# R4ndom's Tutorial #23: TLS Callbacks

by R4ndom on Sep.25, 2012, under Intermediate, Reverse Engineering, Tutorials

Unfortunately, our lives as reverse engineers is not always easy. If all it took to patch an app was a deleted resource or a quick patch, a lot more people would do it. Sometimes we must get a little 'low-level', wallow around in the operating system files, single-step an exception handler, or reverse engineer an unknown packer. To have a well-rounded skill set as a cracker, we must know a lot about a lot (or at least where to look about a lot) and it can get pretty technical.

This tutorial is about one of those technical areas: TLS callbacks. It is not easy, nor is it simple, but it can ruin an otherwise nice day of a reverse engineer that doesn't at least understand the basics of what they are, when they are used, and how to overcome them.

As in all tutorials on my site, the required files are included in the download of this tutorial on the tutorials page. We will be looking at three binaries, all included. We will also be using an Olly plugin called TLSCatch by Walliedassar, also included. Lastly, we will be using CFF Explorer, available on the tools page.

So get focused and let's tackle the subject of TLS Callbacks…

## Introduction

TLS stands for Thread Local Storage. As you probably know, threads are execution entities that run inside of a process. Programs make use of threads when they wish to accomplish multiples actions concurrently, even though sometimes 'concurrently' is just an illusion. For example, let's say you want to print a document. You press the 'print' button and the program formats the document and sends it to the printer. This activity would be run in a separate thread. The reason for this is we do not want to stop down the entire application until the document is done printing. We want it to start the print process and then immediately return to us, perhaps to do some work while it's printing.

If you have multiple processors, each thread can run on a separate processor. This can speed up applications as multiple processors can be doing work at the same time. Concurrency can also benefit from a single processor system. Take for example out print scenario above. Once the application sends the document off to the printer, the application will sit around, waiting for the printing activity to finish. This is A LOT of time, especially for a processor. During this waiting time, we can be doing other things. Threads allow a processor to split up activities, and while waiting for a response from one, can be working on another.

When these multiple threads are created, they usually share the same memory. For example, if we have an address book application and we decide to print a contact, the print thread will begin and have access to the main contact data. If, right after we start the print thread, we want to start another thread that begins showing the contact data on the screen (after all, the print dialog covered some of it), this new thread also has access to the contact data.

Threads access this pooled memory by calling the same addresses. In other words, thread A calls address 1000 to get the first contact, and thread B calls 1000 and gets the same data. The two addresses are the same. But what happens when we want a thread to have it's own data? Perhaps we want the printing thread to have a variable for if the printing was successful or not. All threads do not need to have this variable. Therefore, this thread needs a 'local' variable, one that only that specific thread has access to. This becomes really important when a single thread needs access to a large class or union. We do not want every thread started to have access to such a large chunk of memory.

Windows provides a way that a thread can have it's own 'local storage'. This storage is similar to a stack, but is only accessible to a specific thread. There is a certain chunk of memory that will be reserved for this thread, and variables can be stored in it. This way, only this one thread has access.

We can also set up the threads so that they all have a local copy of a variable, but they all access it through the same address. For example, we could have a count variable in every thread, and every thread accesses it through memory location 1000. But they are all different. Even though they are all the same address, Windows separates each thread's storage, so that location 1000 to thread A will not be the same variable (in memory) as thread B.

This TLS storage area can be used for other, often malicious, activities. Code can be put into this TLS section and can be run. The interesting thing about this is that the TLS code will run BEFORE the main entry point of the binary is run. When the Windows loader first loads the binary into memory, right after it loads in the DLLs needed, it checks a location in the PE header to see if there is a TLS section set up, and if there is, it looks for a callback address. If one is provided, this address is called, and the code in this section is run. After this runs, the loader then hands control over to the main application.

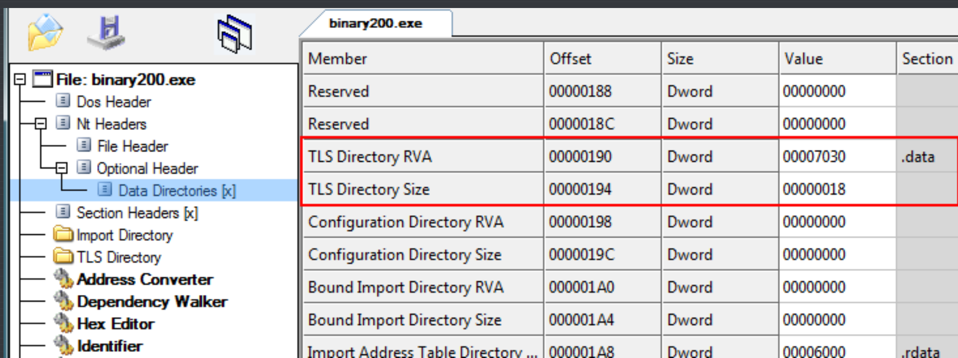
What all this boils down to is that when you load a binary into a debugger, often times we have the debugger set to stop at the module's main entry point. Once our debugger has stopped here, out TLS code has already been run. This code can do many things including checking for a debugger, infecting a system, or formatting a hard drive. And an unwary (or unskilled) reverse engineer will load this binary into Olly, and before you know it, your system is infected (or worse).

You may see this behavior empirically when you load a binary into Olly and the program immediately terminates, without ever touching any code in the actual main module. If this ever happens, your first thought should always be "check for a TLS section".

Now let's look at an actual example…

## Investigating the Binary

First load the binary200.exe into CFF Explorer. Clicking on Data Directories we can immediately see that there is a TLS section specified:



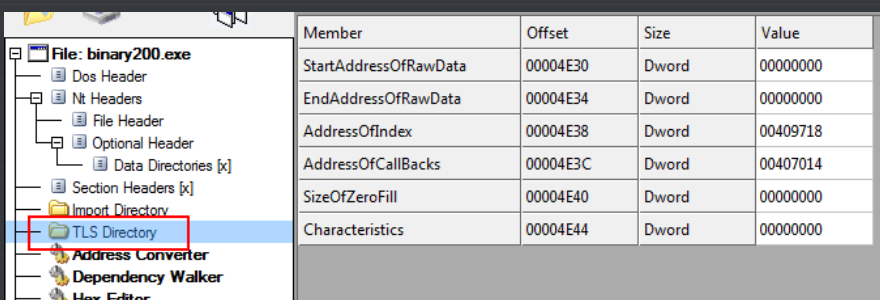

| Member | Offset | Size | Value | Section |
|---|---|---|---|---|
| Reserved | 00000188 | Dword | 00000000 | |
| Reserved | 0000018C | Dword | 00000000 | |
| TLS Directory RVA | 00000190 | Dword | 00007030 | .data |
| TLS Directory Size | 00000194 | Dword | 00000018 | |
| Configuration Directory RVA | 00000198 | Dword | 00000000 | |
| Configuration Directory Size | 0000019C | Dword | 00000000 | |
| Bound Import Directory RVA | 000001A0 | Dword | 00000000 | |
| Bound Import Directory Size | 000001A4 | Dword | 00000000 | |
| Import Address Table Directory ... | 000001A8 | Dword | 00006000 | .rdata |

*Note: Very few targets will ever have a TLS section specified unless they are using it as an anti-debug mechanism as most program never use TLS. The exception is Delphi programs which use them for internal reasons.*

There are two properties here. The first is TLS Directory RVA. This is a relative virtual address that points to the directory for the TLS. The directory contains various attributes of the TLS structure including its' starting and ending address and its' characteristics. Next is the TLS Directory Size, which in this case (and most cases) is 0×18 bytes.

Another thing you should notice is that the TLS itself is located in the .data section. This does not always have to be the case, and this will be important shortly.

Fortunately, CFF Explorer makes looking at the TLS directory very easy- simply click on the TLS Directory tab:




| Member | Offset | Size | Value |
|---|---|---|---|
| StartAddressOfRawData | 00004E30 | Dword | 00000000 |
| EndAddressOfRawData | 00004E34 | Dword | 00000000 |
| AddressOfIndex | 00004E38 | Dword | 00409718 |
| AddressOfCallBacks | 00004E3C | Dword | 00407014 |
| SizeOfZeroFill | 00004E40 | Dword | 00000000 |
| Characteristics | 00004E44 | Dword | 00000000 |

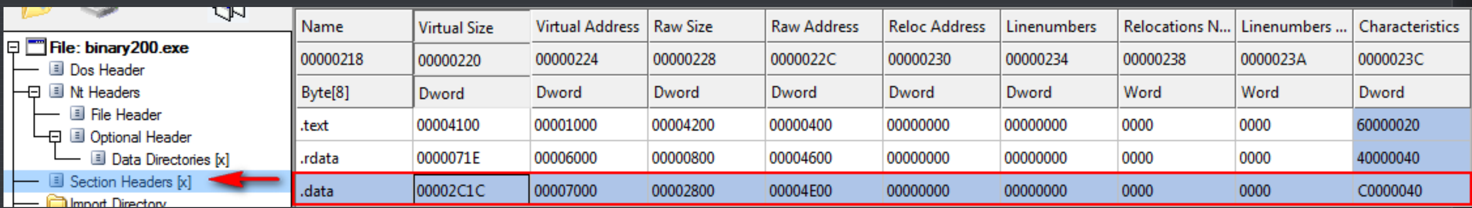
Let's go over these fields.

- StartAddressOfRawData: The address (offset) of the raw data on disk. Rarely used.
- EndAddressOfRawData: The end address on disk. Rarely used.
- AddressOfIndex: The slot in the TLS array that the TLS takes.
- AddressOfCallbacks: A pointer to an array of callback addresses.
- SizeOfZeroFill: Rarely used.
- Characteristics: Rarely used.

The only real field of value in this entity is the AddressOfCallbacks. This is a pointer to an array of callbacks. Because we can have more than one TLS callback code routine, this points to the first one in the list. There can be several callbacks, though, and the only way to see them all is in a hex dump. So that's where we'll go next…

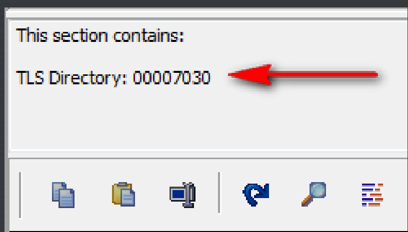
## The Dump

We saw earlier that the TLS directory structure is stored in the .data section, so let's bring that section up in CFF Explorer:




| Name | Virtual Size | Virtual Address | Raw Size | Raw Address | Reloc Address | Linenumbers | Relocations N... | Linenumbers ... | Characteristics |
|---|---|---|---|---|---|---|---|---|---|
| 00000218 | 00000220 | 00000224 | 00000228 | 0000022C | 00000230 | 00000234 | 00000238 | 0000023A | 0000023C |
| Byte[8] | Dword | Dword | Dword | Dword | Dword | Dword | Word | Word | Dword |
| .text | 00004100 | 00001000 | 00004200 | 00000400 | 00000000 | 00000000 | 0000 | 0000 | 60000020 |
| .rdata | 0000071E | 00006000 | 00000800 | 00004600 | 00000000 | 00000000 | 0000 | 0000 | 40000040 |
| .data | 00002C1C | 00007000 | 00002800 | 00004E00 | 00000000 | 00000000 | 0000 | 0000 | C0000040 |

As soon as you click on the .data section, CFF tells you that it contains TLS data and where the directory begins:

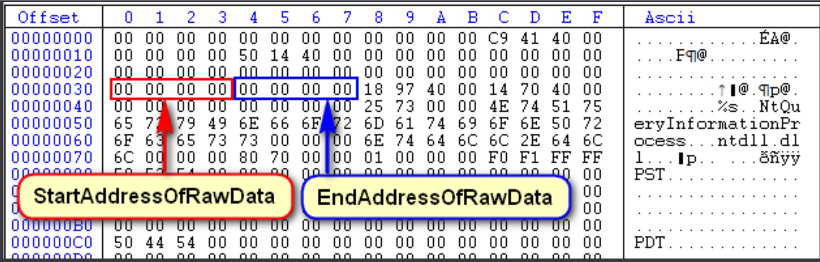



Though keep in mind that this is not the beginning of the TLS section, only the TLS directory. CFF will show a hex dump of the beginning of the .data section:

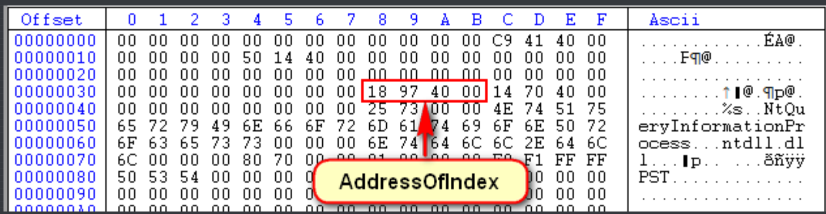
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | Ascii |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | C9 | 41 | 40 | 00 | ............ÉA@. |
| 00000010 | 00 | 00 | 00 | 00 | 50 | 14 | 40 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ....P¶@......... |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 18 | 97 | 40 | 00 | 14 | 70 | 40 | 00 | ........↑▮@.¶p@. |
| 00000040 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 25 | 73 | 00 | 00 | 4E | 74 | 51 | 75 | ........%s..NtQu |
| 00000050 | 65 | 72 | 79 | 49 | 6E | 66 | 6F | 72 | 6D | 61 | 74 | 69 | 6F | 6E | 50 | 72 | eryInformationPr |
| 00000060 | 6F | 63 | 65 | 73 | 73 | 00 | 00 | 00 | 6E | 74 | 64 | 6C | 6C | 2E | 64 | 6C | ocess...ntdll.dl |
| 00000070 | 6C | 00 | 00 | 00 | 80 | 70 | 00 | 00 | 01 | 00 | 00 | 00 | F0 | F1 | FF | FF | l...▮p.. ...ðñÿÿ |
| 00000080 | 50 | 53 | 54 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | PST............. |
| 00000090 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 000000A0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 000000B0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 000000C0 | 50 | 44 | 54 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | PDT............. |
| 000000D0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 000000E0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 000000F0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ................ |
| 00000100 | 80 | 70 | 40 | 00 | C0 | 70 | 40 | 00 | FF | FF | FF | FF | 00 | 00 | 00 | 00 | ▮p@.Àp@.ÿÿÿÿ.... |

We will take a closer look at this section, in order to understand what data is contained in this region.

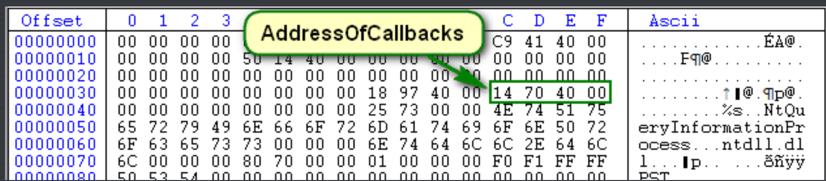
CFF Explorer has told us that the actual directory has started at offset 0×30 (or 0×7030 in the .data section, which is the same address). Following along with the various fields in the above picture of the TLS directory, at offset 30 is the StartAddressOfRawData and the EndAddressOfRawData:



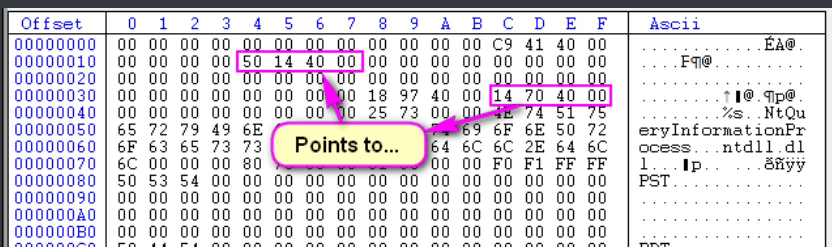

Next up is the AddressOfIndex, which we can see is 409781 (little endian):



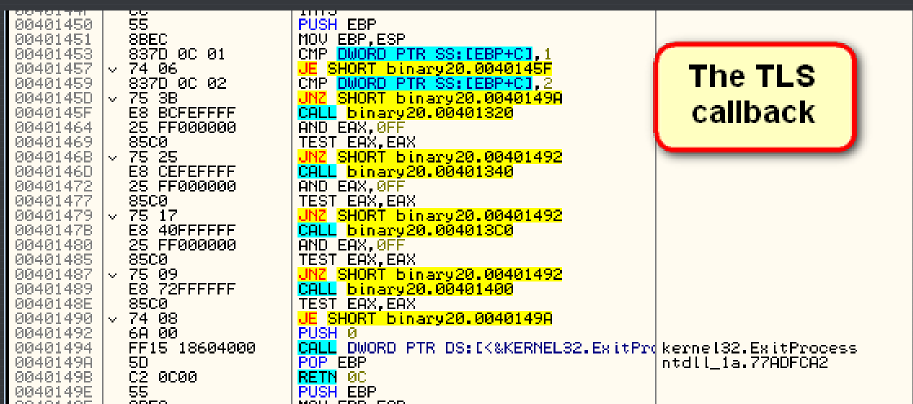

Next is the AddressOfCallbacks. The address here is 407014:



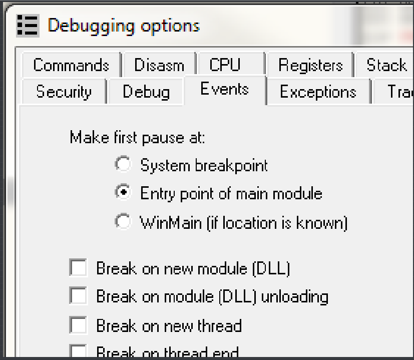

This address should ring some bells…notice that it is at address 7014 after our ImageBase of 40000. This points into the .data section of our binary, the section we are currently looking in. So this address field holds a pointer to a callback, a pointer to another address in the .data section at offset 0×14 (the .data section starts at 0×7000, so 0×7014 is offset 0×14 in section 0×7000). Looking to this address, we see the actual address of the TLS function callback:



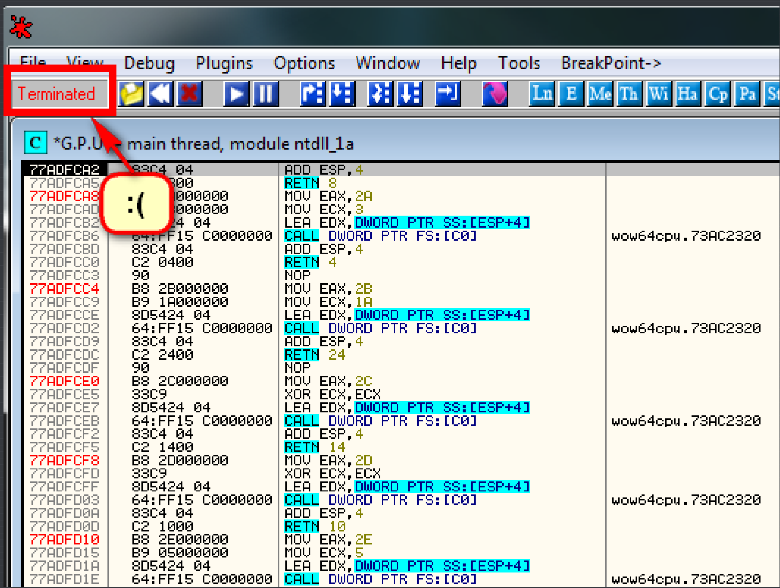

So 401450 is the actual address of the TLS callback code. Let's have a look at this code in Olly:



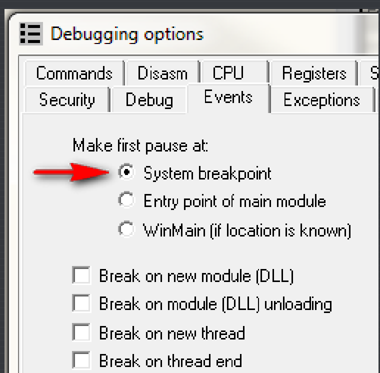

This is the actual code in the TLS callback. Now let's confirm when this callback is actually called. Remove all breakpoints in the code and set Olly to run until the beginning of the main module:




and please make sure the TLSCarch plugin is not in the plugins directory for right now. When we run the app, it automatically terminates, never stopping at the entry point (which is 401000):

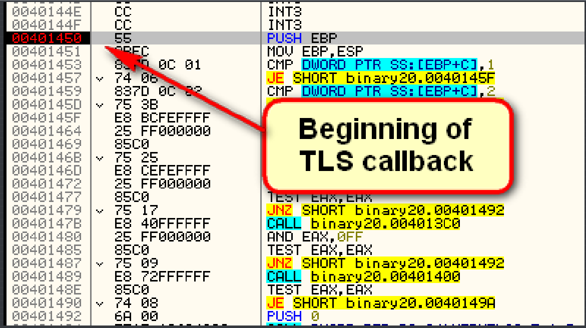



Now let's try something a little different. Set Olly to break at the system entry point:

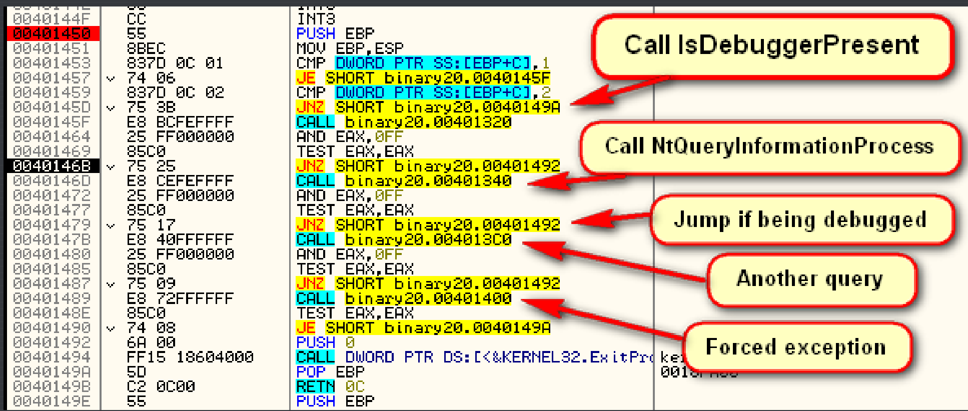



and set a breakpoint at 401000, at the beginning of the actual code. Now, when we re-start the target, we will break in ntdll.dll (before anything has run). Hitting F9 and the target terminates again. We went from the system entry point and never made it to our program's entry point.

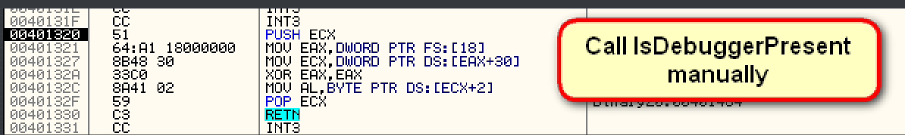
One last thing…place a breakpoint at 401450 (the beginning of the TLS callback) and re-load the target. We first stop at the system entry point. Now run the target. We stop at the TLS callback. This proves that our callback is running between the system entry point and the beginning of our program:



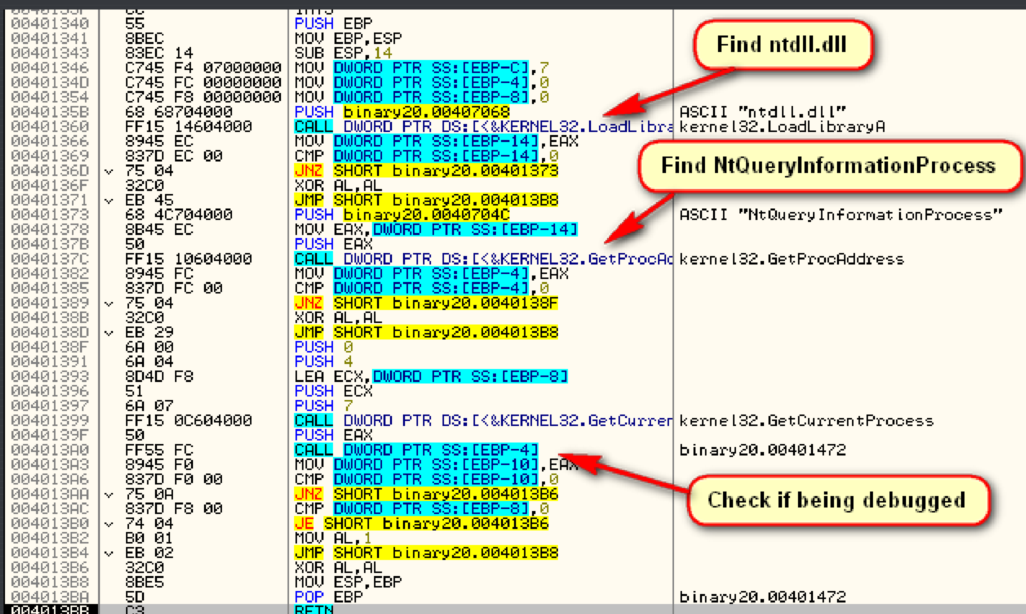

Now that we're here, let's take a look at what this callback actually does:




As you can see, there is some heavy anti-debugging going on here. First is a manual call to IsDebuggerPresent at address 40145F. This calls the following routine:



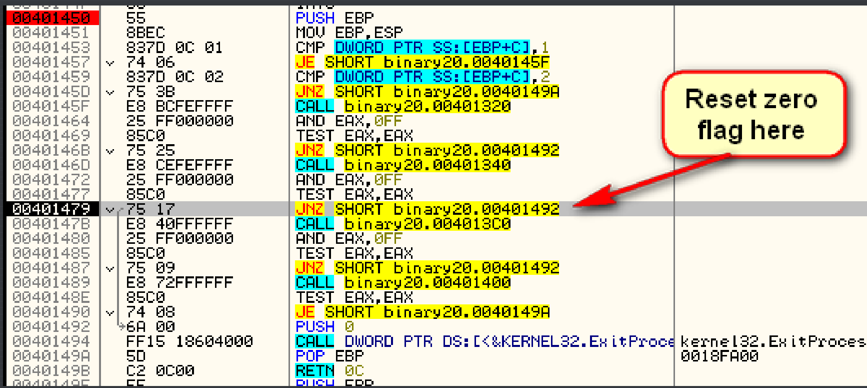

which, if you recall from my last tutorial, is just the manual way of calling this API. Next we call the NQuerryInformationProcess anti-debugging API:



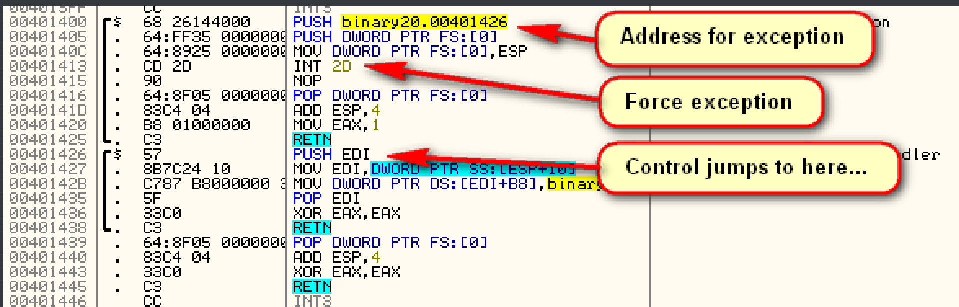

When called with ProcessInformationClass set to 7 (ProcessDebugPort constant), the system will set ProcessInformation to -1 if the process if it is debugged.

Interestingly, this routine further obfuscates itself by loading the address of ntdll and the

NtQueryInformationProcess manually. Next we call another system debug check at 40146D. After this call, we must change the zero flag to keep going (unless you happen to have all of the options in OllyAdvanced set) :




The code then calls it's own exception handler at address 401400:




Here, the target registers its own exception, pointing to address 401426. It then purposely causes an exception, hoping the debugger will get confused. Fortunately, Olly is not confused and passes execution to the proper exception handler at address 401426.

After all this, we finally arrive at the proper entry point, though, this program is very sneaky and later calls the TLS code again, as well as some other anti-debugging techniques. I will stop here as our tutorial is on TLS callbacks and not anti-debugging, but feel free to investigate the target further.

## Multiple TLS Callbacks

Programmers are not limited to only one TLS callback. Let's look at one program that has multiple callbacks and see how it differs. Load TLS_example_1.exe in CFF Explorer and click on the "Data Directories":

| | | | | |
|---|---|---|---|---|
| Reserved | 00000168 | Dword | 00000000 | |
| Reserved | 0000016C | Dword | 00000000 | |
| TLS Directory RVA | 00000170 | Dword | 00003008 | .data |
| TLS Directory Size | 00000174 | Dword | 00000018 | |
| Configuration Directory RVA | 00000178 | Dword | 00000000 | |
| Configuration Directory Size | 0000017C | Dword | 00000000 | |
| Bound Import Directory RVA | 00000180 | Dword | 00000000 | |
| Bound Import Directory Size | 00000184 | Dword | 00000000 | |

Here, we can see the offset of the TLS Directory information is at offset 08 in the .data section, which starts at 03000. Clicking "TLS Directory" in CFF, we see the information displayed in a friendly manner:

TLS_Example_1.exe

| Member | Offset | Size | Value |
|---|---|---|---|
| StartAddressOfRawData | 00000808 | Dword | 00000000 |
| EndAddressOfRawData | 0000080C | Dword | 00000000 |
| AddressOfIndex | 00000810 | Dword | 00403038 |
| AddressOfCallBacks | 00000814 | Dword | 00403020 |
| SizeOfZeroFill | 00000818 | Dword | 00000000 |
| Characteristics | 0000081C | Dword | 00000000 |

The important field here is the AddressOfCallbacks, and we can see it is at offset 03020, or offset 020 in the .data section. Now clicking on the “Section Headers”, and then on the .data section, CFF tells us that the TLS is in this section and shows us a dump:

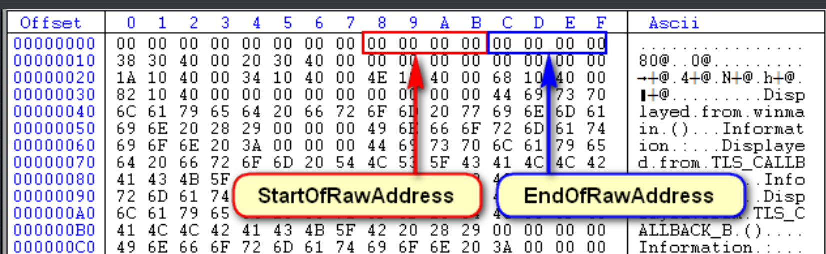
| Name | Virtual Size | Virtual Address | Raw Size | Raw Address | Reloc Address | Linenumbers | Re |
|---|---|---|---|---|---|---|---|
| 000001F8 | 00000200 | 00000204 | 00000208 | 0000020C | 00000210 | 00000214 | 00 |
| Byte[8] | Dword | Dword | Dword | Dword | Dword | Dword | W |
| .text | 000000A8 | 00001000 | 00000200 | 00000400 | 00000000 | 00000000 | 00 |
| .rdata | 00000092 | 00002000 | 00000200 | 00000600 | 00000000 | 00000000 | 00 |
| .data | 0000016C | 00003000 | 00000200 | 00000800 | 00000000 | 00000000 | 00 |

This section contains:

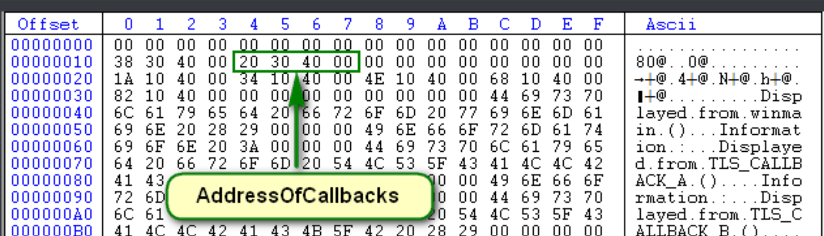
TLS Directory: 00003008

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000010  38 30 40 00 20 30 40 00 00 00 00 00 00 00 00 00  80@..0@.........
00000020  1A 10 40 00 34 10 40 00 4E 10 40 00 68 10 40 00  →+@.4+@.N+@.h+@.
00000030  82 10 40 00 00 00 00 00 00 00 00 00 44 69 73 70  ▌+@.........Disp
00000040  6C 61 79 65 64 20 66 72 6F 6D 20 77 69 6E 6D 61  layed.from.winma
00000050  69 6E 20 28 29 00 00 00 49 6E 66 6F 72 6D 61 74  in.()...Informat
00000060  69 6F 6E 20 3A 00 00 00 44 69 73 70 6C 61 79 65  ion.:...Displaye
00000070  64 20 66 72 6F 6D 20 54 4C 53 5F 43 41 4C 4C 42  d.from.TLS_CALLB
00000080  41 43 4B 5F 41 20 28 29 00 00 00 00 49 6E 66 6F  ACK_A.()....Info
00000090  72 6D 61 74 69 6F 6E 20 3A 00 00 00 44 69 73 70  rmation.:...Disp
000000A0  6C 61 79 65 64 20 66 72 6F 6D 20 54 4C 53 5F 43  layed.from.TLS_C
000000B0  41 4C 4C 42 41 43 4B 5F 42 20 28 29 00 00 00 00  ALLBACK_B.()....
000000C0  49 6E 66 6F 72 6D 61 74 69 6F 6E 20 3A 00 00 00  Information.:...
000000D0  44 69 73 70 6C 61 79 65 64 20 66 72 6F 6D 20 54  Displayed.from.T
000000E0  4C 53 5F 43 41 4C 4C 42 41 43 4B 5F 43 20 28 29  LS_CALLBACK_C.()
000000F0  00 00 00 00 49 6E 66 6F 72 6D 61 74 69 6F 6E 20  ....Information.
00000100  3A 00 00 00 44 69 73 70 6C 61 79 65 64 20 66 72  :...Displayed.fr
00000110  6F 6D 20 54 4C 53 5F 43 41 4C 4C 42 41 43 4B 5F  om.TLS_CALLBACK_
00000120  44 20 28 29 00 00 00 00 49 6E 66 6F 72 6D 61 74  D.()....Informat
00000130  69 6F 6E 20 3A 00 00 00 44 69 73 70 6C 61 79 65  ion.:...Displaye
```

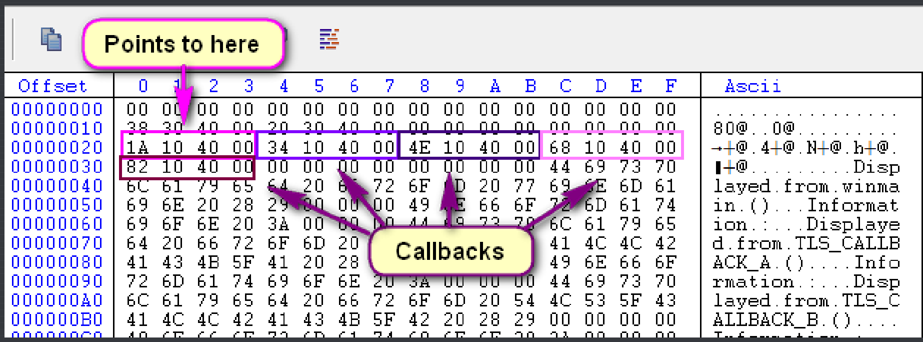
Looking at the raw data, we see the familiar start and end addresses at the beginning of the TLS directory (at offset 08):




Next we see the AddressOfCallbacks (skipping the other fields as they are not important here):

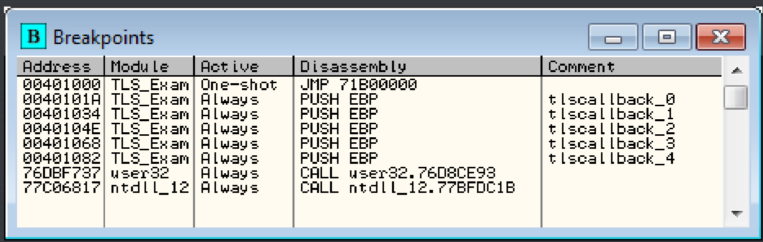



So we know the address of the callback array is at 403020, or offset 03020, or 20 bytes after the beginning of the .data section. Looking at the 20th byte and orward, we see that there are 5 addresses, meaning this binary has five callbacks:




Looking at this, we know that the TLS callbacks are at addresses 40101A, 401034, 40104E, 401068 and 401082.

Now this time, before you load the target in Olly, copy the TLSCatch plugin into the plugins directory. This time, when we load the target in Olly, we see that several breakpoints have been set:




The first breakpoint is the module's main entry point (set because I have the 'break on module's entry point' set in Olly). Next there are 5 breakpoints set, each with a label that begins with "tlscallback_#". This plugin has automatically parsed our binary, extracted the callback address, and has placed a breakpoints on all of the callbacks. Double-clicking one of these shows us the actual code for the callbacks:


Obviously this is a really simple binary, and all that the callbacks do is display a message box, but you get the idea.

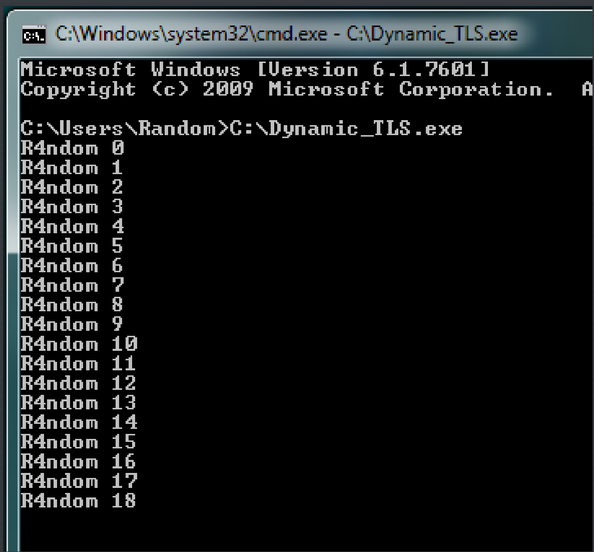
Keep in mind that DLLs can have TLS callbacks just like exe files. This means if we have 3 DLLs that our target requires, all of which have TLS callbacks, when our exe loads, the Windows loader will load each of these DLLs into the target's memory space, and as each is loaded, the callbacks for each will be called. This would be quite a challenge to keep track of. But things can also get a little worse…

## Dynamically Created TLS Callbacks…

One thing that is not widely known (and because of this we're sure to see more of) is the fact that TLS callbacks can be created dynamically, bypassing most of our techniques for discovering them. The way this works is by setting up a single TLS callbcak (or loading a DLL with a callback in it), which then creates another callback dynamically. Our plugin would not catch this, and the callback would not show up in the PE header. The only way to find such a trick would be to start at the system entry breakpoint (in ntdll.dll) and step through until you created the new callback, stepping into it at this time,and debugging it as it's run.

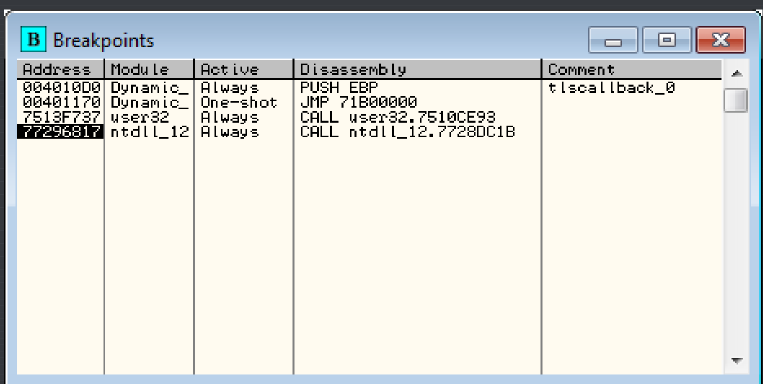
Nothing like keeping things interesting…

Let's take a look at a program that creates TLS callbacks dynamically (thanks to waliedassar for providing the binary). This is a pretty tough executable to reverse in that every time the TLS is called, it basically resets itself to call the TLS callback again. It also has some anti-debugging mechanisms built in. If we run this binary in a command window, we see that a message is displayed over and over with an incrementing counter. This counter is actually keeping track of every time it calls the TLS callback:

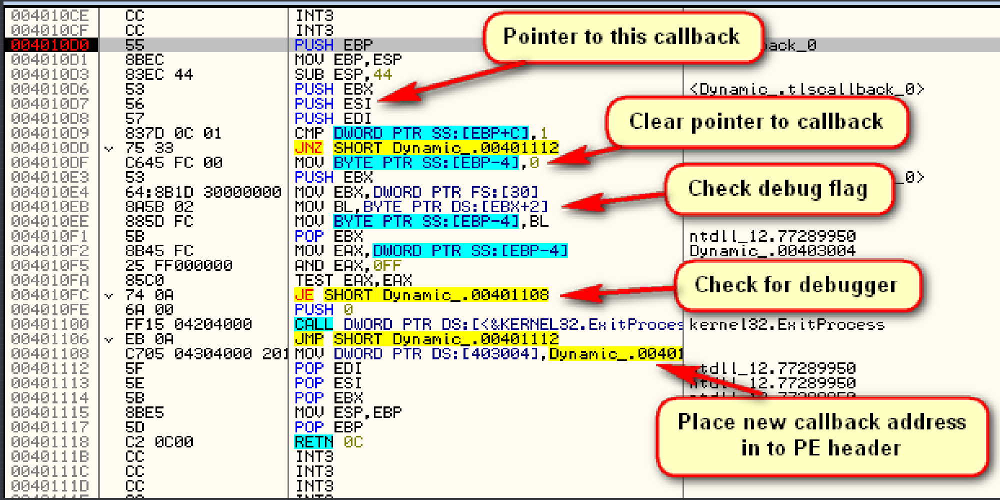



What this program does is modifies itself so that when the TLS is called, it resets it to call it again on the next loop. This loop is deep in the Windows loader. It loads the address of the callback and passes execution to it. It then checks to see if there is another callback, and if there is, it calls it. What the program is doing is making the loader think there is another callback, so the loader keeps calling (the same) callback over and over.

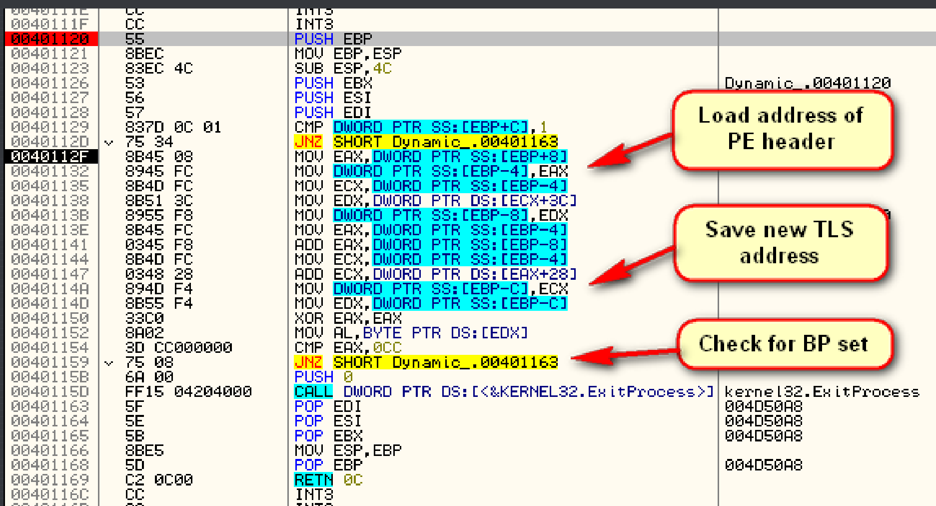
Loading Dynamic_TLS.exe into Olly, we see that Olly has found the first TLS callback:



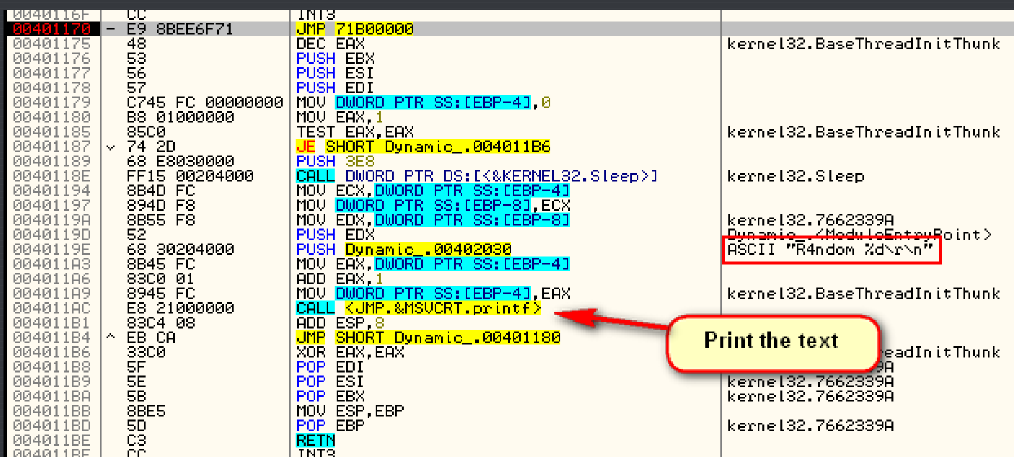

Double-clicking on the tlscallback_0 line, Olly takes us to the actual callback code:




This routine first does some housekeeping, then checks if we're being debugged and exits if we are. If not, it loads another address into the callback array, so that the loader will call this next address ( 401120 ). It then returns control to the loader. The loader then calls what it thinks is the next TLS callback at address 401120. TLS Catch will not break at this new TLS callback, as it was created dynamically:




This routine creates yet another TLS callback at address 401163. It also checks if there is a breakpoint set on this routine and exits if there is. It then returns to the loader which now calls the third callback:




This routine then quietly calls printf to display the message and sets the TLS callback back to the original entry of the first callback. This makes the loader start the process all over again.

This binary is obviously an example of an extreme case, though packers and malware are always looking for extreme cases, so don't be surprised if you don't see something like this in the near future.

## Making Our Own TLS Callback

For the really sadistic out there, I have decided to include a section on making our own binary that has a TLS callback so you can investigate it further. I will use RadASM to create a binary that does nothing but call our own callback, displaying a goodboy or badboy depending on if we're being debugged or not (though this won't work if you are using a plugin that hides Olly).

First, we create an empty Win32 project. I have called it, surprisingly, "TLS Callback". Now create a "TLS Callback.Asm" file and enter the following data (I have also included the source file for this project if you would like to save yourself some typing):

```
.586
.model flat, stdcall
option casemap :none   ; case sensitive

;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
; Includes                                                    ;
;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\comdlg32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\comdlg32.lib


;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
; Data                                                        ;
;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
.data
    DbgNotFoundTitle db "Debugger status:",0h
    DbgFoundTitle db "Debugger status:",0h
    DbgNotFoundText db "Debugger not found!",0h
    DbgFoundText db "Debugger found!",0h

;   TLS Structure

    dd offset   StartAddress
    dd offset   EndAddress
    dd offset   AddressOfIndex
    dd offset   TlsCallBack

    TlsCallBack dd  offset TLS      ; Address of our callback
    dd 0                            ; Spacer
    dd 0                            ; Spacer
    StartAddress     dd     0
    EndAddress       dd     0
    AddressOfIndex   dd     0
    TlsCallBack2     dd  offset TLS
    SizeOfZeroFill   dd     0
    Characteristics  dd     0

.data?
    TLSCalled db ?              ; Flag for if TLS has been called


;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
; Code                                                        ;
;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
.code

start:
    invoke ExitProcess,0
    ret

;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;
; TLSCallback                                                 ;
;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-;

TLS:

    CMP BYTE PTR[TLSCalled],1           ; If callback has already been run
    JE @exit                            ;   ignore it
    MOV BYTE PTR[TLSCalled],1           ; Set flag for next time
    CALL IsDebuggerPresent

    CMP EAX,1                           ; Are we being debugged?
    JE @DebuggerDetected                ;   yes

    PUSH 40h                            ; Show goodboy
    PUSH offset DbgNotFoundTitle
    PUSH offset DbgNotFoundText
    PUSH 0
    CALL MessageBox
    JMP @exit

@DebuggerDetected:
    PUSH 30h                            ; Show badboy
    PUSH offset DbgFoundTitle
    PUSH offset DbgFoundText
    PUSH 0
    CALL MessageBox

@exit:
    RET

end start
```

You can see that I initially create a structure that resembles the TLS structure. I then populate the callback address, TlsCallBack2, as the offset of our TLS code. The main routine does nothing but quits. Finally, the TLS code checks IsDebuggerPresent and displays the appropriate message depending on the results.

This binary keeps track of a flag for if the callback has been called or not. This is because TLS calls can come both at the beginning and at the end of a programs life cycle. We only want to run ours once, hence the flag.

After building the binary, we must change the TLS info inside of the PE header. Load our compiled program into CFF Explorer and click on the Data Directories tab:

| | | | |
|---|---|---|---|
| Architecture Directory Size | 00000174 | Dword | 00000000 |
| Reserved | 00000178 | Dword | 00000000 |
| Reserved | 0000017C | Dword | 00000000 |
| TLS Directory RVA | 00000180 | Dword | 00000000 |
| TLS Directory Size | 00000184 | Dword | 00000000 |
| Configuration Directory RVA | 00000188 | Dword | 00000000 |
| Configuration Directory Size | 0000018C | Dword | 00000000 |

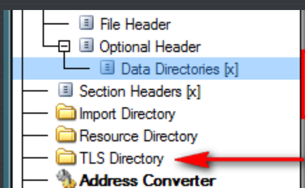
You will notice that there is no TLS information in the binary. Clicking on the Section Headers tab, then on the .data section, we see that our TLS is actually in there and it begins at offset 0×46:

| | | | | | | |
|---|---|---|---|---|---|---|
| .text | 0000005E | 00001000 | 00000200 | 00000400 | 00000000 | 00000000 |
| .rdata | 000000AE | 00002000 | 00000200 | 00000600 | 00000000 | 00000000 |
| .data | 00000081 | 00003000 | 00000200 | 00000800 | 00000000 | 00000000 |
| .rsrc | 00000010 | 00004000 | 00000200 | 00000A00 | 00000000 | 00000000 |

This section contains:

```
Offset     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    Ascii
00000000  44 65 62 75 67 67 65 72 20 73 74 61 74 75 73 3A   Debugger.status:
00000010  00 44 65 62 75 67 67 65 72 20 73 74 61 74 75 73   .Debugger.status
00000020  3A 00 44 65 62 75 67 67 65 72 20 6E 6F 74 20 66   :.Debugger.not.f
00000030  6F 75 6E 64 21 00 44 65 62 75 67 67 65 72 20 66   ound!.Debugger.f
00000040  6F 75 6E 64 21 00 62 30 40 00 66 30 40 00 6A 30   ound!.b0@.f0@.j0
00000050  40 00 56 30 40 00 08 10 40 00 00 00 00 00 00 00   @.V0@..+@.......
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 10   ..............+
00000070  40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   @...............
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Now, clicking back in the Data Directories, double-click in the TLS  Directory RVA and change it to 3046. Then change the TLS Size to 18. Now save the binary (I saved it as “TLS Callback_modified.exe” then re-load it in CFF Explorer. We can see that our TLS is there and that CFF Explorer has created a directory for it:

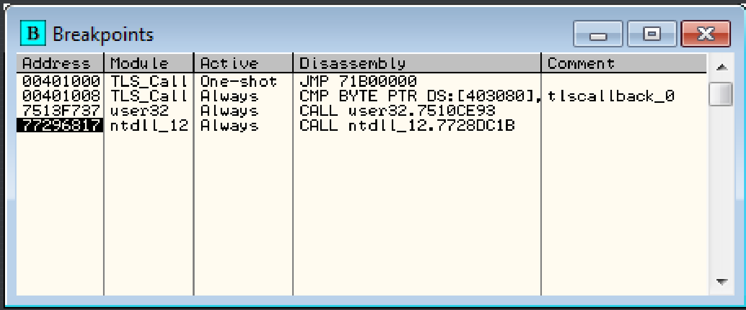



| | | | | |
|---|---|---|---|---|
| Reserved | 0000017C | Dword | 00000000 | |
| TLS Directory RVA | 00000180 | Dword | 00003046 | .data |
| TLS Directory Size | 00000184 | Dword | 00000018 | |
| Configuration Directory RVA | 00000188 | Dword | 00000000 | |
| Configuration Directory Size | 0000018C | Dword | 00000000 | |

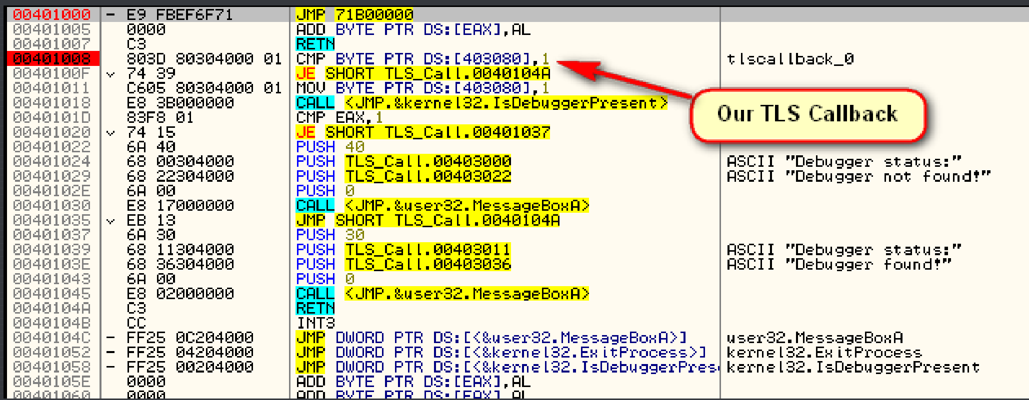
Clicking on the TLS Directory tab, we see the information we hard-coded into the binary:

| Member | Offset | Size | Value |
|---|---|---|---|
| StartAddressOfRawData | 00000846 | Dword | 00403062 |
| EndAddressOfRawData | 0000084A | Dword | 00403066 |
| AddressOfIndex | 0000084E | Dword | 0040306A |
| AddressOfCallBacks | 00000852 | Dword | 00403056 |
| SizeOfZeroFill | 00000856 | Dword | 00401008 |
| Characteristics | 0000085A | Dword | 00000000 |

Now load the binary in Olly. There is now a breakpoint for our callback routine in the breakpoints window:




and double-clicking on this, we can see our actual TLS callback:




and if you run the app, you will see that it works just like expected…

Special Thanks to MRHPx for his injection info, Ange Albertini , ax0s, and Waliedassar & Eric Carrera for help with the more technical stuff.

-Till next time

R4ndom