

# Hex-Rays, GetProcAddress, and Malware Analysis

msreverseengineering.com/blog/2021/6/1/hex-rays-getprocaddress-and-malware-analysis

June 1, 2021



June 1, 2021 [Rolf Rolles](#)

This entry is about how to make the best use of IDA and Hex-Rays with regards to a common scenario in malware analysis, namely, dynamic lookup of APIs via `GetProcAddress` (and/or import resolution via hash). I have been tempted to write this blog entry several times; in fact, I uploaded the original code for this entry exactly one year ago today. The problem that the script solves is simple: given the name of an API function, retrieve the proper type signature from IDA's type libraries. This makes it easier for the analyst to apply the proper types to the decompilation, which massively aid in readability and presentability. No more manually looking up and copying/pasting API type definitions, or ignoring the problem due to its tedious solution; just get the information directly from the IDA SDK. [Here is a link to the script](#). EDIT LATER: I decided to write a small GUI for the functionality. [You can find the Hex-Rays GUI plugin here](#).

## Background

Hex-Rays v7.4 introduced special handling for `GetProcAddress`. We can see the difference -- several of them, actually -- in the following two screenshots. The first comes from Hex-Rays 7.1:

```
int v5; // [rsp+28h] [rbp-20h]
HANDLE hObject; // [rsp+30h] [rbp-18h]
FARPROC v7; // [rsp+38h] [rbp-10h]
DWORD dwProcessId; // [rsp+50h] [rbp+8h]
_DWORD *v9; // [rsp+58h] [rbp+10h]

v9 = a2;
dwProcessId = a1;
v5 = 0;
v2 = GetModuleHandleW(L"kernel32.dll");
v7 = GetProcAddress(v2, "IsWow64Process");
if ( !v7 )
    return 1;
hObject = OpenProcess(0x400u, 0, dwProcessId);
if ( !hObject )
    return GetLastError();
if ( ((unsigned int (__fastcall *)(HANDLE, int *))v7)(hObject, &v5) == 0 )
```

The second comes from Hex-Rays 7.6:

```

BOOL v5; // [rsp+28h] [rbp-20h] BYREF
BOOL v6; // [rsp+2Ch] [rbp-1Ch]
HANDLE hObject; // [rsp+30h] [rbp-18h]
BOOL (__stdcall *IsWow64Process)(HANDLE, PBOOL); // [rsp+38h] [rbp-10h]

v5 = 0;
ModuleHandleW = GetModuleHandleW(L"kernel32.dll");
IsWow64Process = (BOOL (__stdcall *)(HANDLE, PBOOL))GetProcAddress(ModuleHandleW, "IsWow64Process");
if ( !IsWow64Process )
    return 1;
hObject = OpenProcess(0x400u, 0, a1);
if ( !hObject )
    return GetLastError();
if ( IsWow64Process(hObject, &v5) )

```

Several new features are evident in the screenshots -- more aggressive variable mapping eliminating the first two lines, and automatic variable renaming changing the names of variables -- but the one this entry focuses on has to do with the type assigned to the return value of `GetProcAddress`. Hex-Rays v7.4+ draw upon IDA's type libraries to automatically resolve the name of the procedure to its proper function pointer type signature, and set the return type of `GetProcAddress` to that type.

This change is evident in the screenshots above: for 7.1, the variable is named `v7`, its type is the generic `FARPROC`, and the final line shows a nasty cast on the function call. For 7.6, the variable is named `IsWow64Process`, its type is `BOOL (__stdcall *)(HANDLE, PBOOL)` (the proper type signature for the `IsWow64Process` API), and the final line shows no cast. Beyond the casts, we can also see that applying the type signature also changes the types of other local variables: `v5` in the first has the generic type `int`, whereas `v5` has the proper type `BOOL` in the second.

These screenshots clearly demonstrate that IDA is capable of resolving an API name to its proper type signature, the desirable effects of applying the proper type signature on readability, and the secondary effects of setting the types of other variables involved in calling those APIs.

## Relevance to Malware Analysis

---

Hex-Rays' built-in functionality won't work directly when malware looks up API names by hash, or uses encrypted strings for the API names: the decompiler must see a fixed string being passed to `GetProcAddress` to do its magic. Although the malware analysis community seems very comfortable in dealing with imports via hash and encrypted strings, they seem less comfortable with applying proper type signatures to the resultant variables and structure members. Only one publication I'm aware of bothers to tackle this, and it relies upon manual effort to retrieve the type definitions and create `typedef`s for them. This is unfortunate, as applying said types dramatically cleans up the decompilation output, but this is understandable, as the manual effort involved is rather cumbersome.

As a result, most publications that encounter this problem feature screenshots like this one. Note all of the casts on the function pointer invocations, and the so-called "partial types" `_QWORD`, etc.:

```

896 LABEL_11:
897 v16 = 0i64;
898 v17 = (char *)evilCmdAddr_dup1 + *evilCmdAddr_dup2; // ptr to MZ header (start of PE file)
899 v18 = *((_DWORD *)evilCmdAddr_dup1 + 1);
900 v19 = &v17[*((signed int *)v17 + 15)]; // ptr to PE header
901 v20 = *((unsigned __int16 *)v19 + 10) + 24;
902 v21 = *((_DWORD *)v19 + 44) == 0;
903 v256 = &v19[v20]; // ptr to .text header (code segment)
904 if ( v21 && v19[22] & 1 )
905     v16 = *((_QWORD *)v19 + 6);
906 result = ((__int64 (__fastcall *) (__int64, _QWORD, signed __int64, signed __int64))VirtualAlloc_func3)(// kernel32_VirtualAlloc(
907     v16,
908     *((unsigned int *)v19 + 20), // size of 39000 bytes
909     0x3000i64, // MEM_COMMIT | MEM_RESERVE
910     0x40i64); // PAGE_EXECUTE_READWRITE
911 // => allocate memory for PE file. returns pointer to that memory
912 v23 = result;
913 if ( result )
914 {
915     v24 = v20 + 40 * *((unsigned __int16 *)v19 + 3);
916     ((void (__fastcall *) (unsigned __int64, char *, _QWORD))memcpy_func4)(result, v17, *((signed int *)v17 + 15));
917     // => copy MZ header into new allocated region
918     ((void (__fastcall *) (unsigned __int64, char *, _QWORD))memcpy_func4)(v23 + *((signed int *)v17 + 15), v19, v24);
919     // => copy PE header into allocated region after MZ header
920     v25 = *((unsigned __int16 *)v19 + 3);

```

(I chose not to link the analysis from which the above screenshot was lifted, because my goal here is positive assistance to the malware analysis community, and not to draw negative attention to anyone's work in particular. This pattern is extremely frequent throughout presentations of malware analysis; it is immaterial who authored the screenshot above, and I had other examples to choose from.)

## The Solution

I did not know how to resolve an API name to its type signature, so I simply reverse engineered how Hex-Rays implements the functionality mentioned at the top of this entry. The result is a function `PrintTypeSignature(apiName)` you can use in your scripts and day-to-day work that does what its name implies: retrieves and prints the type signature for an API specified by name.

The script includes a demo function `Demo()` that resolves a number of API type signatures and prints them to the console. It begins by declaring a list of strings:

```

def Demo():
    apiNames = [
        "RegCloseKey",
        "RegQueryValueExW",
        "RegCreateKeyExW",
        "RegSetValueExA",
        "RegSetValueExW",
        "RegQueryValueExA",

```

The output of the script is the type signatures, ready to be copied and pasted into the variable type window and/or a structure declaration.

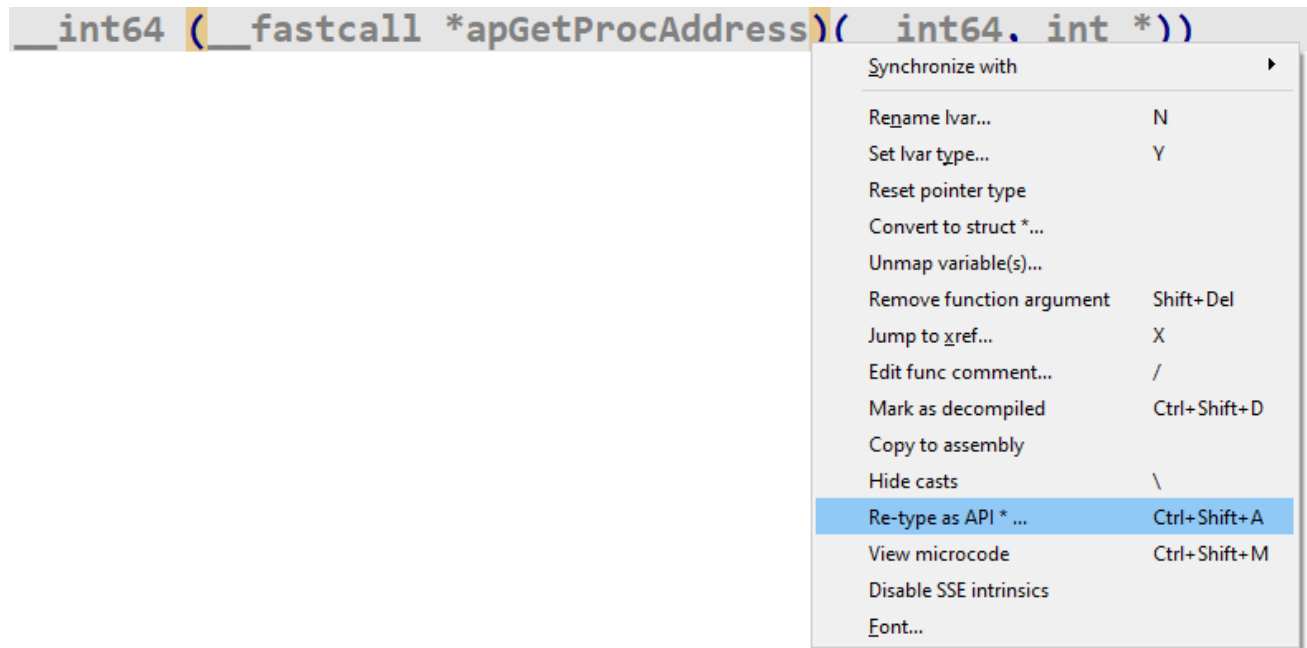
```

Python>Demo()
LSTATUS (__stdcall *RegCloseKey)(HKEY hKey);
LSTATUS (__stdcall *RegQueryValueEx)(HKEY hKey, LPCWSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData);
LSTATUS (__stdcall *RegCreateKeyEx)(HKEY hKey, LPCWSTR lpSubKey, DWORD Reserved, LPWSTR lpClass, DWORD dwOptions, REGSAM samDesired, const LPSECURITY_ATTRIBUTES lpSecurityAttributes, PHKEY phkResult, LPDWORD lpdwDisposition);
LSTATUS (__stdcall *RegSetValueEx)(HKEY hKey, LPCWSTR lpValueName, DWORD Reserved, DWORD dwType, const BYTE *lpData, DWORD cbData);
LSTATUS (__stdcall *RegSetValueExW)(HKEY hKey, LPCWSTR lpValueName, DWORD Reserved, DWORD dwType, const BYTE *lpData, DWORD cbData);
LSTATUS (__stdcall *RegQueryValueExA)(HKEY hKey, LPCSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData);

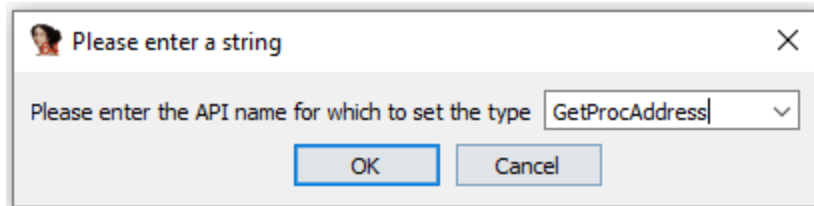
```

## GUI

I decided to add a small GUI to the functionality. After you run [this plugin](#), you will have a new entry on your Hex-Rays right-click menu that appears when your cursor is over a pointer-sized local variable, as follows:



Once you click on that, it will ask you to enter an API name:



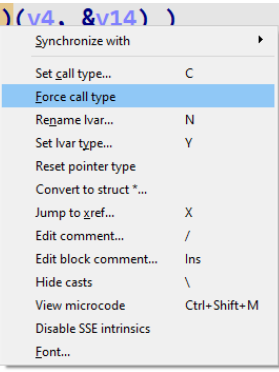
If there are no errors, the script will change the type of the variable to the function prototype for the API.

**FARPROC (\_\_stdcall \*apGetProcAddress)(HMODULE, LPCSTR)**

## A Final Note

Architecturally, there is a discrepancy between how the Hex-Rays microcode machinery handles type information for direct calls versus indirect ones. To summarize, you may still see casts in the output after applying the proper type signature to a variable or structure member. If this happens, right-click on the indirect call and press **Force call type** to force the proper type information to be applied at the call site. However, only do this once you have set the proper type information for the function pointer variable or structure member.

```
if ( ((unsigned int (__fastcall*)(void *, unsigned int*))GetAdaptersInfo)(v4, &v14) )
{
    v5 = 2;
}
else
{
    sub_18000EB60(v10, v14);
    v13 = 0;
    *(_OWORD *)Block = 0i64;
    v5 = 0;
    v12 = 0i64;
    sub_18000ECC0(Block, 8i64, &v13);
    v6 = Block[11];
}
```



Mostly I published this because I want to see more types applied, and fewer casts, in the malware analysis publications that I read. Please, use my script!

## Addendum

---

After publishing this article, I was made aware of [prior work](#) that is strongly related. That work focuses on the same problem, albeit in the disassembly listing rather than in Hex-Rays (and hence does not discuss some of the considerations from my entry above). Its ultimate solution is very similar to mine; it includes two out of three API calls from the one I came up with in this entry.