Code Obfuscation in 64-Bit Land

Russ Osterlund, 11 Dec 2016 CPOL

5.00 (6 votes)

Code Obfuscation in 64-Bit Land

Download source - 615.1 KB

(**Note**: If you are unable to download the source code from this link, you will also find it on the SmidgeonSoft website - travel to the "News" page and look for the entry dated 02/14/2016. An updated version (dated 12/11/2016) of the source has been uploaded to and a link made available on my website - a pattern found in the HDSpoof tutorial has been adapted and incorporated into the code adding stress to a debugger and making it harder to reverse.)

Introduction

During my career as a reverse-engineer, I have been tasked on several occasions with finding the reason why my employer's product would mysteriously fail in the presence of a 3rd party application. Without access to the original source code, inevitably, I would fire up a static analyzer and/or debugger and start stepping through the binary code, trying to understand what it was doing and how it was interfering with our product's operation. On my website, www.smidgeonsoft.com, you can find more detailed documentation about two of these incidents (as well as two utilities, **PEBrowse64 Professional** and **PEBrowseDbg64 Interactive**, used during the creation of the project):

- "Reversing HDSpoof - A Tutorial"
- "Exposing a Resource Leak in Yoda's Protector"

Another project was not documented because it required reversing copy-protection by an installer, but, nevertheless provided inspiration for what you will learn about later in this article. As a result of these adventures and other experience, I am convinced of the following...

If It Can Be Debugged, It Can Be Reversed!

I hope in this discussion to pass along this motto/belief by examining a sample program employing some of the tricks and traps I picked up in my reverse-engineering adventures. The aforementioned projects were all 32-bit challenges. I have found little in the way of examples of how some of these techniques can be implemented in 64-bit platforms, hence, my offering of . You will need **Visual Studio 12** or greater in order to build the binaries. However, any mistakes, errors, or bugs in the source code I proudly call my own - the code is provided AS-IS and can be found in the attached file, *SimplestWithFooSourceCode.zip*.

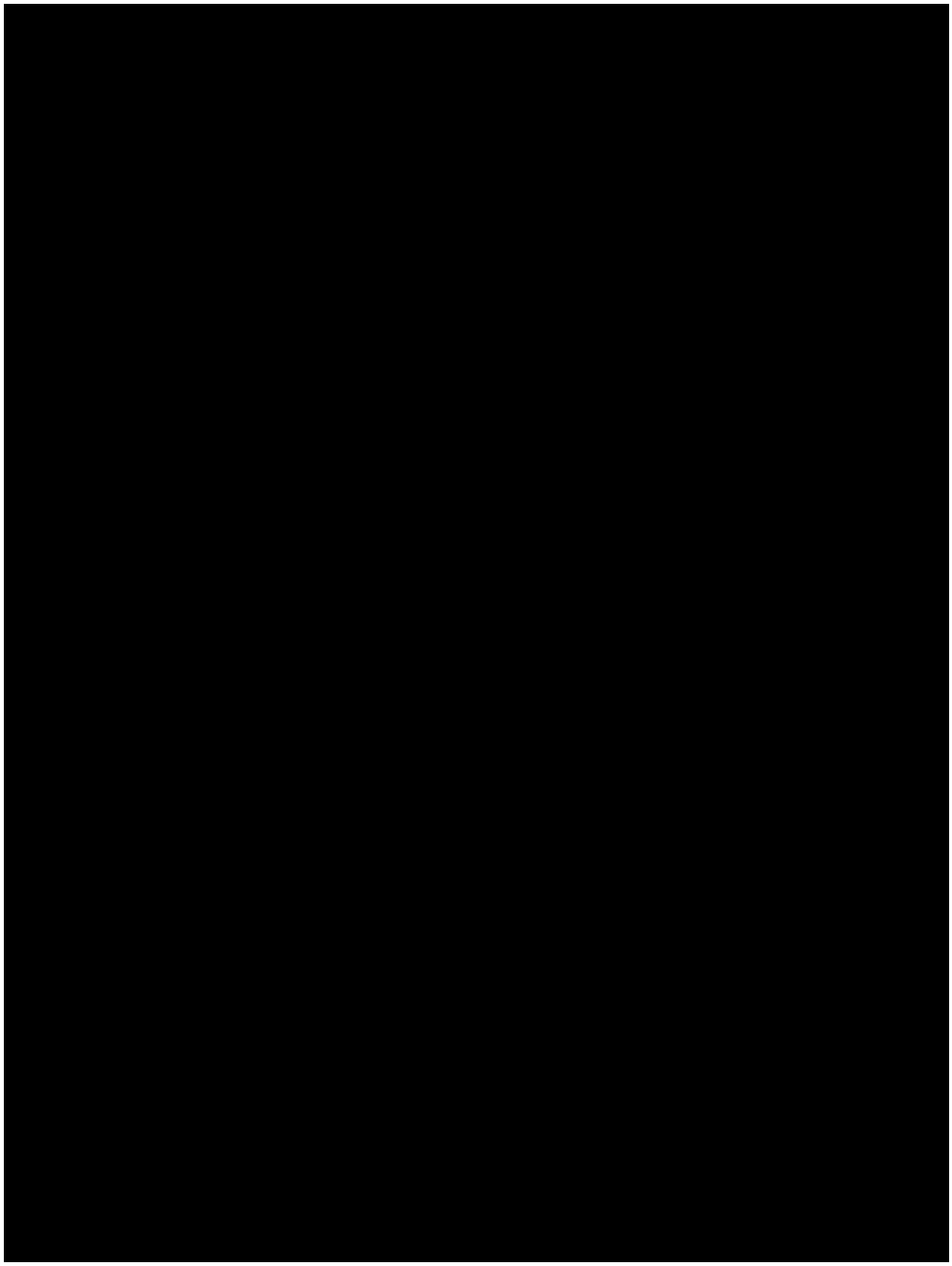
Simplest

The main program in this article, , started out as an attempt to create a trivial Windows program that contained no imports. e.g. , , etc. Inspired by another source:

- TinyPE NG - http://www.ragestorm.net/blogs/?p=46

I was able to produce a program with the structure you can see in this image (displayed using **PEBrowse64 Professional**, a static-analyzer you can download from my website's software page):

As anyone with some knowledge of Portable Executable files and experience investigating unknown executables can see, the program structure looks a bit odd - no import directory or imports list - not even **NTDLL** - and some missing sections, e.g., . Launching the original would produce a simple message-box displaying "Ok". But it required some ingenuity to reach this result. Loading the needed support DLLs and accessing their APIs meant developing code without the usual , mechanisms. Adding to my self imposed challenge, I decided to add to the code base some of what I had seen in previous projects, i.e., deliberate obfuscation of code that would defeat any attempts at static analysis. Also, these obfuscated programs introduced tricks designed to thwart debuggers and any other tools that could be employed "to see what was going on." This led to the introduction of structured exception handling, which meant I now had to abandon the goal of no import references in .



Along with the exception handling support routines from (having little desire to create my own) and the reference to , note the presence of the mysterious binary resource, "*BIN*" and the addition of several more directories. As an aside, I resisted at this point changing the name of the program to- *Simple.EXE*.

Foo.exe

Before plunging into the innards of , let us first take a brief tour of the executable *Foo.exe* embedded in the binary resource, "*BIN*" (and found in the separate project and source file, *Foo.cpp*). was initially generated by the **Visual Studio** wizard to produce a program containing a call that would display either "Okay" or "Fail" depending on a simple anti-debugging trick - Was the program launched independently (presumably by extracting or saving the binary resource after an earlier reversing session) or was it launched by the parent program, ? Two sequences of calls to functions determine the parent process and return a value. Although not very sophisticated, this code and the entire program, , are stand-ins for more complicated ideas needing obfuscation, such as defending copy-protection, hiding proprietary algorithms, or shielding secret code/program defences. Producing "Okay" inside of would signify the defeat of the protection code, while "Fail" meant that the code was not successfully reversed.

SimplestWithSpeedBumps Organization

's (or) design goal is a bit "different". That is to say that launching outside of a debugger will result in the creation of the Foo process and the display of the message-box with the word, "Fail". However, starting in a debug session will end in a program crash or (after successfully navigating the tricks and traps of the program) launch the program and display the word, "Okay". (The - suffix is merely a whimsical idea acknowledging my objective of slowing down a reversing guru.)

The program structure for is fairly straightforward, though more complicated than what one usually sees in a normal C++ program. There is supporting code found in three separate source code files: , , and . The first, , contains implementations for the routines, , , and - all replacements for what you will find in 's APIs. These functions in part rely upon locating the address to the Program Environment Block, or PEB, and its reference to the address of the loader table (whose structure definition and description can be found in many places out on the internet). 's role answers requests for comparisons and lengths. houses three anti-debugging tricks, all designed to detect the presence of a debugger in the process.

There is a "bridging" layer that assists in connecting the C++ code with some obfuscation code written in IA64 assembly language. One of the things I discovered while writing the code was that **Visual Studio** dropped its support for inline assembler instructions, i.e., the reserved token, , at some point, and, thus, forced code like this to be contained in a separate source file (you may need to change the directory pointer in the project to match the location of the assembler found on your system). There are also a series of callbacks in the assembler source used to transfer control back to the C++ routines doing the "heavy lifting". These callbacks were purposefully written to be repetitive and boring (except for one!) with the goal of adding to the obfuscation.

Finally, the C++ layer, as stated before, performs the heavy lifting, e.g., extracting the program from the resource section, creating and writing the temporary, independent file, calling a create process API, deleting the temporary file, and, finally, cleaning up by issuing calls to . While perusing the source code here, you may be tempted/inspired by some of the techniques to add more obstacles. Although the code does not look efficient, keep in mind that the purpose is to confuse the person debugging the code and that most of this code will only execute once.

SimplestWithSpeedBumps Obfuscation

Now, we can get to the heart of this article - code obfuscation and the ideas behind it. The main entry-point "gets our feet wet" and "sets the table". Right out of the starting gate, the code calls a routine called - notice there is no or other similar code. Stepping into the routine with a debugger or carefully analyzing the instructions will show you that the following is what is executed.

Most of the instructions are nonsense and, except for the careful balancing of pushes and pops on the stack, the only meaningful instruction is the one before the return, i.e., returning the contents of R8. Obtaining the address to the PEB allows later calls to , etc. do their thing. The choice of R8 is practically the only fragile assumption in all that follows - the assignment of the PEB value and where to look at program startup has already changed from prior versions of Windows.

Besides the bogus reference to , the next items are an access violation and all of the remaining code - contained inside of an exception handler! After generating the initial access violation (any severe error will do) and "handling it", any further unhandled exceptions will result in the program crashing. This creates "tension," the first of several elements that should be the design objectives in creating obfuscated code. Here are some other elements:

- tension
- misdirection/deception
- fatigue/no shortcuts
- boredom
- eliminating strings and other meaningful bits of data, and
- hiding meaningful values in plain-sight.

The presence of these elements can be found throughout the source-code to . The tension introduced by the initial exception is increased by numerous checks in the code for things like the presence of a debugger and debugger breakpoints all of which will generate, if missed or not recognized, another access violation - this was originally a call to but then inlined to avoid **NOP-**ing (modifying the code during the debug session) the routine away. Any of these will eventually crash the program. Examples of misdirection or deception can be found in the initial bogus call but also in the checks for breakpoints on , , and other process creation APIs (mostly to thwart those who know a second process is created and are trying to shortcut the reversing exercise by guessing where the process will be created). Fatigue sets in while stepping through seemingly meaningless code and some lengthy loops found in . Or, it can come about as a result of missing one of the register value changes needed while debugging, seeing the program crash, and then having to start at the beginning again. The dull sameness and boring appearance of the routines not only serve to warn of an impending transition to or from C++ code, but also to reserve large stack areas, perhaps large enough to remove easily visible clues among debris of stack garbage. Only one of these allocates a different stack area, but that was added only after fixing a real crash inside of due stack alignment - see the code comment for more information. An attempt has been made to avoid the use of s until the last possible moment by assigning, character by character (sometimes in random order) the name of the API service routines used by . And, when an address is returned by , in order to avoid early detection, the address value is negated to hide it in "plain-sight".

Debugging/Reversing the Code

While developing , I almost exclusively used my own debugger, **PEBrowseDbg64 Interactive** (available from my website's software page). Indeed, some of the areas where I think my debugger is superior to others, e.g., displaying context whenever and wherever possible, intelligent disassembly of code, etc., provided inspiration on ways to further obfuscate the code. In reality, most any heavy-duty debugger should do the trick, i.e., it should support register/memory content manipulation and resetting/changing the execution path of a program. I have not attempted to debug the program though with **Visual Studio**, so I cannot say how successful that exercise might be - YMMV.

Closing Thoughts or How to Improve the Obfuscated Code

In addition to suggestions and ideas found as comments in the source-code, there are almost unlimited ways one can find to slow down others from stepping through the code and discovering its secrets. First and foremost and probably the easiest - increasing the volume of the noise in the code like adding more spurious and imaginative instructions and sequences. Some of these will require a more than passing knowledge of assembly language - like how the addition of a few bytes can result in the complete jumbling of a code sequence - the HDSpoof article contains references to some aesthetically pleasing examples. This technique interferes with all but the most determined static analysis almost forcing the use of a debugger. Code self-modification, like simply negating a large section of code and then reversing the process before execution, is another possibility. Complicating the startup of the entire process, though difficult to

debug, would be an inviting thought. For example, could launch another copy of itself and test whether the code was the first or second instance and follow one code path or another as needed. And testing a debugger's capability to debug different PE types, one could create 32-bit and 64-bit versions of with the 64-bit launching the 32-bit and checking for *bitness* to determine this critical path in the code. And, there are other ways to check if code is being debugged that could be added as more anti-debugging tricks. In the final analysis, you must weigh the additional effort and reward against the fact that your scheme can and will be decoded by any determined reverse engineer.

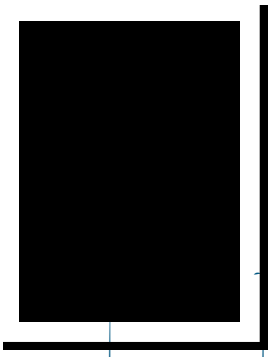
With this article, I think I have stripped away some of the mystery and danger of working with obfuscated code. If not inspirational for finding new ideas, tricks, and traps on your own, I hope to have convinced you of my opening gambit:

If it can be debugged, it can be reversed!

License

Share

About the Author



Russ Osterlund

United States

My name is Russell Osterlund and I live in Merrimack, NH. I was an independent consultant and software developer but am now retired. I can be reached at: RussellOsterlund@gmail.com or via my website: www.smidgeonsoft.com.

You may also be interested in...

C a S M Ab S a c C d S a c C d A a a W d
A a

Ob ca Ma d b c d SAP - N ca - P c D a

Comments and Discussions

 **4 messages** have been posted for this article Visit **https://www.codeproject.com/Articles/1159077/Code-Obfuscation-in-Bit-Land** to post and view comments on this article, or click **here** to get a print view with messages.