

Computer science

Final report: Software security

02.2017

**Memory Leak Detection**

**Student:** Moché Cohen

**Year:** 3 **semester:** 1

**Teacher: Arie Haenel**

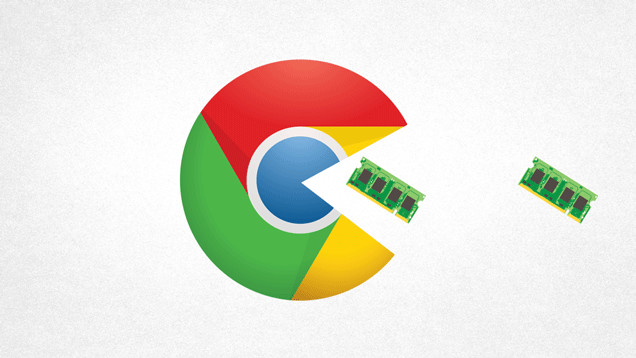


Table of Contents

[ABSTRACT 3](#_Toc474095883)

[How to use it? 3](#_Toc474095884)

[DLL INJECTION 5](#_Toc474095885)

[*CreateRemoteThread* 6](#_Toc474095886)

[API HOOKING 8](#_Toc474095887)

[*SSDT HOOKING* 8](#_Toc474095888)

[*INLINE HOOKING* 9](#_Toc474095889)

[IAT HOOKING 11](#_Toc474095890)

[*PE FILE* 12](#_Toc474095891)

[*HOOKING PART* 14](#_Toc474095892)

[***Example of output:*** 15](#_Toc474095893)

[Stack Frame 16](#_Toc474095894)

[Improvement: 18](#_Toc474095895)

[Conclusion: 18](#_Toc474095896)

# ABSTRACT

**Until today, memory leaks remains a problem in the world of software design. With increasingly complex programs, the detection of memory leaks is becoming more and more difficult. It doesn’t happen only to apprentices programmer: chrome, Firefox and also designers of drivers like AMD[[1]](#footnote-1) had stuck there.**

**The purpose of this project is therefore to detect the memory leaks of a program, without of course, having access to the source code or a debugger file. The detector must generate a log that informs about the memory leaks of the program.**

Memory Leak Detector

To detect memory leaks, the program must know all dynamic allocations as well as their release that have been made on the target program.

The aim of the project was to detect memory leaks in c programs, but it’s also work for c++. (dependent on the compiler, see on limits).

To do this, we will hook functions such as malloc or free and redirect them to our functions, which will record the different calls before returning the result of the real function. The technique used here is the IAT hooking with Dll injection.

So the program will contain a Dll with the hooked functions, an injector to inject the Dll into the target, and a target example program.

## How to use it?

To run it, simply drag and drop the target program in the injector or more specificly, with the command line: 

Injector FULL PATH

The injector will run the target process before inject.

You can also inject in a target that already run: 

Injector name of the target 1

Later I will give the [output](#_Example_of_output:).

Few words on memory leak:

A memory leak is an unintentional form of memory consumption whereby the developer fails to free an allocated block of memory when no longer needed.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing.

Memory leaks are therefore a daily issue for a developer, In fact, forgetting to free with free() is rather a beginner's mistake, so how companies as mentioned above could fall into this traps? Maybe if we understood the most common mistakes[[2]](#footnote-2) that were made, we wouldn’t need to use this detector ...

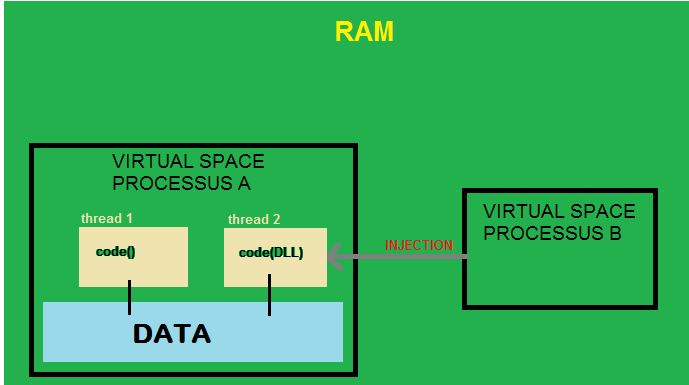
# DLL INJECTION

Everyone (or anyone who reads this) knows that RAM is shared between processes that are in execution.

And each process does not have access to the virtual space of the other, that is, even if a password is clearly written in the space of a process A, no other process can access it.

But here we want the target program to call our functions instead of its own, how can they call functions that are outside of its virtual space?

That is why we will directly inject these functions as a DLL[[3]](#footnote-3) into the process.



Here the process B is the injector.

They are many ways to inject Dll, fortunately, windows took care of created our tools.

First, this can be done manually with the registry HKEY\_LOCAL\_MACHINE \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ Windows \ AppInit\_DLLs.

It just happens that every process that loads USER32.dll loads also our dll, and as the majority of processes use USER32.dll we can be sure that our process will be injected without using an injector. But to do that, you have to have admin rights but also, the dll will inject into any process that uses User32.dll and that's not what it wants here.

So I'll talk about the [CreateRemoteThread](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx) I used in my injector.

## [*CreateRemoteThread*](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx)

As I said, windows made life easier; using this function, we can create a thread inside another process. And we don’t need to be an administrator!

This is the declaration of the function:

HANDLE WINAPI CreateThread(

Optional HANDLE HandleOfProcess,

\_In\_opt\_  LPSECURITY\_ATTRIBUTES  lpThreadAttributes,

\_In\_      SIZE\_T                 dwStackSize,

\_In\_      LPTHREAD\_START\_ROUTINE lpStartAddress,

\_In\_opt\_  LPVOID                 lpParameter,

\_In\_      DWORD                  dwCreationFlags,

\_Out\_opt\_ LPDWORD                lpThreadId

);

In each steps of the injection I will inform of each parameter of the function.

So, how the steps of the injections go:

1. First, we must have a pointer or handle of the target process with which we can access this process. Two options here, we are launching ourselves the process in the injector and in this case we will have the handle thanks to the [CreateProcess](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx) function, either we want to inject the Dll in a process already in execution, and then we will use [OpenProcess](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684320(v=vs.85).aspx) which will return a Handle.

The handle is the first parameter : HandleOfProcess.

What happened next is the same for both cases.

1. dwStackSize and lpThreadAttributes do not interest us and can keep their default settings 0 and null.

Now we have to deal with lpStartAddress, which is simply the address of the function we want to run as a thread.

This function is [LoadLibraryA()](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx) which loads a dll and takes in parameters a char \*. This function is in the kernell32.dll, and Dll are share between process so the address of this function is the always the same.

To get it : GetProcAddress(LoadLibrary("kernel32"), "LoadLibraryA");

1. So now lpParameter, there is a little trick here, because we couldn’t just put “MemoryHook.dll” in, why? Because this parameter is a LPVOID(a pointer), and what And what happens if the target process tries to use a data that is written in our process? Exactly! This is not possible because of the virtual space. To overcome this problem, we must write the path of the dll into the data memory of the target. How to do? Using [VirtualAllocEx()](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366890(v=vs.85).aspx) with for parameters the handle of the target and the length of the dllpath. We must keep the pointer that it return.

Now to write in the dllpath: [WriteProcessMemory()](https://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx): parameters: the handle, the path and the length.

1. So we have all our parameters, we just have to start CreateRemoteThread ()

which will launch a thread inside the target and and the "goal" of the thread is simply to load our dll.

hRemoteThread = CreateRemoteThread(

hProcess,

NULL,

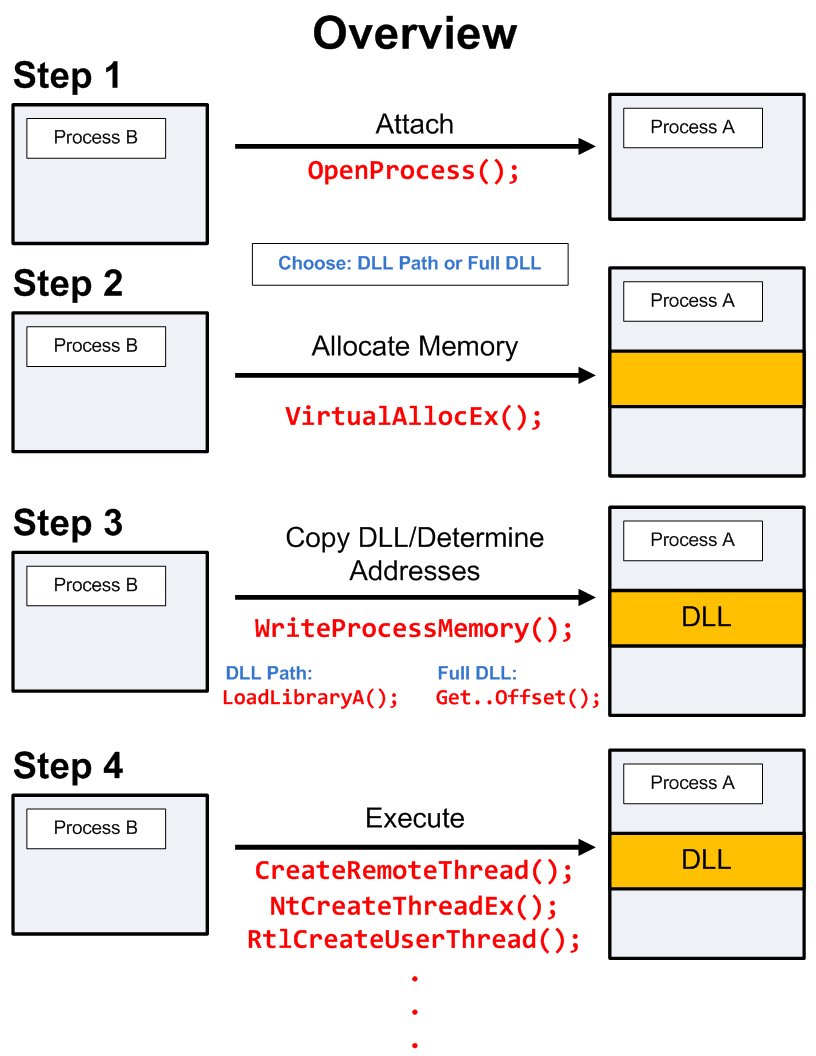
0,

(LPTHREAD\_START\_ROUTINE)pLoadLibrary, //LoadLibraryA()

hMemory, //MemoryHook.dll

0,

&lpThreadId);

[](http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html)

Finally, the dll is inside the target and already load, but dll have just functions, how the load lead to a change in the target?

Dll have also an entry point! Indeed, like a normal program with the main() function, dll have an DllMain() with three parameters: one of them is fdwReason, a DWORD which tells us why dllmain was launched.

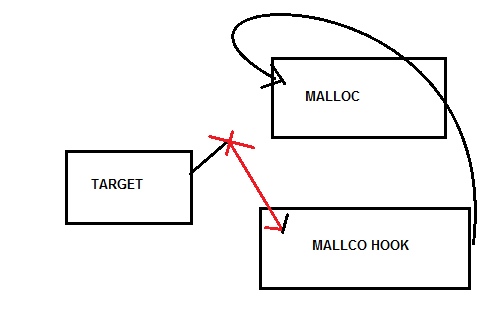
For example if the process load the library, it call the dllMain with

fdwReason = DLL\_PROCESS\_ATTACH, if it unload it will be DLL\_PROCESS\_DETACH, same for threads.

Which that, we can begin to “play” with the process. Make some Hook for example.

# API HOOKING

The detector must to record all the allocate and deallocate functions, a way to do this, is to redirect thus. How to redirect? “Place” a Hook.



There are several ways to hook (redirect) API functions of a program, but three of them stand out: SSDT hooking, inline Hooking and IAT hooking.

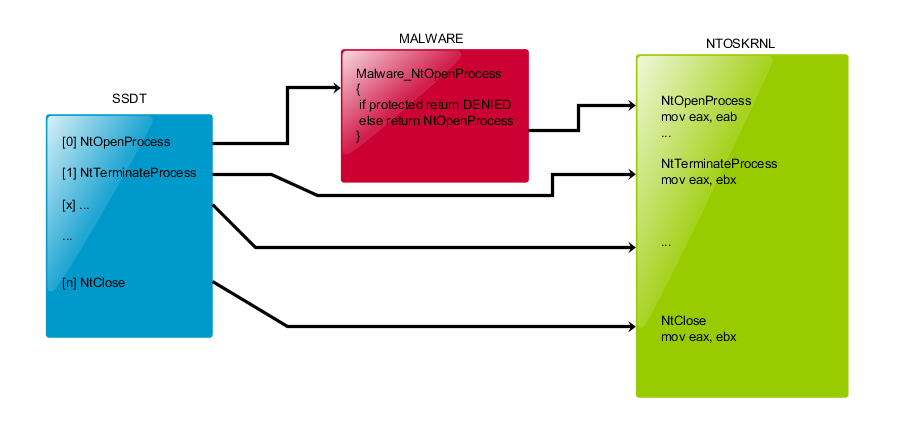
In the project I use the IAT hooking but I will try to explain why I didn’t want to use the other.

## *SSDT HOOKING*

The most powerful, I think, because it redirects the system call (kernel).

[The System Service Dispatch Table](http://en.wikipedia.org/wiki/System_Service_Dispatch_Table) is a table containing pointers to service functions (APIs) in ntoskrnl.exe (NtOpenProcess, NtOpenThread, … ). It is possible to modify the whole table, but in general we only modify the entries that interest us.

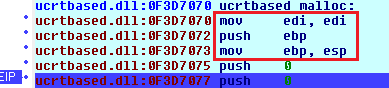
I didn’t use this Hook, first because I don’t need to modify the functions of the kernel, but only user-functions such as malloc ,also because it was a little more complicated to set up than the other two.

This kind of hook is usually used by rootkits and malwares, to have access to the kernel and be able to hide themselves. 

## *INLINE HOOKING*

The hooks are placed by directly modifying code within the target function (in the dll), usually by overwriting the first few bytes with a jump; this allows execution to be redirected before the function does any processing. Most hooking engines use a 32-bit relative jump (opcode 0xE9), which takes up 5 bytes of space.

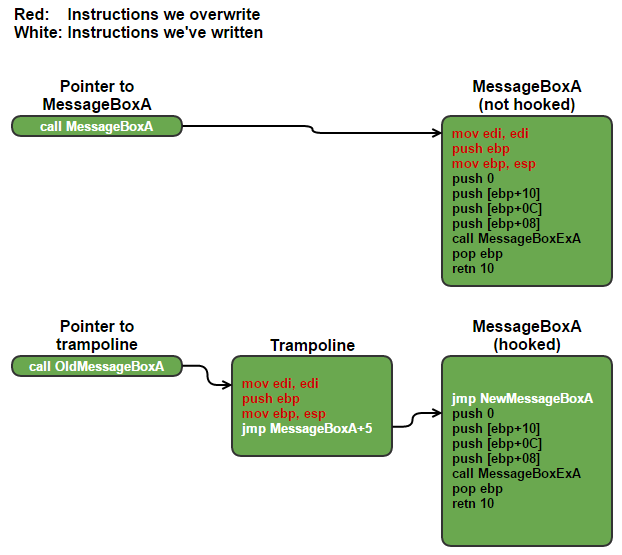
Of course, this jump function “jump” into our function.

When calling an API function there is usually a preamble to this function such as 

So the preamble of Windows API is exactly 5 bytes like for the unconditional jump! It’s seems that Microsoft want to help us. (Because if it was a instruction with for example 10 bytes when we change the 5 first byte with the jump, what comes after is an half instruction… (yes, we can also keep the whole instruction aside).)

So now, we have replace the preamble by the jump to our hook function, we can call the real function, but… the Dll are shared so when we call the real function in the modified inline dll, there is the jump function, which is redirect in our function whis is call… infinite recursive!

This is why, we must create a “trampoline” that keep the old preamble and “put” it when we call the real function.



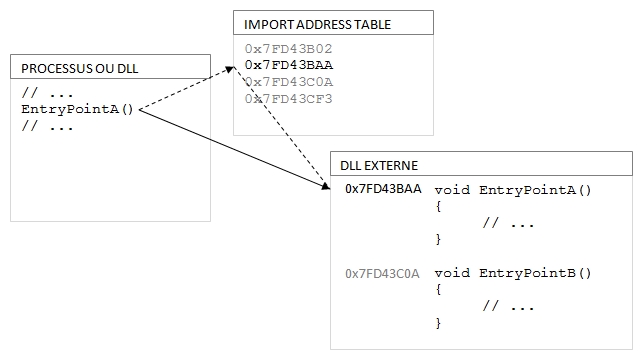
But I also decided not to use it naturally because you need to check the number of preamble bytes of each function with IDA each time, and there are also problems of race conditions. (If another thread uses the function while the trampoline is set it will not pass through our function).

So I decided to use the simplest but also the least powerful: IAT hooking.

# IAT HOOKING

We have seen how to modify the functions of the kernel, how also modify the functions in the dll what remains?

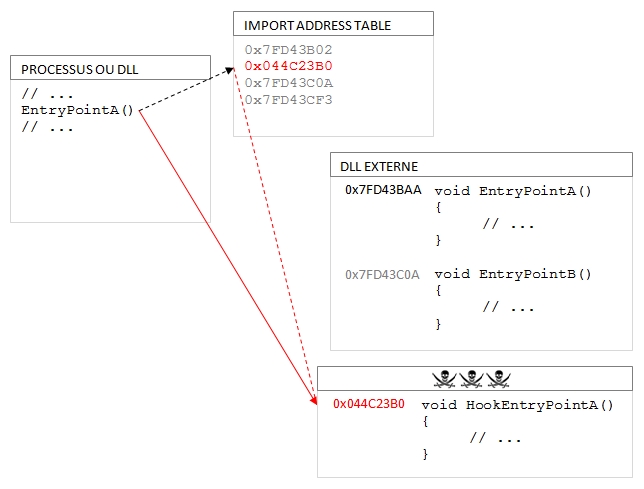
Each PE file or .exe has a header and in it are listed all imported functions used by the program and their address. This list of all functions is called “Import address table” (IAT) , the loader edit them on load time and give to each function the address to “jump”.



So when the program want to call EntryPointA(), it search in the IAT the address of the function and jump there. (see the image).

From there, hooking is a snap: we must just change the address in the IAT with the address of our function.

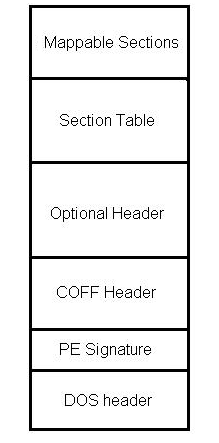
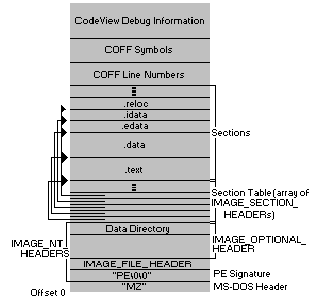
Without forgetting to keep the old address to call it in the hook functions.



So now, how to implement that: first we have to search the function we want to hook (here malloc for example).

To do so, we must learn what they are in a PE file:

## *PE FILE*



(DOS header is the first header, it is reversed)

In the DOS we have several information about the PE file, what we are interested in here is the e-> flanew which tells us the NT header offset.

In the NT header we have the IMAGE\_NT\_HEADER and the IMAGE\_OPTIONAL\_HEADER but don’t think that it’s really optional, It is inside that our IAT is found.

In this header there is a lot of information about the program like the address of the beginning of the code section or the number of bytes to reserve for the stack.

To find the IAT you have to take a look at the datadirectory where several IMAGE\_DIRECTORY\_ENTRY are listed, such as the one for resources, exports or IMPORTS, where the IAT is located. (The truth is that we already have an IMAGE\_DIRECTORY\_ENTRY\_IAT at this point but we can not know anything about it, I will explain below.)

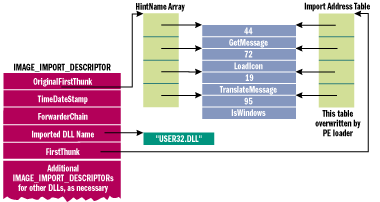
Finally, we take the virtual address of IMAGE\_DIRECTORY\_ENTRY\_IMPORT which is actually the import table. (or .idata in the first image).

Still there? It is soon finished: in this import table there is an IMAGE\_IMPORT\_DESCRIPTOR by DLL.

And what does each of these "images" contain? The IAT! That is to say a table of all the functions of this dll that are used in the program.

But then why did not we take the one that is already in the data directory? Because here we have an INT (import name table), and this is the only way to search the desired functions.

The IAT and the INT have both the same indexes: that is, if INT [1] = malloc then IAT [1] will be the address of malloc. So for the search it will be enough to increment the two table at the same time.[[4]](#footnote-4)



Finally! Now let's see how to implement this:

1. The process handle is converted to a DOS header.

PIMAGE\_DOS\_HEADER pPE = (PIMAGE\_DOS\_HEADER)hProcess;

1. NT headers are retrieved thanks to e\_flanew.

PIMAGE\_NT\_HEADERS pNTHeaders = (PIMAGE\_NT\_HEADERS)(pPE->e\_lfanew + (DWORD)pPE);

1. We retrieve the virtual address in the datadirectory (which leads us to the import table).

LPVOID pImageDirectory =

(LPVOID)pNTHeaders->OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_IMPORT].VirtualAddress;

1. To get the address of the table import of the first dll:

PIMAGE\_IMPORT\_DESCRIPTOR pImageImportDescriptor = (PIMAGE\_IMPORT\_DESCRIPTOR)((DWORD)pImageDirectory + (DWORD)pPE);

1. We do a loop that "lists" all DLLs imported by the program.

while (\*(LPDWORD)pImageImportDescriptor != 0)

1. Inside this loop, one traverses the INT array and retrieves the name of each function of INT; If the name is the name of the function sought, then the element with the same index of the IAT is the element containing the address of the procedure.

if (!strcmp((char\*)&pImageName->Name, FunctionName))

{

return &(pImportAddressTable->u1.Function);

}

So, in DllMain(), in the DLL\_PROCESS\_ATTACH section, we do a search for

functions for all our intended functions and then? We hook them.

## *HOOKING PART*

For this detector, I decided to hook the following functions: malloc, calloc, realloc free, \_free\_dbg.

Why \_free\_dbg? This is the function used by delete, and for new? new use malloc. (All this is based on my research with IDA, but it is not always true.[[5]](#footnote-5))

There are other functions that make it possible to make a dynamic allocation but this one are the most used.

So we have with the search in the IAT, all the addresses of each of these functions, all that remains is to change them.

Thanks to the DLL injection, our dll is already in the process, so we just need to change the pointer while keeping it aside for later use.

I'll take the example of malloc.

originalMalloc = (LPVOID)\*AdressIAT;//keep it aside

\*AdressIAT = (DWORD)&HookMalloc;//change the adress with the adress of my Function (THE HOOK IS HERE)

And in my Hookmalloc, I recall the old function, without forget to log the call.

LPVOID ptr = ((MallocFunc)originalMalloc)(size); // call the original malloc

if (ptr != NULL)

{

add\_mem\_info(ptr, size, file, line, func); // add this malloc to the list

}

return ptr;

I use a list of MEM\_INFO (That contain the address, size, func etc…) to record all the allocations.

The Hookfree function search the address in the list and delete the MEM\_INFO.

Let us conclude that the IAT Hook is probably the easiest for several reasons…

* No need to touch the assembly
* No need to take care of the multi-thread, so no mutex.[[6]](#footnote-6)

But it is also the least powerful…

* Easy to detect: checking the IAT. (but here this is no a malware so…)
* User-mode Hook (I can’t hook kernel functions like HeapAlloc or HeapFree)

At the end of the target, there will be the unload of my dll, so I use the DLL\_PROCESS\_ATTACH to write in the log file (leak\_info.txt) all the information about the memory leaks.

## ***Example of output:***

Memory Leak

-----------------------------------

address : 19557552

size : 5 bytes

file name: c:\temp\new\target\target\source.cpp

function: func

line : 8

-----------------------------------

Memory Leak

-----------------------------------

address : 19558056

size : 10 bytes

file name: c:\temp\new\target\target\source.cpp

function: main

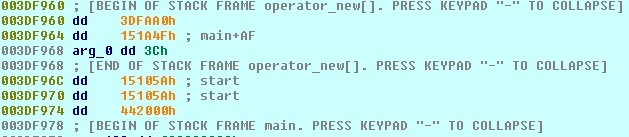
line : 28

-----------------------------------

Final Question: How to get the filename and the name of the function?

CaptureStackBackTrace !

# Stack Frame



In this Capture of IDA we see that when a function is called, all the data necessary for the execution of the function, as well as for the return to the initial state, are stacked.

The stack frame of a function is a memory area, in the stack, in which all the information needed to call this function is stored. There are also the local variables of the function.

What matters to us is that in this stack frame we also have the name of the calling function as well as the line from where it made the call.

Thanks to the function CaptureStackBackTrace we can have a capture of number of stack frame and use it to know who is the caller function and the line. (also the file name)

USHORT WINAPI CaptureStackBackTrace(

\_In\_      ULONG  FramesToSkip,

\_In\_      ULONG  FramesToCapture,

\_Out\_     PVOID  \*BackTrace,

\_Out\_opt\_ PULONG BackTraceHash

);

FramesToSkip: How many Frame we don’t need to see.

FramesToCapture: number of frame to capture

\*BackTrace: table to put in the stack frame

This function saves the stack frame in the "array" backtrace.

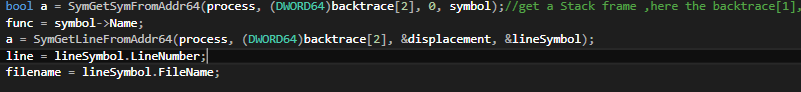
To use this table and have readable information there are two other functions to use: SymGetSymFromAddr64 and SymGetLineFromAddr64.

These two functions allow us to retrieve information like the symbol that contains the name of the function or the linesymbol that contains the line.

So, for example when the malloc function is call in the different stack frams we have:

1. The stack frame of our HookMalloc
2. The stack frame of the caller (the main)

So we retrieve the information of backtrace[1].



Here I use backtrace[2] because, before I call CaptureStackBacktrace, I maid an other call. (see the source code).

# Improvement:

* Make a GUI and show the allocation in real time.
* Garbage collection? This is easy to implement. (to all addresses that they are in the list I run the free function).
* Use kernel-mode hook like SSDT for HeapAlloc and HeapFree
* Try to have the line and name of the function even if the target was compiled with release mode.
* One of student say to me that it’s possible to free a part of the allocation, I didn’t know so also add some lines for this option
* There is an issue in realloc: I am obliged to use remove \_mem\_info () after the add\_mem\_info (), in fact, I erase the old address that was allocated after putting the new one. The problem is that realloc can keep the same address and so I will erase a memory that has not been freed.

# Conclusion:

The most interesting project I have had to do until today. (Out of three projects in total;))

When the mentor Arié Haenel gave all the subjects of the projects, the only ones that interested me were immediately taken.

So I had to fall back on a project, which I believed, not interesting.

But finally, even if he ate me days, this project is one of the things that most taught me during those years at the mahon.

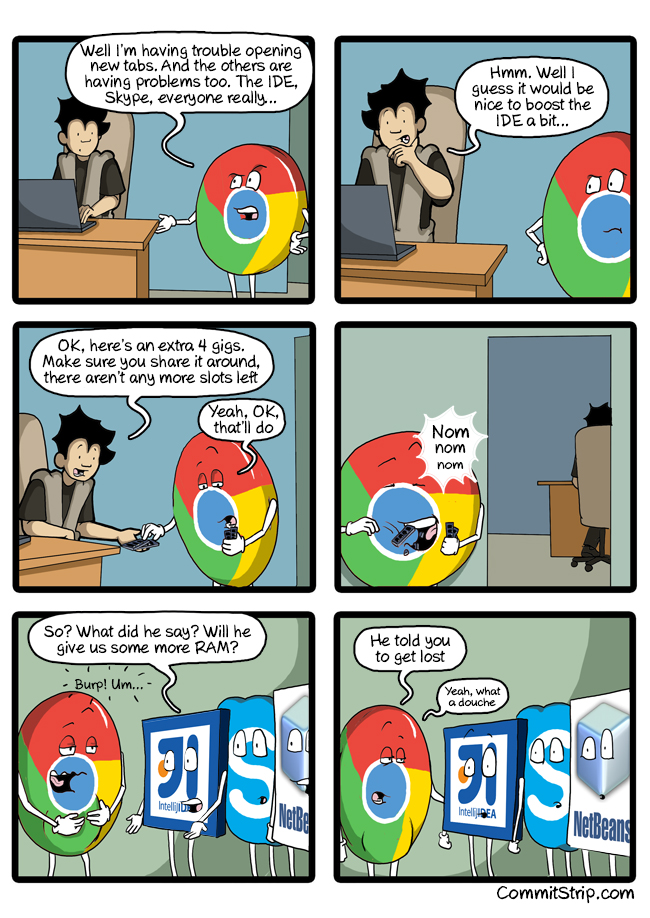
So if I have someone to thank it is obviously, Arié Haenel.

(A small thank you to moshe Uzan also for encouraging me to finish this project. ;)

That would be a crown if my project contained a memory leak, no?

Tools that I use for this project:

* [Dependency walker](http://www.dependencywalker.com/) (to see the dll and there functions)
* [IDA](https://www.hex-rays.com/products/ida/support/download.shtml)



1. <http://support.amd.com/en-us/kb-articles/Pages/AMD-Catalyst-15.9-Beta-Memory-Leak.aspx> [↑](#footnote-ref-1)
2. <https://www.toptal.com/c-plus-plus/top-10-common-c-plus-plus-developer-mistakes> [↑](#footnote-ref-2)
3. Inject code directly Is a little more complicated. [↑](#footnote-ref-3)
4. For more informations : <http://sandsprite.com/CodeStuff/pe_imptbl_headers.jpg>

   <https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files>

   <https://msdn.microsoft.com/en-us/library/ms809762.aspx> [↑](#footnote-ref-4)
5. For some compiler (in release mode) like cl.exe , there is issues, like malloc use HeapAllocate and I couldn’t hook this. [↑](#footnote-ref-5)
6. Indeed, the change of address in the IAT will remain all along the execution unlike the inline hooking which in its Hookfunctions cancels the change with the trampoline. [↑](#footnote-ref-6)