Detecting memory leaks in executable files

Segev Nevo Burstein, Yakir Demri

Jerusalem College of Technology
Department of Computer Science and Software Engineering
Fundamentals of Software Security – Arie Haenel

Abstract- There are many solutions on the internet that offer tools for detecting memory leaks and the idea itself has been known and researched for years. In this paper, we will present our explanation to some of the questions that arise in the detection process with an emphasis on solving the following questions: (1) What are memory leaks? (2) How do they happen? (3) How are they detected? (4) What tools did we use? And so on. Remember that the solution we bring does not replace the existing ones today but demonstrates capability.

Index Terms - Memory Leak Detection, DLL Injection, Hooking, Microsoft Detours, C Executables.

I. INTRODUCTION

When one writing code in C, and want to store information in a buffer, in run-time, the developer is given the ability to use the memory allocation mechanism and get a memory area where he can store the data. But memory allocations in run-time require to remember that he must release the memory at the end of the use to make room for future allocations. In systems that are required to provide high reliability over time, for example, servers, there should be optimal control over memory allocations and their release for the simple reason that improper tampering with memory can cause service disruption and even system crashes. Hence, programmers need to validate memory management during development – to make sure there are no memory corruptions of any kind. A mechanism such as those described in this paper gives the developer tools through which he can check the correctness of the software in run-time and if indeed, a problem occurs – obtain a detailed diagnosis on it. In the understanding journey, we will start by briefly review how automatic/manual memory allocation and deallocation are handled.

II. MEMORY ALLOCATION IN C

First, we must understand that there are several types of assignments.

```
static int x = 0;
```

Assigning **static variables** causes them to be stored in process memory, in the data segment (if initialized, else in the BSS). Automatic assignments (Local off course. Global variables, which need to be accessible anywhere in the program, also behave as static and stored in the same way) declared in the code are pushed on the stack.

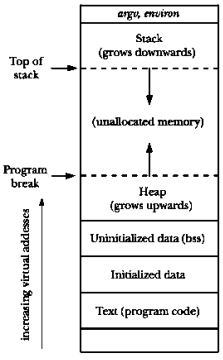
Of course, once the flow of the program exits the scope in which these automatic variables were set (In the next example the main scope), they are popped out of the stack. Thus, no memory leak can occur from the stack area nor in the memory of static assignments (these types are identical in their behavior because the allocation size must be clear at compilation time).

```
int main(){
    // allocated on stack
    int a;
    int b[10];
    const int n = 20;
    int c[n];
    return 0;
}
```

Therefore, it remains for us to understand what happens with - dynamic allocations. This kind of allocation is great whenever we need something between automatic and static, something that will not disappear at the end of the scope nor persist across the program lifetime. But, given that the compiler is unable to predict how much memory we will need while running, it also cannot manage the memory - making it so vulnerable. To use the dynamic allocation, an explicit call must be made to the **malloc** function. The job of this function is to access the memory area of the data, in particular, the area called the heap (provided by the system for every process) and return a reference to an area of sufficient size in memory (as requested).

```
int *array = malloc(10 * sizeof(int));
```

`void* malloc(size_t _Size)` takes a single argument which is the amount of memory to allocate in bytes and returns a void* to that specifically allocated space (In Windows, guaranteed to be suitably aligned for storage of any type of object that has as alignment requirement less than or equal to 8 bytes in x32 target platform or 16 bytes in x64 ones). Usually, the allocated block may be larger than the requested for alignment and metadata. In addition, to change the allocation at runtime, the `void* realloc(void* p, size_t Size)` function can be used. It attempts to resize the memory block pointed by p that was previously allocated with a call to malloc. It can be done either by expanding the existing space pointed by p or by allocating new memory block of size – Size, copying the memory area, and finally – freeing the old block. The last function is `void free(void* p)`, which deallocates the space previously allocated by malloc or realloc.



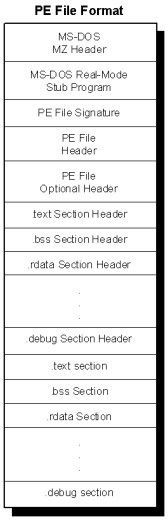
Now, given that these actions are the sole responsibility of the programmer, many errors can occur while running such as:

- Access freed memory.
- Forgetting to free allocated memory when no longer needed.
- Freeing memory that is not allocated dynamically.
- Non-allocating sufficient memory for an object.
- Calling free() on a block of memory twice.

Remember those blunders and we will address them later in the presentation of our solution. We will continue the journey of understanding the Detection with a glimpse into the world of-

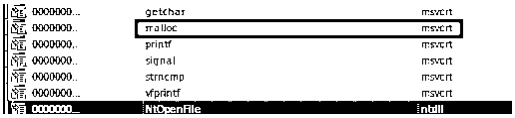
III. PORTABLE EXECUTABLE FORMAT

Our goal, during the seek-after solution, was to find a way that we could get some alert every time the user made a memory allocation/release so that we could track the numbers down. Of course, in case we had the code file itself, we could write a malloc(), free() override function and manipulate it as much as we would like. But our detection tool does not get a **.c** file, but **.exe**. So, this solution will not fit this scenario. Executable and Dynamic Link Libraries files under the Microsoft Windows operating system are in PE format, which, as the title says – "Portable Executable". that means that no matter what the architecture is, as long as it's under Windows system, it'll know how to run it. The format consists of:



Typical applications for windows usually predefined the nine sections (based on the functionality the application needs). For the sake of the required understanding, we will cover only the important part – The idata. as written in [3] – "This particular section represents the beginning of the list of import module and function names. If you begin examining the right section part of the data, you should recognize the names of familiar Win32 API functions and the module names they are found in". This table is called – **I**mport **A**ddress **T**able (or IAT).

Basically, what happens is that the Loader takes the PE file and uploads it to RAM as a process. It also takes care of filling in all the places in the table where the correct dynamic linking libraries addresses were missing in the corresponding addresses. So, let us examine the behavior of the import section in an exe file that contains some memory allocations:

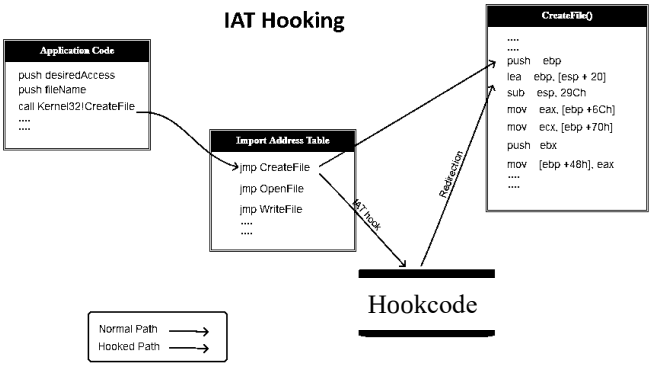


Surprisingly, as you can see, many functions have been imported to run the program. But, and here comes the important part, look at the marked part – this is the dynamic library which both implemented and exported all the functions listing there, and especially – **malloc**. This is the function that we aim to override!

Thus, the only information we have at hand at the moment is the knowledge that when, in run-time, the program reaches the place where the malloc function needs to be executed, it loads the code from there (msvcrt.dll) and execute it. At the end of the function, control is returned to our code to continue normal flow as usual. Okay, what can we do now?

IV. HOOKING

If you are beginning to understand where we are heading, then you may figure out that now we need some kind of magic that will give us such control that if a function we want (in our case memory allocation function, let's say malloc or free) is called – instead, control will be passed to our hands. Fortunately, such a capability does exist and is called – Hooking. By Wikipedia definition [4] – "used to alter or augment the behavior of an operating system, of applications, or other software components by **intercepting** function calls or messages or events passed between software components".



As this [5] picture shows, the application reached a line in the program where the CreateFile function needs to be executed. This function is not locally defined but imported from the kernel32.dll. However, somehow, we managed to change the normal flow of the program and succeeded to call our code. The way we do it is by creating a temporary dll that has two roles. The first one is to hold the logic of the alternative flow i.e. the hook body. The second job is, whenever our hook's logic is finished – call the REAL function, from the original dll, and let it perform the actions which the client code expected. (And if exists, also return any value back).

Of course, the sharp-eyed must have noticed that if the user passed variables to the original function, then our "overriding" function must also accept them to pass them to the original function. Basically, a consequence of this is that if we hook a function then the prototype of the function should be **exactly the same** as the one we hooked.

V. MICROSOFT DETOURS

So far, we have been able to understand that there is a tool that allows us to snatch the natural flow of running code and implant it inside a certain logic that we need. Of course, the idea we were looking for, to solve the memory leak detection problem, is to perform hooking on the three functions we described above and record the data received to an external file, which we can parse later and run and analyzer to get the necessary information.
But, of course, we must deal with the following questions:

(1) How to **inject our dll** which holds the hooked function to the original executable import address table?
(2) How to **change the IAT** itself so that when it is time to call the original function (which we want to hook) - It will read our hooked function from our dll?

Thankfully, the solution we found is extremely simple. Detours [6] – "is a software package for **re-routing** Win32 APIs underneath applications … Detours intercepts Win32 functions by **re-writing the in-memory code** for target functions … The Detours package also contains utilities to **attach arbitrary DLLs and data segments** (called payloads) to any Win32 binary." In a nutshell, exactly what we need.
To deploy Detours in our dll, we had to add this template into the DLL_PROCESS_ATTACH/DETACH sections into the dll:

```
BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    if (DetourIsHelperProcess()) {
        return TRUE;
    }

    if (dwReason == DLL_PROCESS_ATTACH) {
        DetourRestoreAfterWith();

        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)original_func, hooked_func);
        DetourTransactionCommit();
    }
    else if (dwReason == DLL_PROCESS_DETACH) {
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourDetach(&(PVOID&)original_func, hooked_func);
        DetourTransactionCommit();
    }
    return TRUE;
}
```

Let's understand a little the mechanism offered by Detours.
First of all, some preparations are required. The DetourTransactionBegin function is called to announce that a change in memory is about to take place and intervention is required on the part of the Detours mechanism. Then we call DetourUpdateThread and pass the current thread to it so that it can handle it. Now comes the most important step - we ask DetourAttach to set every call to original_func to be redirected, by adding JMP at the beginning of the function, to the hooked_func function – which is implemented inside the dll. We decided to implement our solution (that you are about to see) with a combination of these two approaches (IAT and Detours) although it is not really necessary, to learn more and expend our knowledge.

VI. OUR IMPLEMENTATION – HOOKING

Hence, we must access and use the function we found in [2]:

```
bool IAThooking(HMODULE hInstance)
{
    PIMAGE_IMPORT_DESCRIPTOR importedModule;
    PIMAGE_THUNK_DATA pFirstThunk, pOriginalFirstThunk;
    PIMAGE_IMPORT_BY_NAME pFuncData;

    importedModule = getImportTable(hInstance);
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    while (*(WORD*)importedModule != 0){
        pFirstThunk = (PIMAGE_THUNK_DATA)((PBYTE)hInstance + importedModule->FirstThunk);
        pOriginalFirstThunk = (PIMAGE_THUNK_DATA)((PBYTE)hInstance + importedModule->OriginalFirstThunk);
        pFuncData = (PIMAGE_IMPORT_BY_NAME)((PBYTE)hInstance + pOriginalFirstThunk->u1.AddressOfData);
        char* module_name = (char*)((PBYTE)hInstance + importedModule->Name);
        while (*(WORD*)pFirstThunk != 0 && *(WORD*)pOriginalFirstThunk != 0){
            if (std::string(pFuncData->Name).compare(TARGET_ALLOCATION) == 0){
                origin_alloc = (MemAlloc)DetourFindFunction(module_name, pFuncData->Name);
                DetourAttach(&(PVOID&)origin_alloc, new_alloc);
            }
            else if (std::string(pFuncData->Name).compare(TARGET_DEALLOCATION) == 0){
                origin_free = (MemFree)DetourFindFunction(module_name, pFuncData->Name);
                DetourAttach(&(PVOID&)origin_free, new_free);
            }
            else if (std::string(pFuncData->Name).compare(TARGET_REALLOCATION) == 0){
                origin_realloc = (MemRealloc)DetourFindFunction(module_name, pFuncData->Name);
                DetourAttach(&(PVOID&)origin_realloc, new_realloc);
            }
            pOriginalFirstThunk++;
            pFuncData = (PIMAGE_IMPORT_BY_NAME)((PBYTE)hInstance + pOriginalFirstThunk->u1.AddressOfData);
            pFirstThunk++;
        }
        importedModule++;
    }
    if (DetourTransactionCommit() == NO_ERROR)
        printf("detoured successfully\n");
    else
        printf("detoured un-successfully\n");

    return false;
}
```

What this function does is go through line by line in IAT and look for the names of the function on which we want to perform the hooking. Once it finds the name we are looking for, it performs DetourAttach, as we saw from the previous example, and changes the definition. But still, for us to change the memory as it is, from the executable file on which we want to perform the memory leak detection, this dll must run **from the context of the executable file**. The way we do this is with the help of the following trick. First of all, create an **additional** program that will receive the executable file. Using Detours, call the DetourCreateProcessWithDllEx. This function, as its name says, creates a process with the requested executable and **inject**s the additional libraries we ask him (In our case – "InjectedDLL.dll"). This trick is called – [7] DLL Injection.

```
HANDLE run_exe() {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    si.cb = sizeof(STARTUPINFO);
    char* DLLPath = (char*)R"(..\Debug\InjectedDLL.dll)";
    bool res = DetourCreateProcessWithDllEx(NULL, get_exe(),
        NULL, NULL, TRUE, NULL, NULL, NULL, &si, &pi, DLLPath, NULL);

    if (!res) {
        cout << "Failed to run executable";
        exit(0);
    }

    ResumeThread(pi.hThread);

    return pi.hProcess;
```

In this way, we were able to achieve full control over the calls to the functions we defined and all that is left is to manage the allocations/deallocations which were written into the diagnostic file during run-time by the injected dll (for analyzing purposes only).
Now, let's try to understand in more depth what the **DetourCreateProcessWithDllEx** function does behind the scenes and why it is called dll injection. The following function describes what we learned in class theoretically and introduces the subject in great detail. (This part is not related to our personal implementation, but it is very interesting to understand how things are done in practice):

```
bool Inject(wchar_t* command, const char* dllName)
{
	CreateProcess(NULL, command, NULL, NULL, FALSE,
			CREATE_SUSPENDED /* flags */, NULL, NULL, &si, &pi);

	LPVOID LoadLibAddr = (LPVOID)GetProcAddress(GetModuleHandleA("kernel32.dll"), "LoadLibraryA");
	LPVOID path_buffer = VirtualAllocEx(pi.hProcess, NULL, strlen(dllName),
			MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
	WriteProcessMemory(pi.hProcess, path_buffer, dllName, strlen(dllName), NULL);
	HANDLE thread_loader = CreateRemoteThread(pi.hProcess, NULL, NULL,
			(LPTHREAD_START_ROUTINE)LoadLibAddr, path_buffer, 0, NULL);
	WaitForSingleObject(thread_loader, INFINITE);
	ResumeThread(pi.hThread);
}
```

First, the si and pi parameters that hold, respectively, the startup information and process information are initialized and passed to the CreateProcess function. In addition, we provide an argument which is the executable file that we want to create as a process (=command). Also, and this is an important part, we demand at the time of creation of the process, that it be created as **suspended**. The purpose for this will be brought later. Once the process is created at our request, we performed the following steps:

(1) Find the address of the **LoadLibraryA** function inside **kernel32.dll** using **GetProcAddress**.
(2) Using **VirtualAllocEx**, we will be performing a memory allocation within the context of the suspended process with the size of the path of the dll (in our case – strlen(dllName) bytes) we want to load into the remote process we've just created.
(3) Using the **WriteProcessMemory** function we write into the memory area of the process remotely and insert into the buffer we created in (2), the dllName value.
(4) Now, in order to be able to run code in the context of the process **without running the process itself**, we create an internal thread and define it in such way that its entry point is the LoadLibraryA function we obtained and provide it as an argument which is the path_buffer that fills with the path of the dll We want to inject.
(5) Let our little thread execute until termination using the **WaitForSingleObject**.
(6) After thread_loader has finished loading our dll into the process, the PROCESS_ATTACH area is enabled within the dll so that all hooks settings are made. From now on, if we run the remote process, we will get all the capabilities we defined in the dll and they will run out of the context of the desired process.
(7) Let the remote process run by using **ResumeThead** (Refers to the main thread of the target process, from now on, he's not suspended anymore) and complete its entire execution, again, with **WaitForSingleObject** method (This is the program that we want to analyze and detect memory leakage if exists any).
(8) perform deallocation to path_buffer with **VirtualFreeEx** which exists within the memory area of the process. Also, free all memory in our handles (with **CloseHandle** function).

VII. Existing solutions

When we started researching existing tools for detecting memory leaks in executables. We have found that there are quite a few great solutions that many large organizations use during development and say that they provide very good analysis tools.

The first one is the famous [9][10] **Valgrind** which was originally designed to be a free memory debugging tool for **Linux** on x86 but has since evolved to become a generic framework for creating dynamic analysis tools such as checkers and profilers. The tool we examined from Valgrind's framework is the **memcheck** tool. Memcheck inserts extra instrumentation code around almost all instructions, which keeps track of the validity (Invalid = unallocated memory, unknown memory address points to an allocated, non-freed memory block, and so on…). Memcheck can detect:

- Use of uninitialized memory
- Reading/writing memory after it has been freed
- Reading/writing off the end of malloced blocks
- Memory leaks

Unfortunately, the problem with such tools is complexity. Typically, each time you run a test executable with memcheck, the run time complexity is doubled by about 20 to 30.

Another solution found is the **AddressSenitizer**. This is an open-source tool with a little more power, able to detects memory corruption bugs (Buffer Overflow, code section with computer security issues), accessing dangling pointer, and more. implemented in Clang, GCC, Xcode and MSVC. Like Valgrind (You can already understand that this is not a simple problem when performing code analysis. Especially in large scales) the instrumentation increases processing time by about 73% and memory usage by 240%.

VIII. Additional Features

- Our tool is capable of **supporting x64** based applications as well as x86. The main challenge in this feature involved the fact that the function CreateRemoteThread does not work as expected whenever the architecture of the parent process and the target process are different. To solve this problem, we compiled the main program both in x86 and x64 environment and created a simple primary service which determines the architecture of the target executable and runs the appropriate MemLeakDetector(x86/x64).exe. Of course, these executables also work with dll compiled in x86/x64 environments accordingly.
- In order to display the exact location where the memory leak occurred, we created in our code an implementation capable of returning the **call stack-trace** and returning the full trace until "main" function containing all sorts of information including the name and addresses of the functions, their origin file and line number and more. In case we cannot figure out the function name where the leak occurred (this behavior happens whenever the input executable was compiled in **Release** mode) we manage to inform the user of the module that the leak occurred.

IX. Limits

- **Compiler optimization problem**. For example, whenever developing in Visual Studio on Release mode (or any fully optimized configuration), the compiler applying all sorts of optimizations. This changes are affecting the expected flow execution of the program such as produce the most efficient and compact code by repositioning the assembly instruction. This example and other makes the output of the stack trace unpredictable and more.

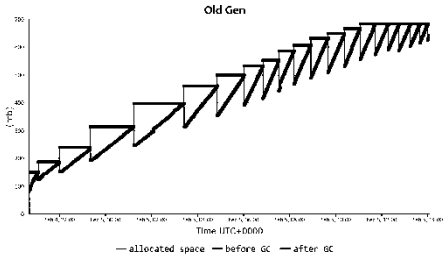
- **Performance issues**. The hooking system requires both dealing with the dynamic allocation/deallocation flow of the user's executable and writing the collected data into a diagnostic file, it makes the whole analyzing robust so execution time increases.
- **Memory footprints**. When handling with the hook system, a separate allocation is required. So we cannot avoid using dynamically allocated memory which basically multiplies the usage of heap memory.
- **Detection happens at run-time**. Unlike other systems, such as intellisense which is integrated in the IDE and capable performing statically, give an estimate according to the correctness of the syntax in compilation time. Our system is unable to do so and must run the executable file.

X. FUTURE

- **Support for C++ programs**. By hooking memory management operators of the C++ language i.e. new, new[], delete, delete[] our tool will be able to detect memory leaks that was written in C++.
- **advanced memory analysis**. There are nine main types of memory leaks [12]. One example is:
  "Losing reference to allocated memory block"

```
ptr1 = (char *) malloc (512);
ptr2 = (char *) malloc (512);
ptr2 = ptr1; /* causes the memory leak of ptr1 */
```

  Future development would be able to detect the type of each leakage.
- **Wider scan**. Currently, our tool is hooking a sub set of the allocation/deallocation function. A wider hooking system that hooks more functions such as HeapAlloc, HeapFree, calloc, etc. will be able to determine more accurately memory leaks and problems.
- Memory utilization presented with graphs like:



REFERENCES

[1] https://en.wikipedia.org/wiki/C_dynamic_memory_allocation
[2] https://www.digitalwhisper.co.il/files/Zines/0x12/DW18-3-IAT_Hooking.pdf
[3] https://blog.kowalczyk.info/articles/pefileformat.html
[4] https://en.wikipedia.org/wiki/Hooking
[5] https://nagareshwar.securityxploded.com/2014/03/20/code-injection-and-api-hooking-techniques/
[6] https://www.microsoft.com/en-us/research/project/detours/
[7] https://en.wikipedia.org/wiki/DLL_injection
[8] https://en.wikipedia.org/wiki/Memory_leak
[9] https://en.wikipedia.org/wiki/Valgrind
[10] https://valgrind.org/docs/manual/mc-manual.html
[11] https://en.wikipedia.org/wiki/AddressSanitizer
[12] https://blog.softwareverify.com/the-nine-types-of-memory-leak/

Appendix

[1] The dllmain.cpp inside the DLL

```
1  BOOL APIENTRY DllMain(HINSTANCE hInst, DWORD reason, LPVOID reserved)
2  {
3      if (DetourIsHelperProcess())
4          return true;
5      size_t s = 1;
6      char buf[256];
7      switch (reason)
8      {
9      case DLL_PROCESS_ATTACH:
10         DetourRestoreAfterWith();
11         symbol = (SYMBOL_INFO*)malloc(sizeof(SYMBOL_INFO) + 256 * sizeof(char));
12         line = (IMAGEHLP_LINE64*)malloc(sizeof(IMAGEHLP_LINE64));
13         getenv_s(&s, buf, "MEMCHECK_PATH");
14         file.open(buf, ofstream::trunc);
15
16         IAThooking(GetModuleHandleA(NULL));
17         file << "START\n";
18         break;
19     case DLL_PROCESS_DETACH:
20         file << "END\n";
21         DetourTransactionBegin();
22         DetourUpdateThread(GetCurrentThread());
23         if (origin_alloc)   DetourDetach(&(PVOID&)origin_alloc, new_alloc);
24         if (origin_free)    DetourDetach(&(PVOID&)origin_free, new_free);
25         if (origin_realloc) DetourDetach(&(PVOID&)origin_realloc, new_realloc);
26
27         if (DetourTransactionCommit() == NO_ERROR);
28         //  printf("undetoured successfully\n");
29         //else
30         //  printf("undetoured unsuccessfully\n");
31         file.close();
32         free(symbol);
33         free(line);
34         break;
35     case DLL_THREAD_ATTACH:
36         break;
37     case DLL_THREAD_DETACH:
38         break;
39     }
40
41     return TRUE;
42 }
43
```

Explain - To allow the user to choose where to save the diagnostic.txt file on his file tree we needed some path. Now, since we are "Inject" our dll into the executable file provided by the user and cannot pass it as input to the dll as input - we were required to pass the path of the file to which the user wants to write the file as an **environment variable**. This is what happens in line 13, the environment variable key is called MEMCHECK_PATH and the dll is reading its value using the getenv_s function. Also, when the IAThooking function finishes, a **START** label is pushed into the file (And the corresponding **END** label when the executable is terminated) to mark the USER code.

[2] In many cases, we have found that quite a few dynamic allocations of memory occur before and after the start of the executable. Because these are assignments made by the internal libraries of the language, one can trust that there was no memory leak there and there is no point in checking these areas. In this way, we mark the only part that our tool needs to test.

[2] The tool diagnostic file:

```
segev_burstein_608@LAPTOP-DU2A2BUN:/mnt/c/Users/User/Desktop$ cat diagnostic.txt
START
MALLOC  00F20A88        10
foo|002F1EA0|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\TestSamples\test.cpp|191|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\Debug\TestSamples.exe
REALLOC 00F20A88        100     00F8CF58
foo|002F1EA0|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\TestSamples\test.cpp|191|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\Debug\TestSamples.exe
FREE    00F8CF58
foo|002F1EA0|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\TestSamples\test.cpp|191|C:\Users\yakir\Projects\VisualStudioProjects\MemLeakDetector\Debug\TestSamples.exe
END
```

As you can see from the image below, this is the code that our tool worked on to produce the upper diagnostic file. First of all, variable x is getting 10 bytes from dynamic allocation (Second line in the output file, underneath the START), after that, a re-allocation is required in order to increase the current memory usage. From 10 bytes to 100. (Also written in the output file, line 4) and last, the deallocation with free which (line 6). So after that, basically, our main job was just to determine how to parse that file correctly.

```
#include <stdlib.h>
void foo() {
        void *x = malloc(10);
        x = realloc(x, 100);
        free(x);
}
int main() {
        foo();
}
```

[3] Printing full stack-trace from the current required location.

```
// Initialize all debug information from executable (PE)
SymInitialize(process, NULL, TRUE);
// "Pull" frames from the stack trace
frames = CaptureStackBackTrace(0, MAX_STACK_COUNT, stack, NULL);
char call[512];
for (unsigned int i = back; i < frames; ++i) {
    if (!SymFromAddr(process, (DWORD64)(stack[i]), 0, symbol))
        continue;

    GetModuleHandleEx(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS | GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT,
        (LPCTSTR)(symbol->Address), &hModule);
    if (hModule != NULL)
        GetModuleFileNameA(hModule, mod, BUFFER_SIZE);

    line_flag = SymGetLineFromAddr64(process, symbol->Address, &disp, line);
    sprintf_s(call, 512, "%s|%p|%s|%d|%s\n",
        symbol->Name,
        (void*)(symbol->Address),
        line_flag ? line->FileName : "NULL",
        line_flag ? line->LineNumber : 0,
        hModule != NULL ? mod : "NULL");
    file << call;
    if (strcmp(symbol->Name, "main") == 0)
        break;
}
```

The way this section works (in our built in function) is:

1. By using the **SymInitialize** function, we're able to initialize all the symbol handle which related to the user process. Symbol handler's information are used by debuggers to show much more detailed output while debugging. This data contains all the functions name which we'll need later.
2. Retrieve the stack status with **CaptureStackBackTrace**.
3. Iterate over the given frames array
    a. For each frame check that we can convert it to symbol form (if not, continue).
    b. From the module that the frame related to, locate it handle with **GetModuleHandleEx**.
    c. Using the handle to the module, we're able to retrieve:
        i. Function name (which we currently in its scope).
        ii. Address in the code section.
        iii. File name (the **.c** or **.cpp**)
        iv. Function's line of code.
        v. Module name (the executable file).
    d. Write to the file everything.