1st Ed

Windows Anti-Debugging & Anti-Anti-Debugging Techniques

Arash TC

Windows Anti-Debugging & Anti-Anti-Debugging Techniques

The world is a dangerous place.

Not because of those who do evil,

but because of those that stand by and do nothing.

Contents

Why?

As I got obsessed with documenting everything I learn in a way that if I lose my memory, I can learn them again from the ground up, I decided to prepare a write up on Anti-Debugging and Anti-Anti-Debugging Techniques performed on Windows operating system. This paper will not include pseudo code like lots of other blogs or paper you see (or at least I see.) but it will provide proof of concept code in C++ with hands-on examples. This paper will eventually turn into a book as I add new content because these techniques will never end and depending on your knowledge base and creativity, you can invent your own.

About the Author

It's me again, Arash TC, (the low-level security dude :D). One thing you definitely agree on about me (hope so) is that I don't spam the reader with unnecessary bullshit telling people about the history of assembly language or how my high school friend wrote a kickass virus with library's computer. You open the book, you jump into the actual content. Free of charge as usual because who am I to charge people for gaining knowledge. Remember to Learn and Contribute.

What about Linux?

The Linux part will be handled by my friend Kami[1] and the content will be uploaded (or maybe is uploaded by the time you're reading this) in the same github repository.[2]

Can I use the content in other ways?

Yes. Whatever creative commons lets you to. Specific license file is in the repository.[3]

How the book is organized?

We introduce anti-debugging techniques one after another. After explaining each technique, we explain how to bypass it. This section will be exclusive to reverse engineers dealing with these anti-debugging techniques and also for the curious.

---

[1] https://twitter.com/_k4m1_

[2] https://github.com/Captainarash/Reverse-Engineer/tree/master/Anti-Debugging%20and%20Anti-Anti-Debugging/Linux

[3] https://github.com/Captainarash/Reverse-Engineer/blob/master/LICENSE

Technique 1 – IsDebuggerPresent()

IsDebuggerPresent() is a function available in kernel32.dll, which checks a variable in the PEB structure of the current process. Now you may ask what is PEB? PEB is short for Process Environment Block which contains very useful information about a process. Each process has its own PEB structure loaded in memory. Here is a picture on the left I found on the internet to give you an idea of PEB. On the right, there is the specific variables (flags) defined in PEB found on MSDN[4].

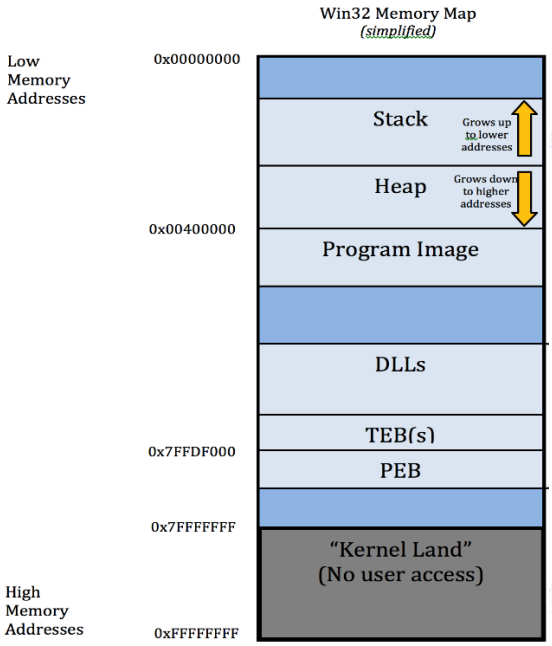

Figure 1- 1

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Figure 1- 2

As you can see, at offset 2 into PEB structure there is BeingDebugged flag. If the process is created inside a debugger, this flag will be set to 1. IsDebuggerPresent() function checks this flag and returns 1 if it's inside a debugger. Here's the code sample you can test inside and outside a debugger and see the results:

```
#include <cstdlib>

#include <Windows.h>

#include <stdio.h>

#include <iostream>


using namespace std;
```

[4] You can read more about PEB on MSDN https://msdn.microsoft.com/library/aa813706(VS.85).aspx

```
int main(int argc, char** argv) {
	if (IsDebuggerPresent())
	{
		printf("Go home kiddo!");
		return -1;
	}
	else {
		printf("Good Boy!");
		std::cin.ignore();
		return 0;
	}
}
```

Compiling and running the above code inside a debugger results in printing “Go Home Kiddo!” and exiting with error.

How to bypass

Let’s see what happens inside a debugger. We open the executable inside Olly and then search for all referenced text strings[5]. You can also find the IsDebuggerPresent() more efficiently by looking for its function call in all intermodular calls[6]. Anyways, after finding the address where the call to IsDebuggerPresent() happens, we set a breakpoint and let the program run. Look at figure 1-3 on the next page. Stepping into the call, we reach a point in kernel32.dll where it will actually check for BeingDebugged flag in PEB structure (Figure 1-4). PEB structure can be found at offset 0x30 into FS segment register on 32-bit versions of Windows and at offset 0x60 on 64-bit versions. After that, we can find BeingDebugged flag at offset 2 into the PEB structure.

[5] Right-click on the disassembler -> Search for -> All referenced text strings
[6] Right-click on the disassembler -> Search for -> All intermodular calls

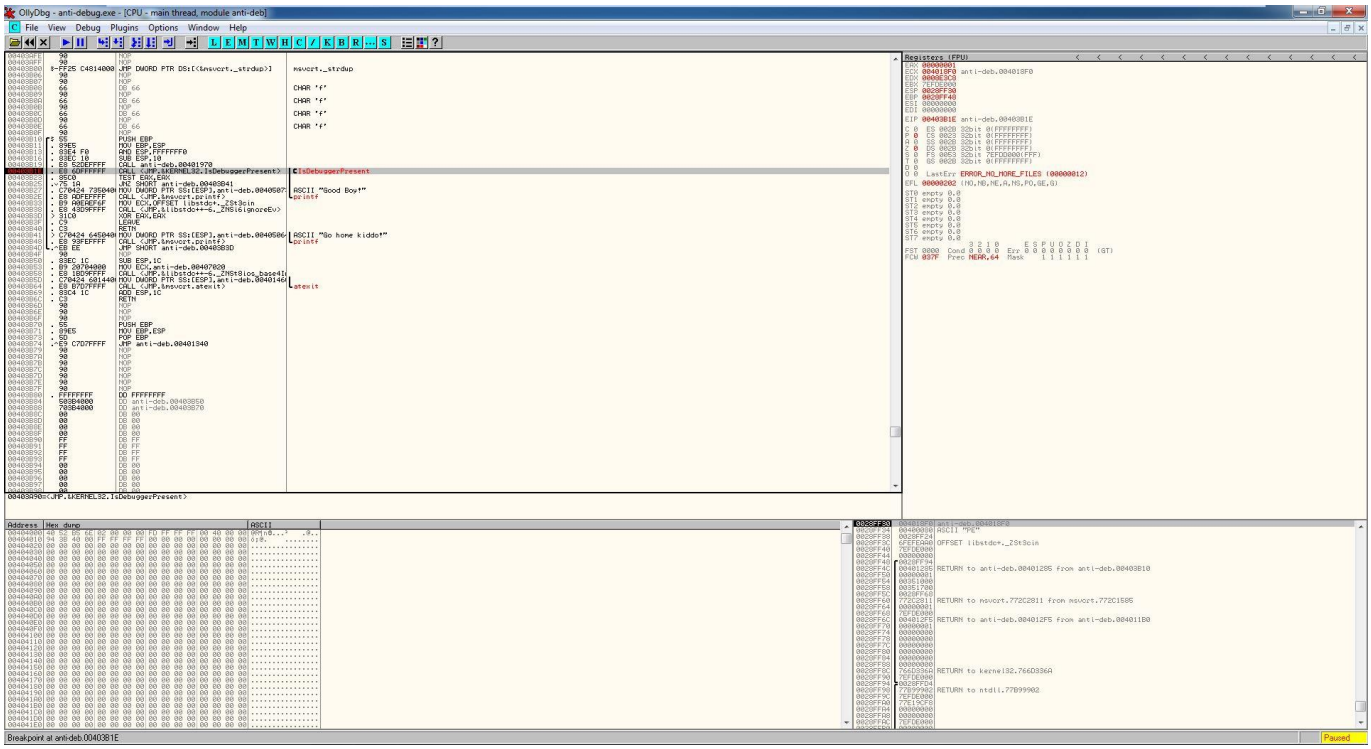

Figure 1- 3

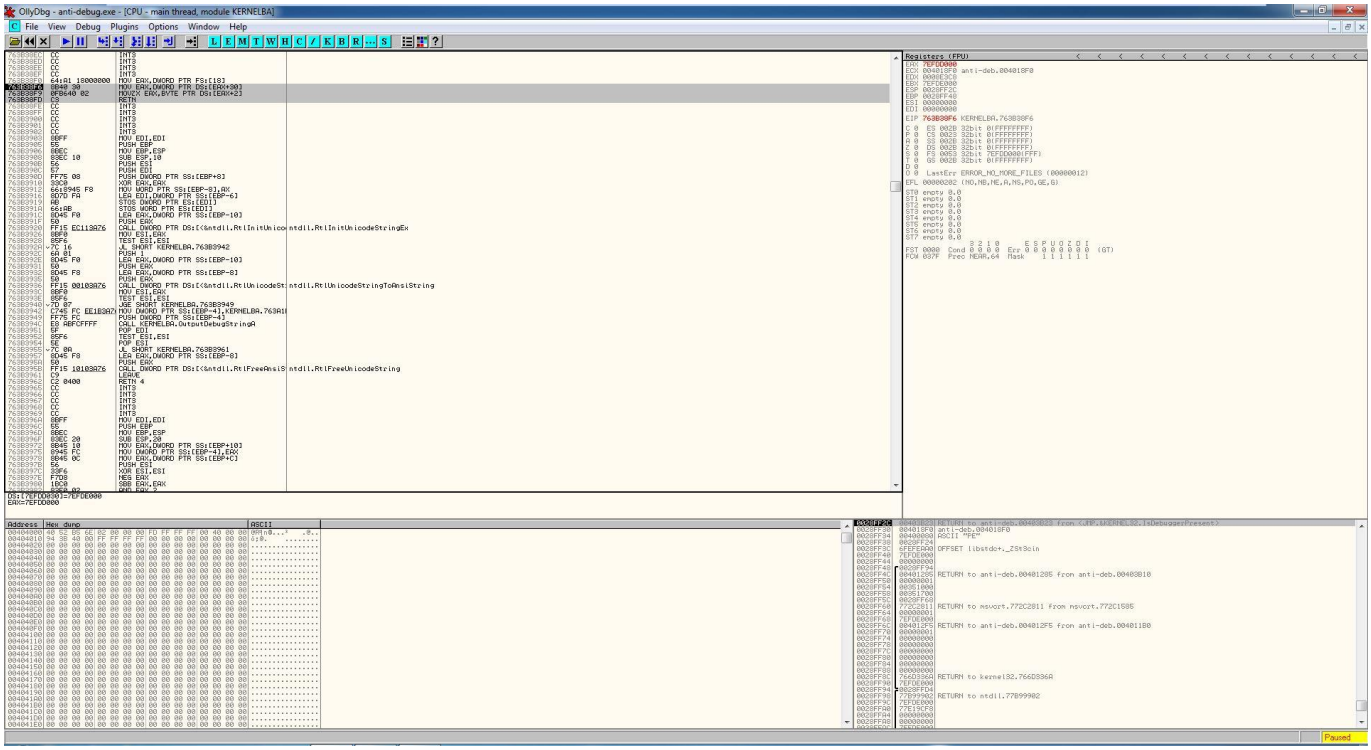

Figure 1- 4

After getting the intended value from PEB, we see that EAX is set to 1 (figure 1-5) which is the return value for IsDebuggerPresent(). By resetting EAX to 0, we can bypass this anti-debugging technique. There is also another way to bypass this technique in the beginning of the execution so we don't have to find the call every time.

Figure 1- 5

In the beginning of the execution we can go to the address fs:[30] (figure 1-6). Then we follow that address in dump and we can see at the offset 2 of that address, the value 1 is set. We are currently looking at PEB structure in memory dump view of Olly (figure 1-7) and we can set it to 0 so IsDebuggerPresent() will always return false (figure 1-8). This flag is an informative flag and does not affect the execution of the program.

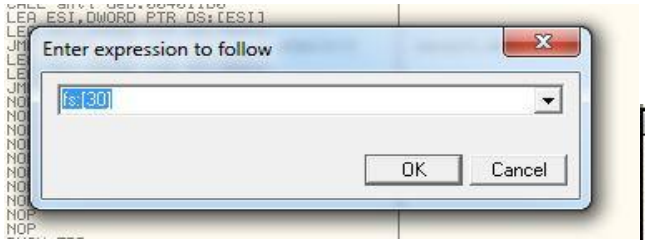

Figure 1- 6

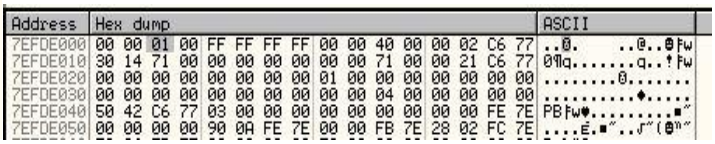

Figure 1- 7

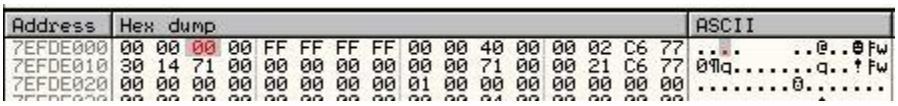

Figure 1- 8

Technique 2 – BeingDebgged in PEB

Yes! I know. This is the same as IsDebuggerPresent() but the difference is that this time, we check BeingDebugged flag in PEB directly with inline assembly code. One advantage of this regarding the previous technique is that there will be no intermodular calls and it's a bit easier to hide it in code. Here's the code sample for it:

```
#include <cstdlib>

#include <Windows.h>

#include <stdio.h>

#include <iostream>


using namespace std;


int main(int argc, char** argv) {
```

```
        // BeingDebugged method same as IsDebuggerPresent()
        asm("mov %fs:(0x30),%eax");
        asm("mov 2(%eax),%eax");
        asm("and $0x1,%eax");
        register int check asm("%eax");

        if(check==1){
                printf("Go home kid!");
                return -1;
        }
        else {
                printf("Good Boy!");
                std::cin.ignore();
                return 0;
        }
}
```

OK! Let's explain one line in the code which is different than what we saw earlier. After getting the value from BeingDebugged in PEB, we have "and $0x1, %eax" which will zero out all bit in EAX except the least significant bit which is the only bit we care about. After that we define a variable named check which will have the same value as EAX. The rest is self-explanatory.

How to bypass

The circumvention for this technique is the same as the first one, setting BeingDebugged flag in PEB to 0.

Technique 3 – NtGlobalFlag

Windows heap manager uses "Debug Heap" if the process in created inside a debugger. This happens in fact to make finding heap corruption and other heap related bugs easier to be investigated. When there is no

explicit value for NtGlobalFlag of the process in registry, the debugger will set 3 bits in NtGlobalFlag which can be used to find out if the process is being debugged or not. The three values are:

```
FLG_HEAP_ENABLE_TAIL_CHECK     0x10
FLG_HEAP_ENABLE_FREE_CHECK     0x20
FLG_HEAP_VALIDATE_PARAMETERS   0x40
```

These 3 bits make the value of NtGlobalFlag to be 0x70. So, we can use that in an anti-debugging technique to find out if our executable is being debugged. On 32-bit Windows, this value can be found at offset 0x68 and on 64-bit Windows, it is located at offset 0xBC into PEB. Right now, our C++ code is a 32-bit console application so we check offset 0x68.

```
#include <cstdlib>
#include <Windows.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

	//NtGlobalFlag method
	asm("mov %fs:(0x30),%eax");
	asm("mov 0x68(%eax),%eax");

	register int check asm("%eax");
	//Checking if eax is equal to 0x70 (112 decimal)
	if(check==112){
		printf("Go home kid!");
		return -1;
	}
```

```
        else {
                printf("Good Boy!");
                std::cin.ignore();
                return 0;
        }
}
```

The code above will check NtGlobalFlag value and will print “Go home kiddo!” and exit with error inside a debugger.

How to bypass

In Olly, we can go to the address fs:[30]+0x68 in the beginning of execution which is the address of NtGlobalFlag and change from 0x70 the value to 0x00. Here’s how:

Figure 3- 1

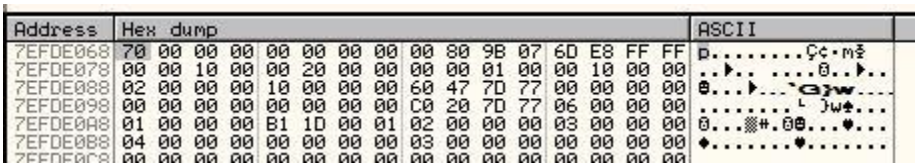

Figure 3- 2

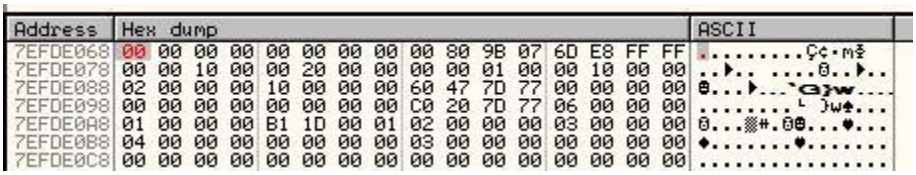

Figure 3- 3

In WinDbg we can start the process with setting all Heap Debug values set to 0 using this command:

Cmd> Windbg -hd example.exe

Technique 4 – Heap Flags

The heap has 2 flags which are affected if the process is created inside a debugger: FLAGS and FORCE_FLAGS. When the process is created normally, the value of FLAGS is set to 2 and the FORCE_FLAGS value is set to 0. If a process is created inside a debugger the value of FLAGS is the sum of these flags:

```
HEAP_GROWABLE                       0x2
HEAP_TAIL_CHECKING_ENABLED          0x20
HEAP_FREE_CHECKING_ENABLED          0x40
HEAP_VALIDATE_PARAMETERS_ENABLED    0x40000000
```

…which will be 0x40000062. The value of FORCE_FLAGS is the sum of these flags when inside a debugger:

```
HEAP_TAIL_CHECKING_ENABLED          0x20
HEAP_FREE_CHECKING_ENABLED          0x40
HEAP_VALIDATE_PARAMETERS_ENABLED    0x40000000
```

…which will be 0x40000060. Remember that these values are very dependent on the version of Windows. Also, the locations of these flags as mentioned before are different on 32-bit and 64-bit version. On Windows Vista and later on, FLAGS and FORCE_FLAGS can be found at offsets mentioned below:

Version	PEB	process_heap	FLAGS	FORCE_FLAGS
32-bit	FS:[0x30]	0x18 into PEB	0x40 into process_heap	0x44 into process_heap
64-bit	FS:[0x60]	0x30 into PEB	0x70 into process_heap	0x74 into process_heap

Here's a C++ snippet on 32-bit version to check FLAGS on Windows 7:

```
#include <cstdlib>
#include <Windows.h>
#include <stdio.h>
#include <iostream>

using namespace std;
```

```
int main(int argc, char** argv) {

   // HEAP Flags method - FLAGS
   asm("mov %fs:(0x30),%eax");
   asm("mov 0x18(%eax),%eax");
   asm("mov 0x40(%eax),%eax");



   register int check asm("%eax");


          //we can also check if check != 2
          if(check==0x40000062){
                    printf("Go home kid!");
                    return -1;
          }
          else {
                    printf("Good Boy!");
                    std::cin.ignore();
                    return 0;
          }
}
```

…and to check FORCE_FLAGS:

```
#include <cstdlib>
#include <Windows.h>
#include <stdio.h>
#include <iostream>

using namespace std;
```

```
int main(int argc, char** argv) {

   // HEAP Flags method – FORCE_FLAGS
   asm("mov %fs:(0x30),%eax");
   asm("mov 0x18(%eax),%eax");
   asm("mov 0x44(%eax),%eax");

   register int check asm("%eax");

         //we can also check if check != 0
         if(check==0x40000060){
                  printf("Go home kid!");
                  return -1;
         }
         else {
                  printf("Good Boy!");
                  std::cin.ignore();
                  return 0;
         }
}
```

How to bypass

To prevent ProcessHeap checks, you can:

- Manually change the ProcessHeap flag as shown in previous techniques.
- In OllyDbg, use a hide-debug plugin (e.g. HideOD[7]).
- In WinDbg, start the program with the debug heap disabled as mentioned before.

[7] https://www.aldeid.com/wiki/OllyDbg/HideOD

Technique 5 – NtQueryInformationProcess / ZwQueryInformationProcess

NtQueryInformationProcess is a function in Windows which as its name suggests, can get some information about the process in question. One of those valuable information is something called a Debug Port. We can check if a process is being debugged by asking the value of the Debug Port (if these is any debug port). If the returned value is -1 (or 0xFFFFFFFF), the process is being debugged.

OK! How can we put this to work? NtQueryInformationProcess is a function inside ntdll.dll. Looking at its definition on MSDN[8], we must load ntdll.dll, search for the address of NtQueryInformationProcess and then call it to check if debugger is present. Here's a sample code snippet written in Visual Studio[9]:

```
// antidbg.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <Windows.h>
#include <stdio.h>
#include <iostream>


typedef NTSTATUS(__stdcall *_NtQueryInformationProcess)(_In_ HANDLE, _In_  unsigned int, _Out_
PVOID, _In_ ULONG, _Out_ PULONG);

using namespace std;

int main(int argc, char** argv) {

	HANDLE hProcess = INVALID_HANDLE_VALUE;
	DWORD found = FALSE;
	DWORD ProcessDebugPort = 0x07;
	HMODULE hNTDLL = LoadLibraryW(L"ntdll.dll"); // defining a handle to ntdll.dll
	_NtQueryInformationProcess NtQueryInformationProcess = NULL;
	NtQueryInformationProcess = (_NtQueryInformationProcess)GetProcAddress(hNTDLL,
"NtQueryInformationProcess");
	hProcess = GetCurrentProcess();
	NTSTATUS status = NtQueryInformationProcess(hProcess, ProcessDebugPort, &found,
sizeof(DWORD), NULL);

	if (!status && found)
	{
		printf("Go home kiddo!");
	}
	else {
		printf("Good Boy!");
		std::cin.ignore();
		return 0;
	}
}
```

[8] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684280(v=vs.85).aspx
[9] Part of this code was taken from https://github.com/cetfor/AntiDBG

The code really doesn't need explanation. MSDN has already given a very good explanation. First, NtQueryInformationProcess has no import libraries and it's completely internal to the OS. That's why we need to locate it in ntdll.dll ourselves. Second, we can also ask for various information but we are only interested in debugger presence. So, we use the value 0x7 to check the debug port. Why this number? Well, this is part of an array inside ntdll.dll named PROCESSINFOCLASS[10] and NtQueryInformationProcess checks the number in that array. Number 0x7 is Debug Port.

Now let's run it inside a debugger and see what happens.

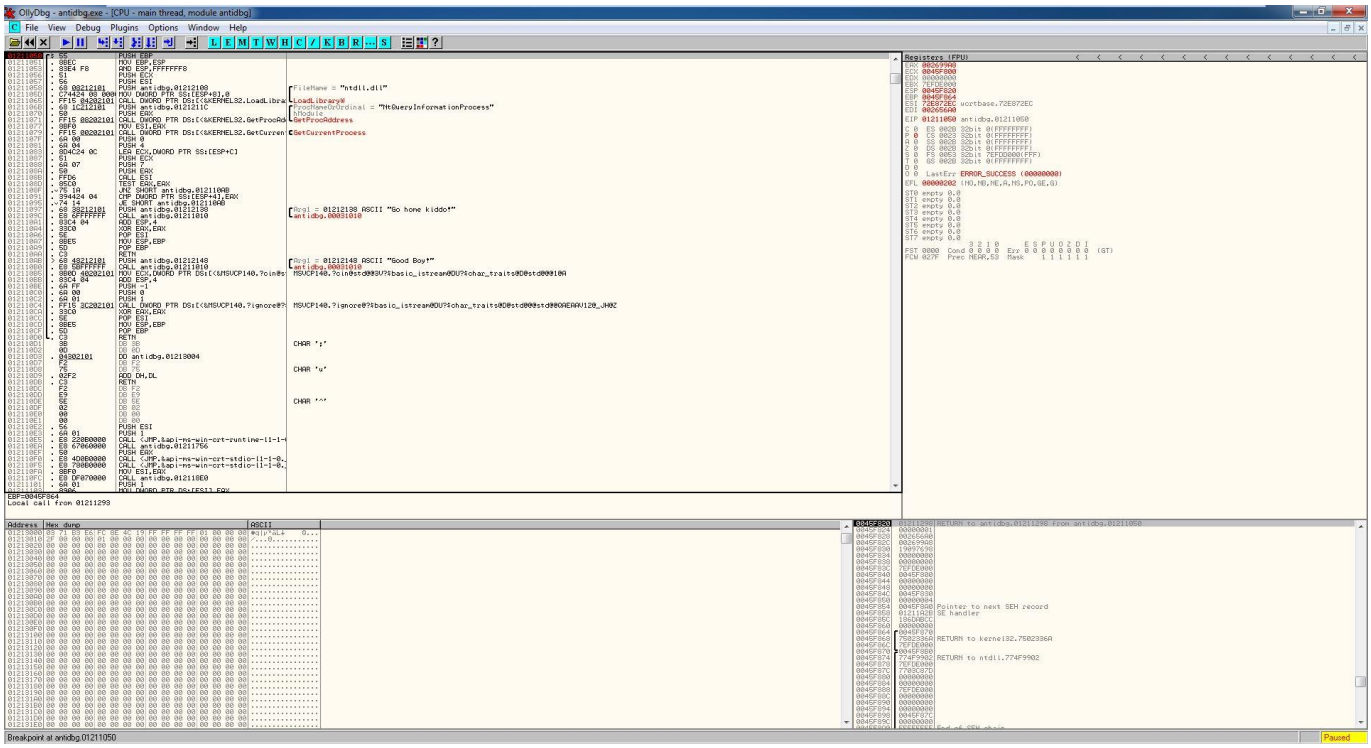

Figure 5- 1

As you can see, We first call LoadLibrayW() to load ntdll.dll. Then Using GetProcAddress() we find the address to NtQueryInformationProcess(). On figure 5-2, we can see setting up the arguments to call NtQueryInformationProcess(). Pay attention to the value of ESI; although we called NtQueryInformationProcess, we ended up getting ZwQueryInformationProcess. I couldn't find any difference between the 2. It got me curious enough to check their address in ntdll.dll and they both had the same address[11]. So, I assume they are the same.[12]

On Figure 5-3 we see 2 value checks. One for NTSTATUS which should be zero (meaning that the operation was successful[13]) and one for the return value of ZwQueryInformationProcess which if it's -1, the presence of the debugger is confirmed. We can see that the first test (NTSTATUS) will pass but the second won't since the value of [ESP+4] is 0xFFFFFFFF as shown in figure 5-4.

[10] http://www.pinvoke.net/default.aspx/ntdll.PROCESSINFOCLASS

[11] On Windows 7 x64, they were both at address 0x777EE740.

[12] Tell me if I'm wrong.

[13] https://msdn.microsoft.com/en-us/library/cc704588.aspx

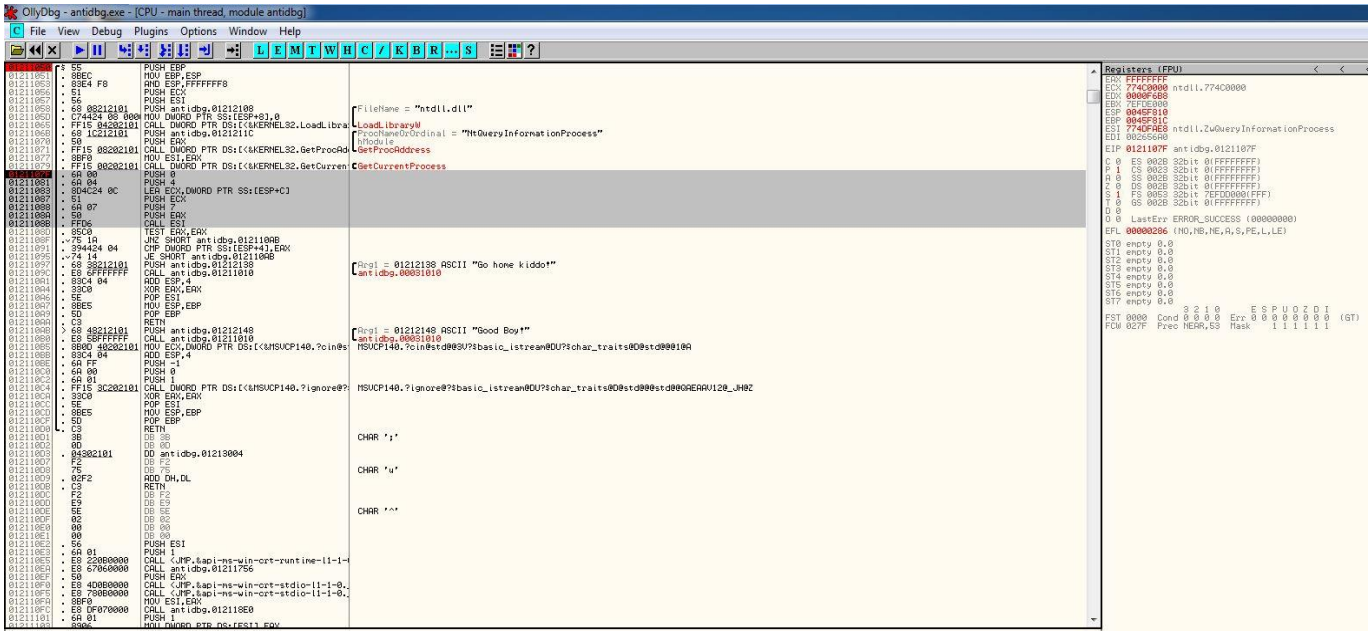

Figure 5- 2

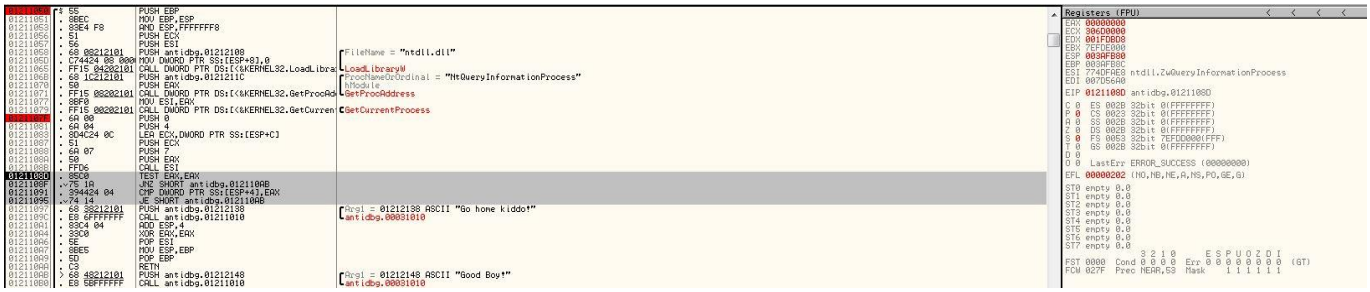

Figure 5- 3

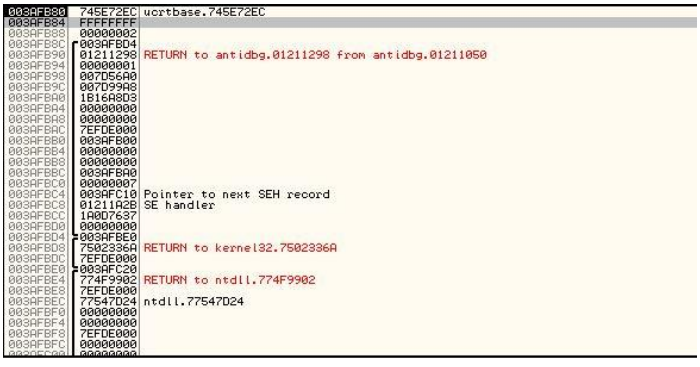

Figure 5- 4