Import Address Table Hooking

By Nick Cano

This tutorial for IAT Hooking is intended for x86 processes. Due to slight differences in the PE format it will not work in x64, but can be easily converted.

Introduction:

Hooking is the practice of re-directing the flow of a program causing it to execute a code-cave or function in an injected module in place of another piece of code. In this tutorial, I will cover a method of hooking known as IAT *(Import Address Table)* Hooking. IAT Hooking is a 4 step process *(I assume you are able to manage to inject the attacking DLL on your own)*.

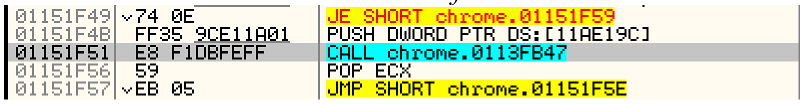
1. Read the PE *(Portable Executable)* Header from the target module
2. Traverse the Import Table of the target module and find the target function
3. Store the original function address so it can be called later
4. Overwrite the function address with the address of our trampoline function

Before Diving In...

Before we begin hooking functions using the IAT, we must first understand why it works. The IAT is a standard piece of a Windows executable, which resides within the PE Header. Every executable file which uses functions from dynamic libraries has an IAT which specifies which libraries it needs and what functions it uses from them.

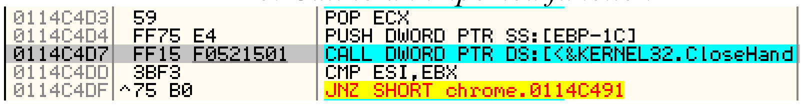
Typically, a function is called by a "CALL" operation followed by the distance to the functions location in memory *(Figure 1a)*. However, since DLL's aren't static in memory and the program knows nothing about their internals, it must "learn" the locations of their internal functions once it loads them. For this reason, functions exported from dynamic libraries are called by a "CALL" operation, followed by a memory read to a spot in the import table where the address for the function is stored *(Figure 1b)*.

1a. A normal function call



```
01151F49 |v74 0E          | JE SHORT chrome.01151F59
01151F4B | FF35 9CE11A01  | PUSH DWORD PTR DS:[11AE19C]
01151F51 | E8 F1DBFEFF    | CALL chrome.0113FB47
01151F56 | 59             | POP ECX
01151F57 |vEB 05          | JMP SHORT chrome.01151F5E
```

1b. Call to an imported function



```
0114C4D3 | 59             | POP ECX
0114C4D4 | FF75 E4        | PUSH DWORD PTR SS:[EBP-1C]
0114C4D7 | FF15 F0521501  | CALL DWORD PTR DS:[<&KERNEL32.CloseHand
0114C4DD | 3BF3           | CMP ESI,EBX
0114C4DF |^75 B0          | JNZ SHORT chrome.0114C491
```

The location which the call reads from is statically compiled into the executable - it is a reference to a specific entry in the import table. Originally blank, the memory at this location is overwritten with the address of the imported function once it is located in the DLL's EAT *(Export Address Table)*. Since this spot in memory is read to obtain the function address every time the function is called, we can overwrite it and trick the target executable into executing our own function.

Step 1: *Read the PE header from the target module*

The first step to hooking a function from the import table is to locate the import table. To locate the import table, we must first read the DOS Header, then read the Optional Header, and finally grab the import table from the Optional Header's data directory. The *GetModuleHandle()* WinAPI function allows us to get the start address of the processes main module and read the PE Header from that.

```
HMODULE module = GetModuleHandle(0);
IMAGE_IMPORT_DESCRIPTOR* importDescriptor = GetImportTable(module);

IMAGE_IMPORT_DESCRIPTOR* GetImportTable(HMODULE module)
{
        IMAGE_DOS_HEADER* dosHeader = (IMAGE_DOS_HEADER*)module;

        if (dosHeader->e_magic != 0x5A4D)
                return NULL;

        IMAGE_OPTIONAL_HEADER* optionalHeader =
                        (IMAGE_OPTIONAL_HEADER*)
                        ((BYTE*)module + dosHeader->e_lfanew + 24);

        if (optionalHeader->Magic != 0x10B)
                return NULL;

        if (optionalHeader->DataDirectory
                [IMAGE_DIRECTORY_ENTRY_IMPORT].Size == 0 ||
        optionalHeader->DataDirectory
                [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress == 0)
                return NULL;

        return (IMAGE_IMPORT_DESCRIPTOR*)
                ((BYTE*)module + optionalHeader->DataDirectory
                [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
}
```

Step 2: *Traverse the Import Table of the target module and find the target function*

Now that we have read the PE header and found the import table, we must loop through it until we can locate the target function. To do this, we iterate through the “Thunk” data of the import table, reading the function names until we find a match.

```
while (importDescriptor->FirstThunk)
{
        int n = 0;
        IMAGE_THUNK_DATA* thunkData = (IMAGE_THUNK_DATA*)
                        ((BYTE*)module + importDescriptor->OriginalFirstThunk);
```

```
          while (thunkData->u1.Function)
          {
                    char* importFunctionName = (char*)((BYTE*)module +
                                        (DWORD)thunkData->u1.AddressOfData + 2);

                    if(strcmp(importFunctionName, functionName) == 0)
                    {
                              //Step3
                              //Step4
                    }

                    n++;
                    thunkData++;
          }
          importDescriptor++;
}
```

Step 3: *Store the original function address so it can be called later*

Once we have located our function, we must obtain the old address so we can access the real function if needed. To do this, we read a 4-byte value from the first thunk of the current Import Descriptor, offset by the index of the thunk containing the function data.

```
PDWORD lpAddr =
          (DWORD*)((BYTE*)module + importDescriptor->FirstThunk) + n;
DWORD original = *lpAddr;
```

Step 4: *Overwrite the function address with the address of our trampoline function*

The last step of IAT Hooking is to actually place the hook. To do this, we ensure the location of the function address is writable and we write our function's address to it. To make certain the memory is writable, we must utilize the *VirtualProtect()* WinAPI function. To hide any traces of our presence, we also restore the memory access to its original state afterward.

```
DWORD oldProtection;
VirtualProtect(lpAddr, sizeof(DWORD), PAGE_READWRITE, &oldProtection);
*(DWORD*)lpAddr = newFunctionAddress;
VirtualProtect(lpAddr, sizeof(DWORD), oldProtection, &oldProtection);
return original;
```

Conclusion:

That's it! We've successfully read the PE Header, located the import table and spoofed the addresses of our target function, tricking the target module into believing the function it needs is our trampoline function. You should be able to find the full project, along with further tutorials, on the same host as this tutorial.