll and the address of the TRAINER function to register a hook procedure for the keyboard event. The 1st argument, WH_KEYBOARD (constant value 2), specifies the type of event that will invoke the hook routine. The 2nd argument is the address of the hook routine, which is the address of the TRAINER function. The 3rd argument is the handle to the tckdll.dll, which contains the hook procedure. The fourth argument, 0, specifies that the hook procedure must be associated with all of the threads in the current desktop.

```
0040105C    call    GetProcAddress
00401061    push    0               ; dwThreadId
00401063    push    hmod            ; hmod
00401069    push    eax             ; lpfn
0040106A    push    WH_KEYBOARD     ; idHook
0040106C    call    SetWindowsHookExA
```

← Registers hook procedure for the keyboard event

After these steps, when the keyboard event is triggered within an application, that application will load the malicious DLL and invoke the TRAINER function.

DLL Injection Using The Application Compatibility Shim

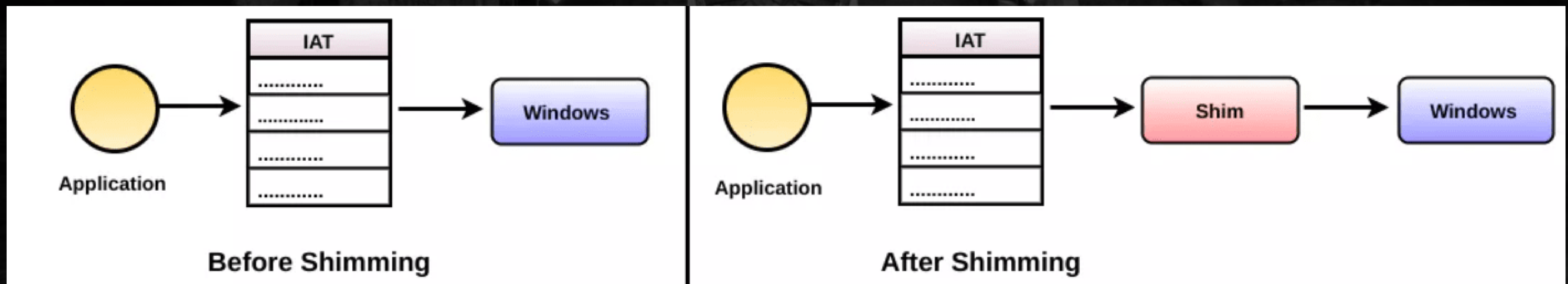
https://t.me/learningnets

DLL Injection Using The Application Compatibility Shim

- The Microsoft Windows application compatibility infrastructure/framework (application shim) is a feature that allows programs created for older versions of the operating system (such as Windows XP) to work with modern versions of the operating system (such as Windows 7 or Windows 10).

- The shims are provided by Microsoft to the developers so that they can apply fixes to their programs without rewriting the code. When a shim is applied to a program, and when the shimmed program is executed, the shim engine redirects the API call made by the shimmed program to shim code; this is done by replacing the pointer in the IAT with the address of the shim code.

The following screenshot should help you to understand the differences in interactions between the normal and shimmed applications in the Windows operating system:



Suppose that a few years back (before the release of Windows 7), you wrote an application (***xyz.exe***) which checked the OS version, before performing some useful operation. Let's suppose that your application determined the OS version by calling the ***GetVersion()*** API in ***kernel32.dll***. In short, the application did something useful only if the OS version was Windows XP. Now, if you take that application (***xyz.exe***) and run it on Windows 7, it will not do anything useful, because the OS version returned on Windows 7 by ***GetVersion()*** does not match with Windows XP. To make that application work on Windows 7, you can either fix the code and rebuild the program, or you can apply a shim called ***WinXPVersionLie*** to that application (***xyz.exe***).

Demo 12 - Injecting DLL Using Shim

https://t.me/learningnets

Hollow Process Injection (Process Hollowing)

Hollow Process Injection (Process Hollowing)

- Process hollowing, or Hollow Process Injection, is a code injection technique in which the executable section of the legitimate process in the memory, is replaced with a malicious executable.
- This technique allows an attacker to disguise his malware as a legitimate process and execute malicious code.
- The advantage of this technique is that the path of the process being hollowed out will still point to the legitimate path, and, by executing within the context of a legitimate process, the malware can bypass firewalls and host intrusion prevention systems.

Demo 13 - Hollow Process Injection