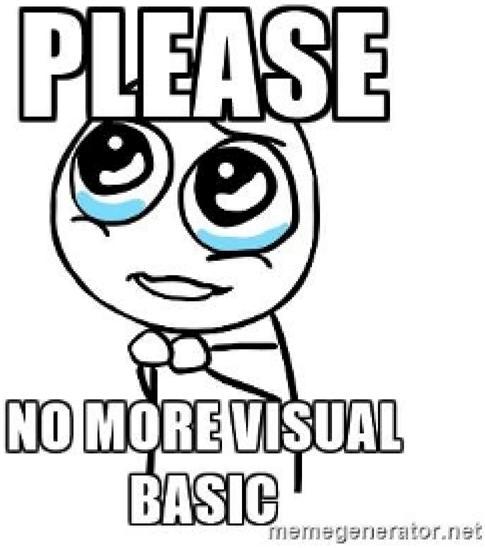
Defeating the VB5 Packer



JUNE 7, 2017JUNE 7, 2017 ~ R3MRUM

The other day I came across a sample (of Pony Malware (https://www.knowbe4.com/pony-stealer)) that had been packed with a VB5 Packer. This was my first introduction to the VB Packer, so I decided to dig in and learn something new.

[Original File Details]
Filename: “PAYMENT ACCEPTANCE COPIES_ JPEG image001.pdf.z”
MD5: 0cd2c6ad9b27cf94debc052113240543
SHA1: 730b761f833a0f0636b04ba315c5ceeedd0baaba
SHA256: 51a2ba55c6134ac346152eb1fd1511248c89b40ac1f6c123c2e4d7541c6ba2ee
Delivery Mechanism: Email Attachment
VirusTotal: File Not Found

This compressed archive contained a single file:

[Decompressed File Details]
Filename: “PAYMENT ACCEPTANCE COPIES_ JPEG image 001.bat”
MD5: 0b6f5f335a7736087a29140082bdd42c
SHA1: b2dbdd3ae1315e50a699f8a271e66ffb506e48f9
SHA256: 50c1454fc1e0d7ed46b92339ff9b85e6be40c3d83492558a4af9b704bd677954
VirusTotal: (06/60) as of 06/06/2017 (Link (https://www.virustotal.com/en/file/50c1454fc1e0d7ed46b92339ff9b85e6be40c3d83492558a4af9b704bd677954/analysis/))

Despite having the extension “.bat”, PEStudio tells me that this is a Microsoft Visual Basic v5.0/v6.0 executable (Figure 1).

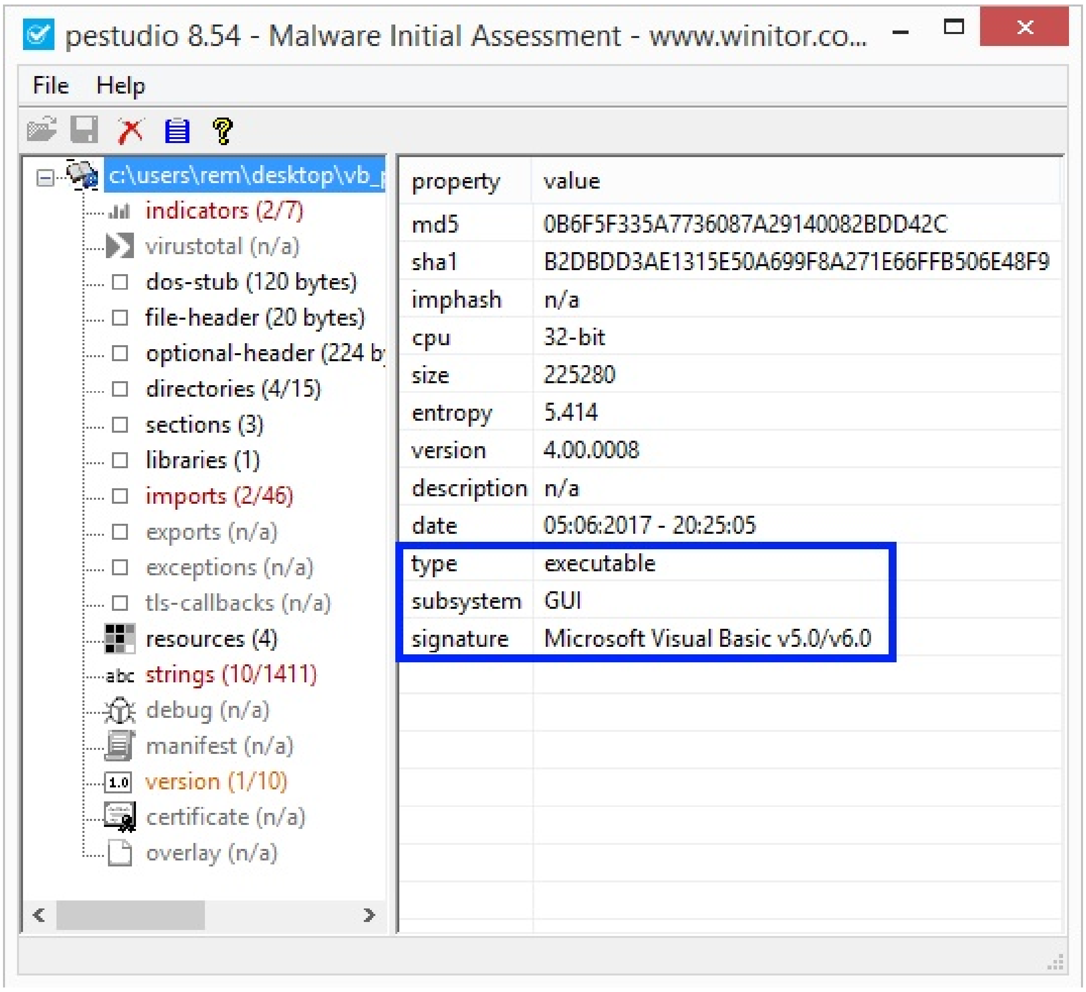


Figure 1: PEStudio depicting sample is a VB Executable

Something that isn't relevant to the unpacking, but might be relevant to tracking, are some details found within the executable's version information:

CompanyName: SPicEvpn.Com (http://www.spicevpn.com/)
ProductName: worlDCoin (https://worldcoin.global/)
OriginalFilename: Vitamina.exe

In the analysis that follows, I have renamed "PAYMENT ACCEPTANCE COPIES_ JPEG image 001.bat" to "Vitamina_Packed.exe".

VBPacker – ThunRTMain

Loading this sample into OllyDBG, you see the following (Figure 2):

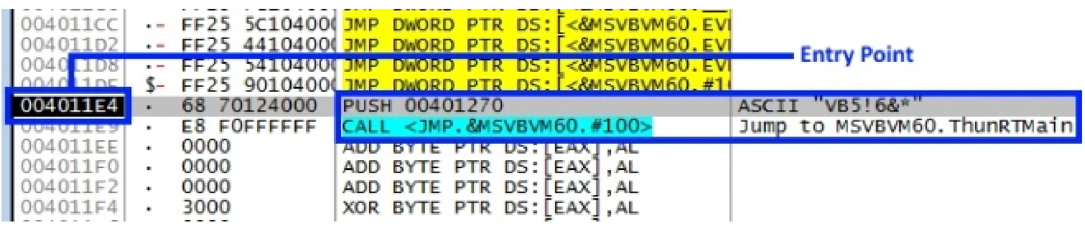
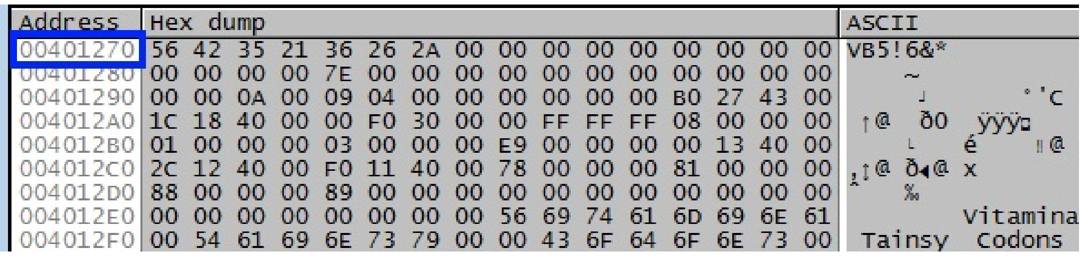


Figure 2: Sitting at Entry Point of the sample

In the image above, we are sitting at the very first instruction for the executable which is pushing an address to the stack (Arg1) before calling MSVBVM60.ThunRTMain. Looking up the ThunRTMain function (https://www.vb-decompiler.org/pcode_decompiling.htm), we find that it takes a single argument (The address 0x401270 being pushed to the stack) and that argument is a pointer to a VBHeader Structure that tells the application how to start. This VBHeader Structure appears in memory, like so (Figure 3):



```
Address  Hex dump                                         ASCII
00401270 56 42 35 21 36 26 2A 00 00 00 00 00 00 00 00 00 VB5!6&*
00401280 00 00 00 00 7E 00 00 00 00 00 00 00 00 00 00 00     ~
00401290 00 00 0A 00 09 04 00 00 00 00 00 00 B0 27 43 00       °'C
004012A0 1C 18 40 00 00 F0 30 00 00 FF FF FF 08 00 00 00  @  ð0  ÿÿÿ
004012B0 01 00 00 00 03 00 00 00 E9 00 00 00 00 13 40 00         é    @
004012C0 2C 12 40 00 F0 11 40 00 78 00 00 00 81 00 00 00 , @ ð @ x
004012D0 88 00 00 00 89 00 00 00 00 00 00 00 00 00 00 00    ‰
004012E0 00 00 00 00 00 00 00 00 56 69 74 61 6D 69 6E 61         Vitamina
004012F0 00 54 61 69 6E 73 79 00 00 43 6F 64 6F 6E 73 00  Tainsy  Codons
```

Figure 3: VBHeader Structure in memory

Parsing these values into the fields they represent, we get the following (Figure 4):

Figure 4: Parsed VBHeader Structure

FIELD	BINARY VALUE	DESCRIPTION
Signature	[56 42 35 21]	VB5!
RuntimeBuild	[36 26]	
LanguageDLL	[2A 00 00 00 00 00 00 00 00 00 00 00 00 00]	
BackupLanguageDLL	[7E 00 00 00 00 00 00 00 00 00 00 00 00 00]	
RuntimeDLLVersion	[0A 00]	
LanguageID	[09 04 00 00]	
BackupLanguageID	[00 00 00 00]	
aSubMain	[B0 27 43 00]	Address representing the true start of the unpacking code
aProjectInfo	[1C 18 40 00]	
fMDLIntObjs	[00 F0 30 00]	
fMDLIntObjs2	[00 FF FF FF]	
ThreadFlags	[08 00 00 00]	
ThreadCount	[01 00 00 00]	
aGUITable	[00 13 40 00]	
aExternalComponentTable	[2C 12 40 00]	
aComRegisterData	[F0 11 40 00]	
oProjectExename	[78 00 00 00]	References the string “Vitamina”
oProjectTitle	[81 00 00 00]	References the string “Tainsy”
oHelpFile	[88 00 00 00]	References a NULL value
oProjectName	[89 00 00 00]	References the string “Codons”

Of these values, the address assigned to aSubMain is the most important as it is the address for the main function that will be called once the executable's environment has been set up. If you set a breakpoint on this address and then allow the sample to run until the breakpoint is reached, you will find yourself at the true start of the unpacking code (Figure 5).

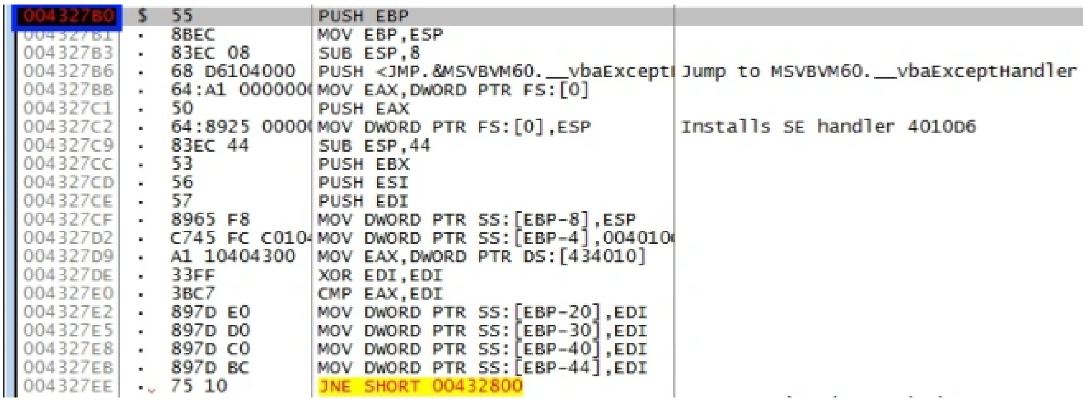


Figure 5: Sitting at true Entry Point of unpacking code

Anti-Analysis #1: Debugger Check [PEB.BeingDebugged]

This sample actually had two different checks for the presence of a debugger. The first being a check for the BeingDebugged flag within the Process Environment Block (PEB). To see what this value currently is, Press CTRL+G within OllyDBG and enter "**FS:[30]**" (sans quotes) as the expression to follow. This will take you into the PEB where you will find that the BeingDebugged flag is set to True (Figure 6).

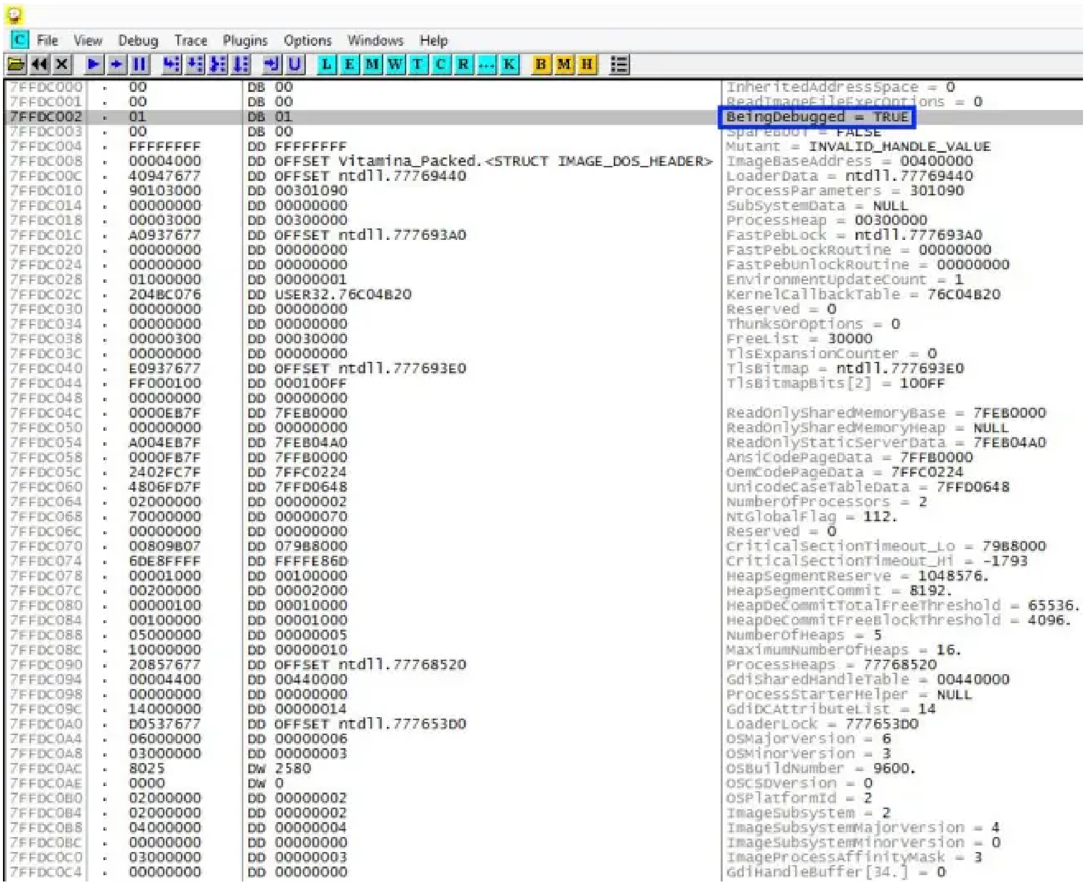


Figure 6: BeingDebugged flag within the Process Environment Block (PEB)

When the malware inspects this value, it learns that it is executing within a debugger and, as a result, the application never unpacks the malicious code. Instead, it just sits idle giving the appearance that it is executing when in fact it is not actually doing anything.

Now, there are multiple methods that we could employ to bypass this check but the one I typically use is the OllyDBGv2 plugin OllyExt, which provides a simple checkbox that enables the bypass (Figure 7).

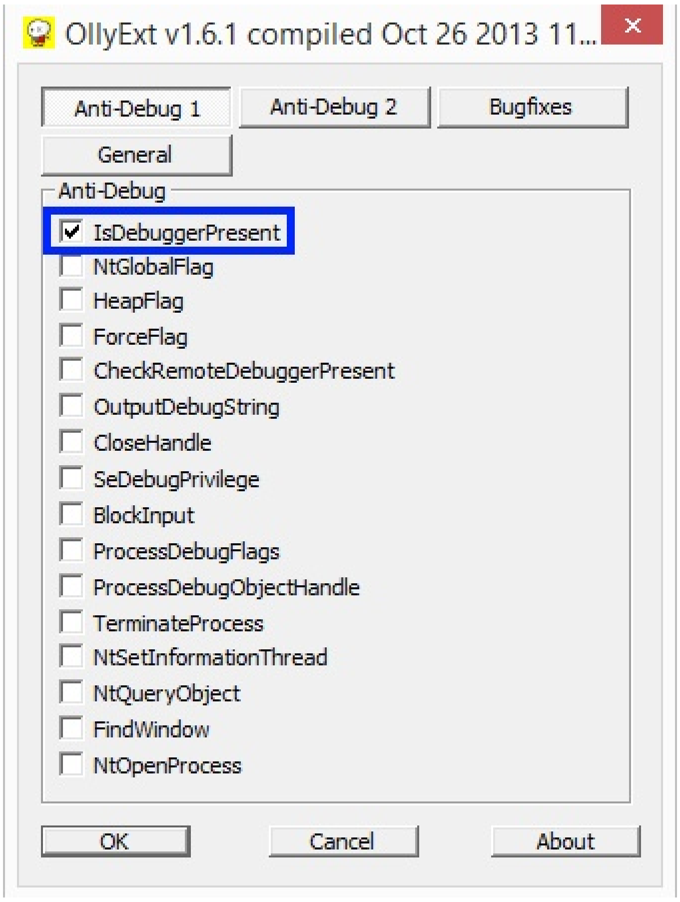


Figure 7: OllyDBG’s OllyExt IsDebuggerPresent bypass

Simply check the IsDebuggerPresent box and restart your application.

Now, if we revisit the BeingDebugged value within the PEB, we see that its value is set to False (Figure 8):

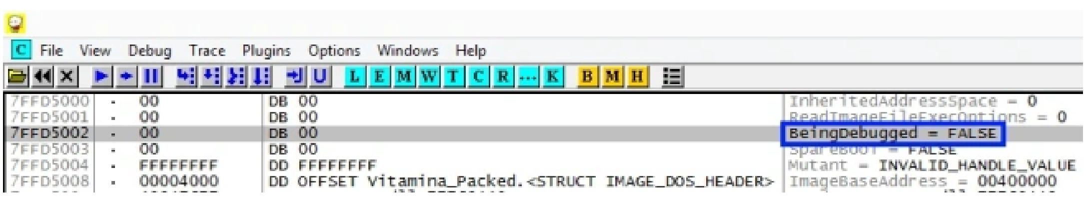


Figure 8: PEB.BeingDebugged flag after enabling OllyExt bypass

An additional indicator that we have successfully bypassed this check is by inspecting CPU utilization within Process Hacker. When BeingDebugged == True, and we allow the sample to fully execute without any breakpoints in place, we find that the CPU utilization for the sample is so low that it doesn’t even register a value (Figure 9).

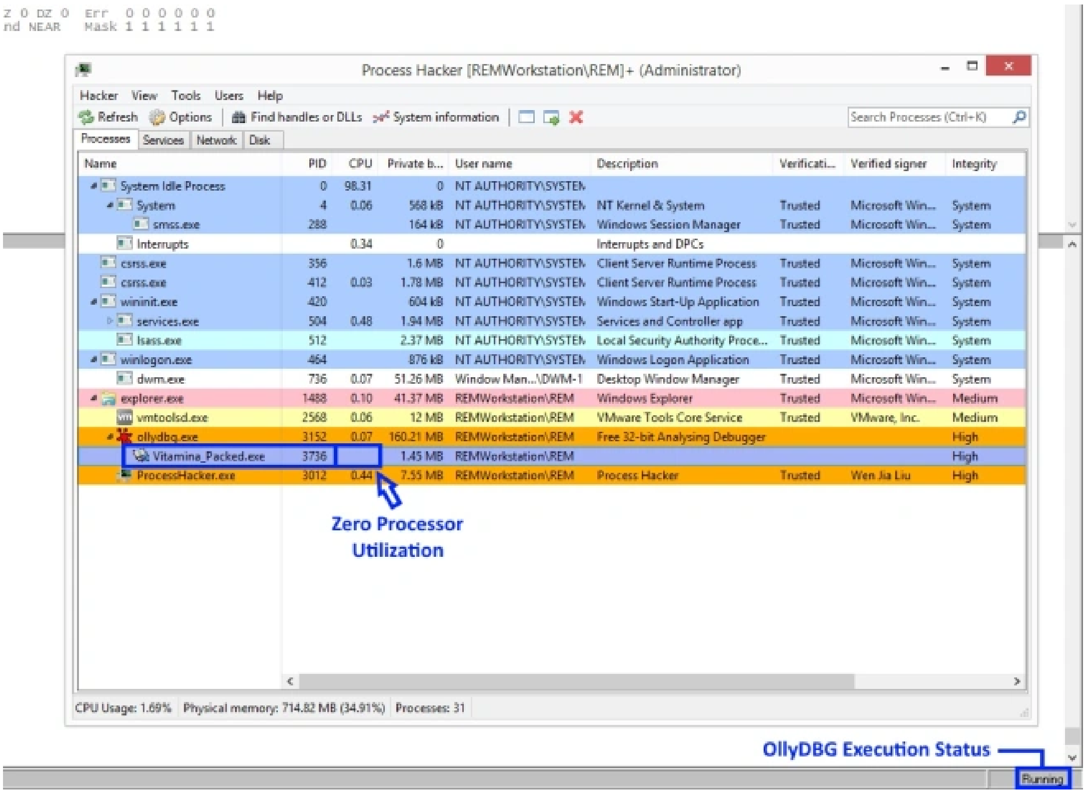


Figure 9: Process Hacker – CPU Utilization before OllyExt bypass

However, when we trick the sample into thinking it isn't running in a debugger (via OllyExt –> IsDebuggerPresent), and allow it to fully execute, we see that the CPU utilization peaks to ~49%* and remains at this level for ~45 seconds* (Figure 10).

** values and timing will vary based on the resources you have provided your virtual machine*

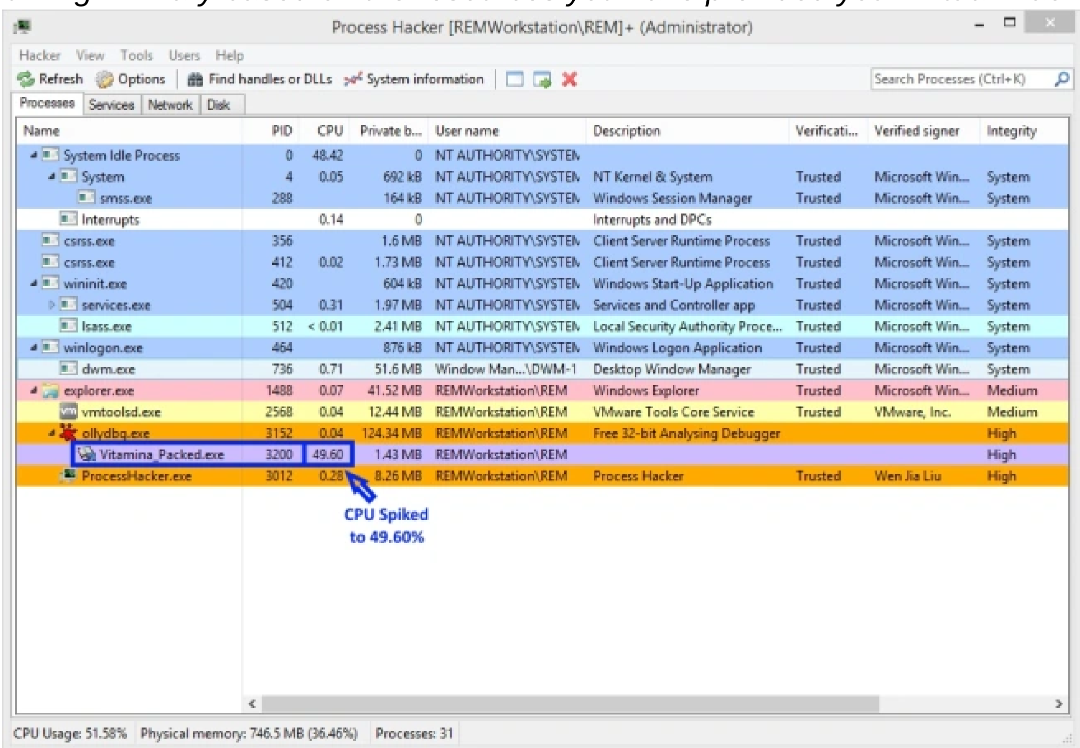


Figure 10: Process Hacker – CPU Utilization after OllyExt bypass

This is how we know that we have bypassed the initial debugger check and that the malware is now working hard to unpack the malicious code.

Anti-Analysis #2: Sandbox Check [GetCursorPos]

With the first debugger check bypassed, when I allow the malware to fully execute within the debugger, it ultimately results in an “Access violation when reading (FFFFFFFF) – application was unable to process exception” error. What triggered this error? In trying to find the answer to this question, I came across the logic shown in Figure 11.

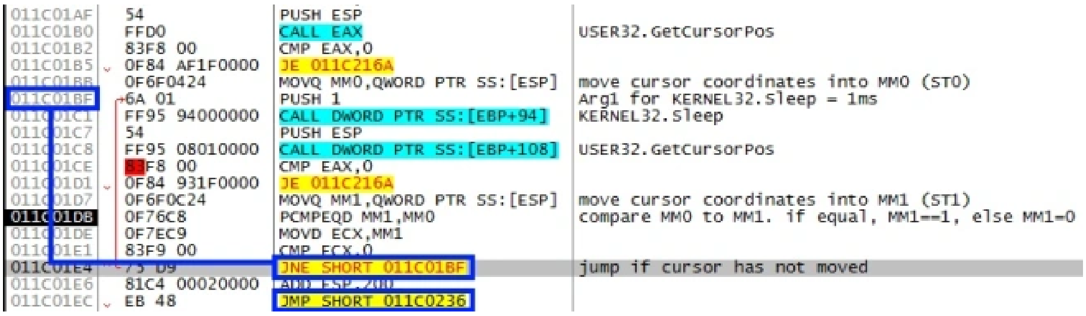


Figure 11: Loop that detects cursor movement

What we see in Figure 11 is an initial CALL to USER32.GetCurosorPos (at 0x011C01B0) that retrieves the current coordinates for the cursor. These coordinates are then stored within the MM0 (MMX) Register. Execution then enters a loop where KERNEL32.Sleep is called with a 1ms sleep time and then the current cursor coordinates are retrieved again. The new coordinates are then compared to the initial coordinates and, if they equal, execution jumps to the top of the loop (0x011C01BF).

What this means is, if the cursor does not move, the malware will execute this loop indefinitely. This is a trick used by malware authors to thwart automated analysis via sandbox where mouse movements wont occur unless configurations are implemented that can simulate such activity. Since I am manually analyzing this sample, cursor movements are detected, thus execution breaks out of the loop and the jump to 0x11C0236 is made.

Anti-Analysis #3: Debugger Check [PEB.NtGlobalFlag]

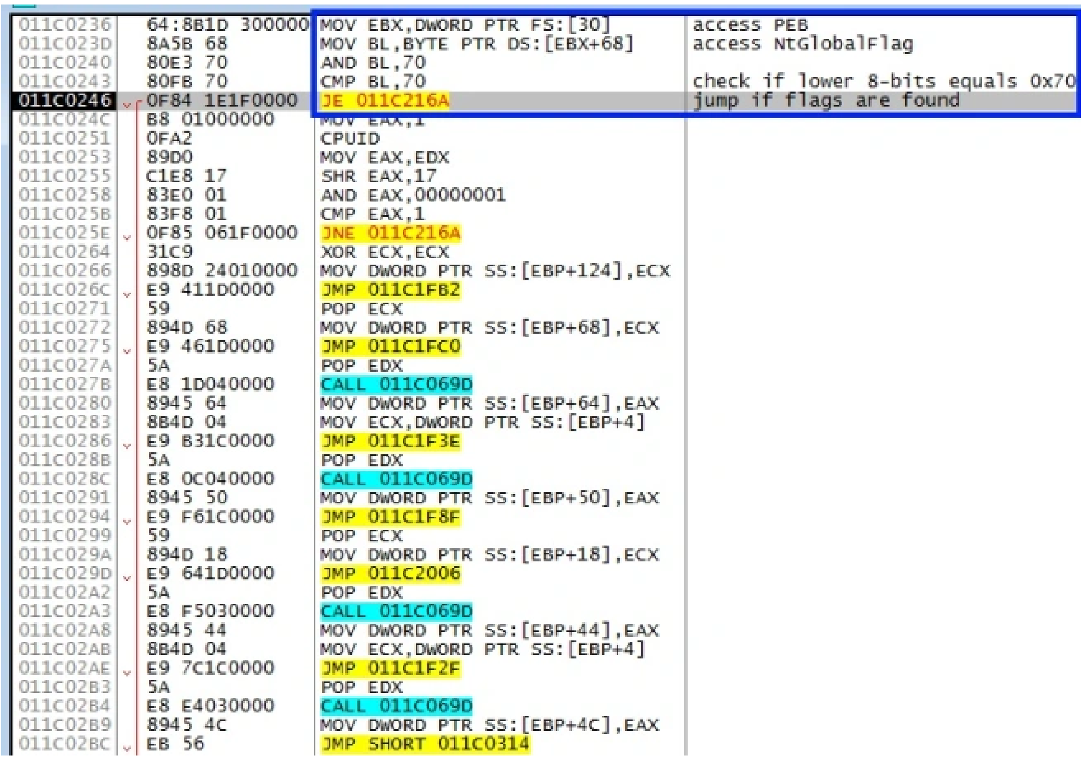


Figure 12: Check NtGlobalFlag within the PEB

This brings us to the instruction "**MOV EBX, DWORD PTR FS:[30]**" shown in Figure 12, which places the address of the PEB into the EBX register. The second instruction ("**MOV BL, BYTE PTR DS:[EBX+68]**") places the value stored at the 0x68 offset within the PEB Structure into the lower 8-bits of the EBX register. Looking at the PEB Structure (https://www.aldeid.com/wiki/PEB-Process-Environment-Block), we see that this value represents the NtGlobalFlag.

To manually verify this value, simply go back into the PEB within OllyDBG (CTRL+G –> "FS:[30]") and locate the NtGlobalFlag value.

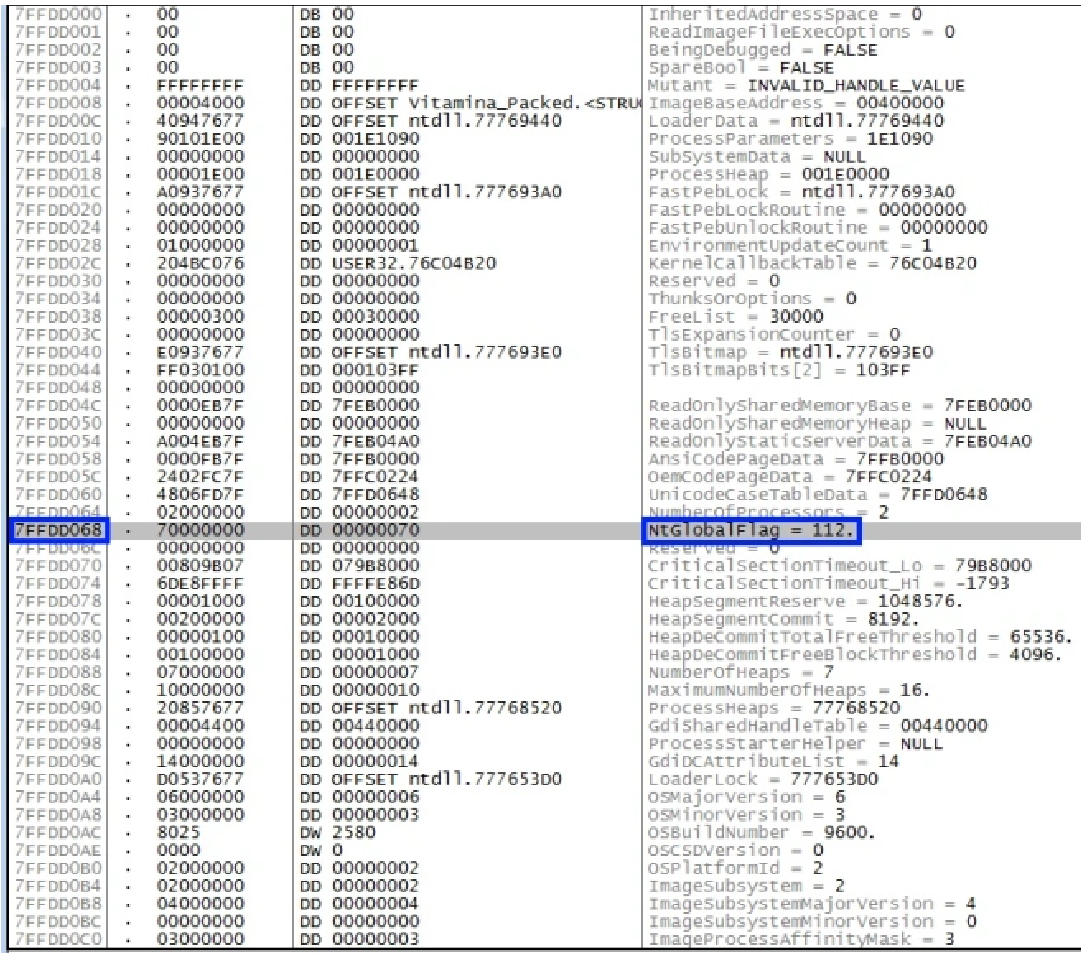


Figure 13: NtGlobalFlag flag within the PEB

In my PEB (Figure 13), we see that my NtGlobalFlag value is set to the decimal value 112 (0x70 in hex), which tells us that the flags FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK, and FLG_HEAP_VALIDATE_PARAMETERS have been set. This is an indication for the malware that it is being debugged.

So, when the CMP at 0x011C0243 is executed, the value within BL (NtGlobalFlag) does – indeed – match the hex value 0x70 and, as a result, the jump to 0x011C216A is taken. This routes execution to a RETF instruction which attempts to return execution to an invalid address, resulting in the “Access violation when reading (FFFFFFFF) – application was unable to process exception” error.

In order to bypass this check, we need to go back into OllyDBG’s OllyExt plugin and check the “NtGlobalFlag” checkbox (Figure 14).

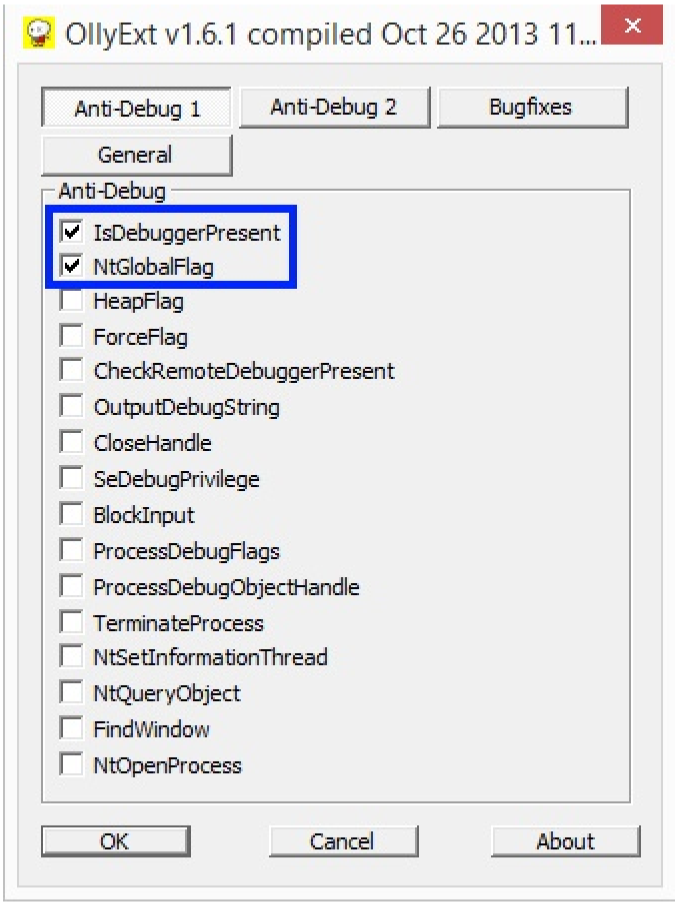


Figure 14: OllyDBG's OllyExt IsDebuggerPresent and NtGlobalFlag bypass

Now, if we restart the executable within OllyDBG and inspect both BeingDebugged and NtGlobalFlag values within the PEB (Figure 15), we see that both values have been set to values that will make the malware think it is not being debugged.

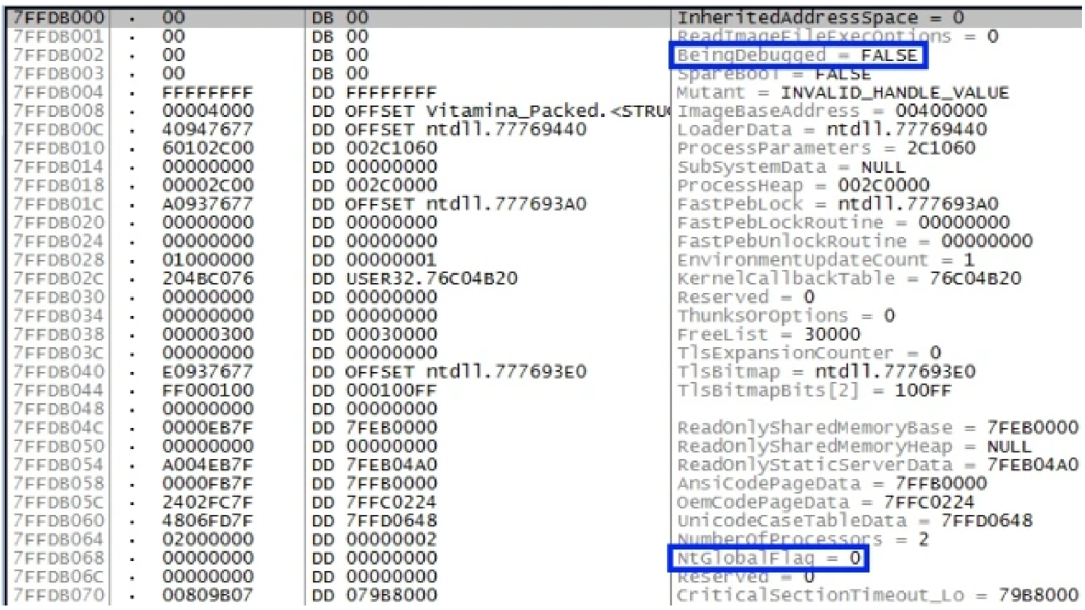


Figure 15: Bypassed BeingDebugged and NtGlobalFlag values within the PEB

With checks for both PEB.BeingDebugged and PEB.NtGobalFlag bypassed, the malware will now fully unpack and execute without error or interruption (as long as you are moving your mouse while it is running).

Obtain the Unpacked Executable

Now that we have bypassed all anti-analysis measures, we can focus on obtaining the unpacked executable. During dynamic analysis, I found that the sample created a subprocess with the same name and command line path as itself. Typically, when you see this, it is an indication the process hollowing (http://www.autosectools.com/Process-Hollowing.pdf) might be taking place.

Unfortunately, this packer does a good job of disguising which functions it is calling and when it is calling them. So, I had to set breakpoints **within the functions of the legitimate libraries** typically used in the process hollowing process. While I wont be able to trigger a break before the function is called, I will be able to halt execution at the first instruction within the function itself. Six of one, half a dozen of the other, as they say.

I can set this breakpoint within OllyDBG by pressing CTRL+G, typing in the name of the function that you want to set the breakpoint within (eg. "ntdll.NtResumeThread"), then clicking the "Follow expression" button. This will bring you to the first instruction of the specified function. It is here that you need to set your breakpoint.

So, what functions does this sample use to perform process hollowing?

1. Kernel32.CreateProcessW (https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx) – Launches an instance of itself as a subprocess in a suspended state.
2. ntdll.NtUnmapViewOfSection (https://msdn.microsoft.com/en-us/library/windows/hardware/ff567119(v=vs.85).aspx) – Hollows out the suspended subprocess.
3. ntdll.NtAllocateVirtualMemory (https://msdn.microsoft.com/en-us/library/windows/hardware/ff566416(v=vs.85).aspx) – Allocates memory within the suspended subprocess.
4. ntdll.NtWriteVirtualMemory – Writes the unpacked malicious code/data into the suspended subprocess.
5. ntdll.NtGetContextThread – Obtain information about the main thread within the suspended subprocess.
6. ntdll.NtSetContextThread – Set the new entry point of the newly inserted malicious code.
7. ntdll.NtResumeThread – Tell the suspended subprocess that it can now begin executing (starting at the newly defined entry point).

I was *really* hoping that there was going to only be a single CALL to NtWriteVirtualMemory that would write the contents of a single buffer (the whole unpacked malicious exe) into the suspended subprocess. Had this been the case, I could have dumped the contents of said buffer out to disk and it likely would have been a fully functional unpacked executable… I was not so lucky. In this sample, There were 6 or 7 CALLs to NtWriteVirtualMemory, which built the contents of the suspended subprocess in sections.

When you come across this scenario, the three key function CALLs that you need to focus on are:

CreateProcessW

When we hit the breakpoint within CreateProcessW, its arguments on the stack will appear as they do in Figure 16 (*Note CREATE_SUSPENDED Creation Flag*):

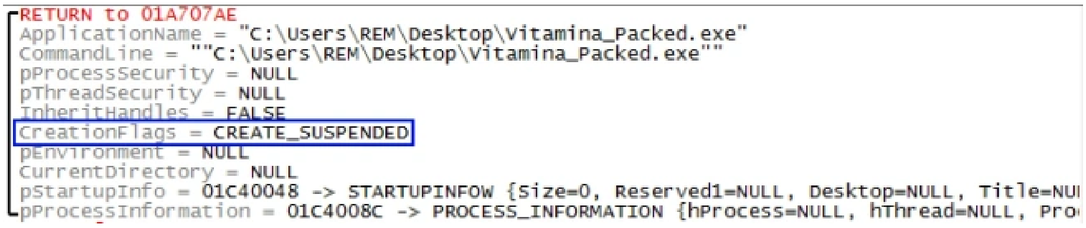


Figure 16: CreateProcessW arguments on the stack

What we are looking for is the Process ID that is created as a result of this function's execution. Sitting at our breakpoint, if we allow the function to execute until it returns to the main thread, we find that the following values have been placed into the address specified by the pProcessInformation argument (Figure 17):

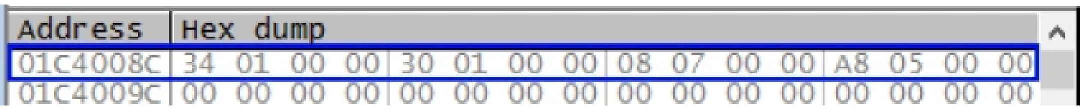


Figure 17: Populated PROCESS_INFORMATION Structure in memory

To visualize this data better, we can instruct OllyDBG to parse it as a PROCESS_INFORMATION Structure by:

1. Right-click on the starting address (0x1C4008C).
2. Select "Decode as structure".
3. Select "PROCESS_INFORMATION" in the dropdown menu.
4. Click Ok.

OllyDBG will then present this data to you, like so (Figure 18):

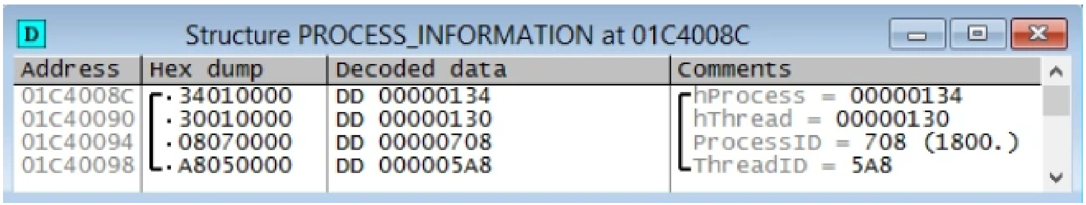


Figure 18: Populated PROCESS_INFORMATION Structure

The Process ID specified, 1800 (or 0x708 in hex), is the one that represents the suspended subprocess that will be the target for the process hollowing.

NtSetContextThread

Since there isn't a single buffer that we could reference that contains a fully unpacked executable that we can dump from memory, we'll need to allow the malware to finish writing to the suspended subprocess. When that has been accomplished, the suspended subprocess will need to know where to begin execution. This new Entry Point is defined via CALL to NtSetContextThread.

Similar to how we handled CreateProcessW, we'll need to do the following:

1. Allow execution to hit the breakpoint within NtSetContextThread.
2. Right click on the second argument passed to NtSetContextThread (on the stack) and select "Decode as structure".
3. Select "Context" from the drop down menu and click Ok.

The CONTEXT Structure appears as follows (Figure 19):

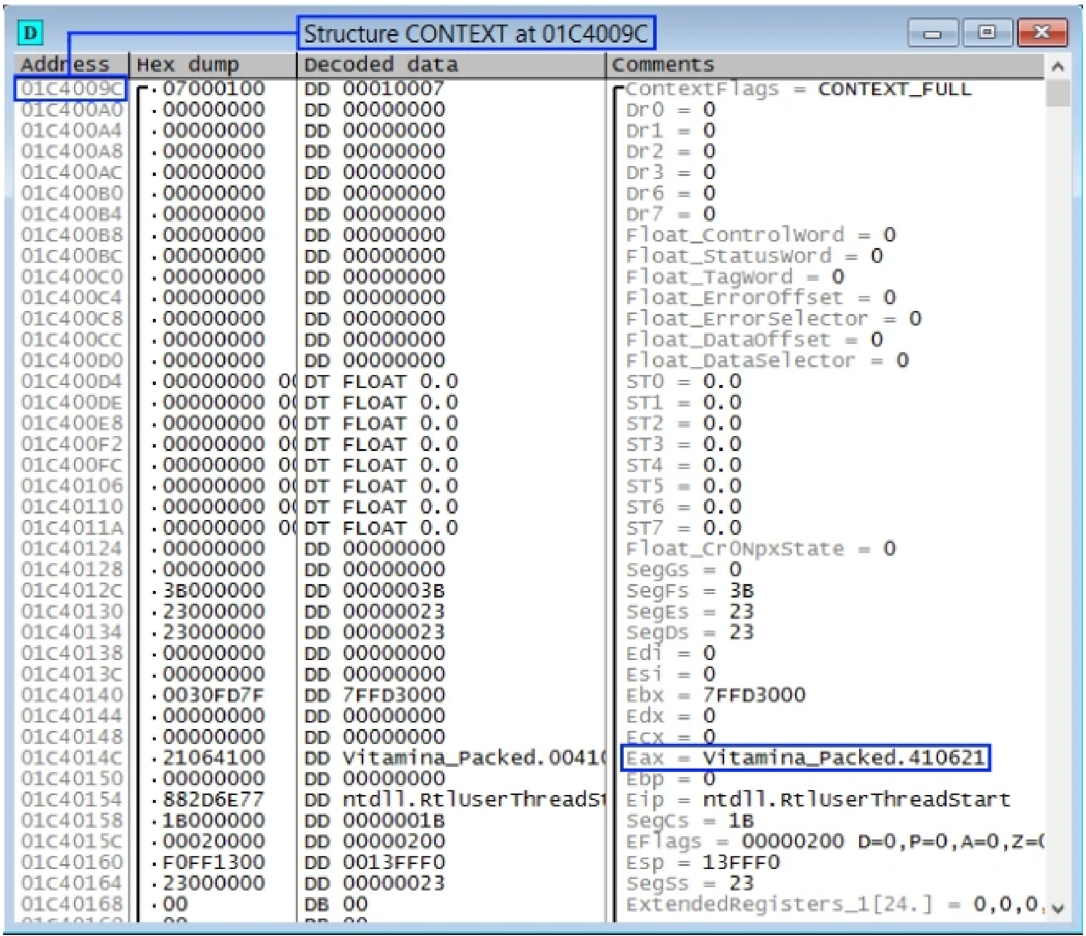


Figure 19: Populated CONTEXT Structure. Highlighting key Entry Point value

Highlighted in the image above is the “Eax” value within the CONTEXT Structure that the new Entry Point gets set to within the target subprocess by NtSetContextThread. We will need this value, 0x410621, in order to produce a fully functional unpacked executable in the next step.

NtResumeThread

Last, but certainly not least, is the CALL to NtResumeThread. If we allow execution to hit the breakpoint within this function, we’ll know that the packed binary has finished writing its unpacked code into the target subprocess.

It is important that you don’t allow the complete execution of this function. We need the target subprocess to remain in a suspended state in order for us to properly dump it to disk.

While sitting at this breakpoint, we need to perform the following*:

** I chose to dump the process using OllyDBG but there are multiple ways this could have been done (eg. Using Scylla).*

1. Open a second instance of OllyDBG and attach it to the suspended subprocess (File –> Attach) that has the Process ID of 0x708 (PID identified in CreateProcessW section).
2. Once loaded, go to the Entry Point that we identified in the NtSetContextThread section by pressing CTRL+G, entering in the address (0x410621), and then clicking “Follow expression”.
3. With this address selected in the CPU window, right-click on it and select “New origin here”.

4. Use the OllyDumpEx plugin to dump the process to disk. To do this, select from the toolbar: Plugins –> OllyDumpEx –> Dump process. You may be presented with an error that states "Cannot Get Debuggee Filename". Just click Ok.
5. You will be presented with the pop-up window shown in Figure 20. Click the "Get EIP as OEP" button to ensure that you have the proper Entry Point set, click the "Dump" button, and save the file to a path and filename of your choosing (eg. "Vitamina_Packed_dump.exe"):

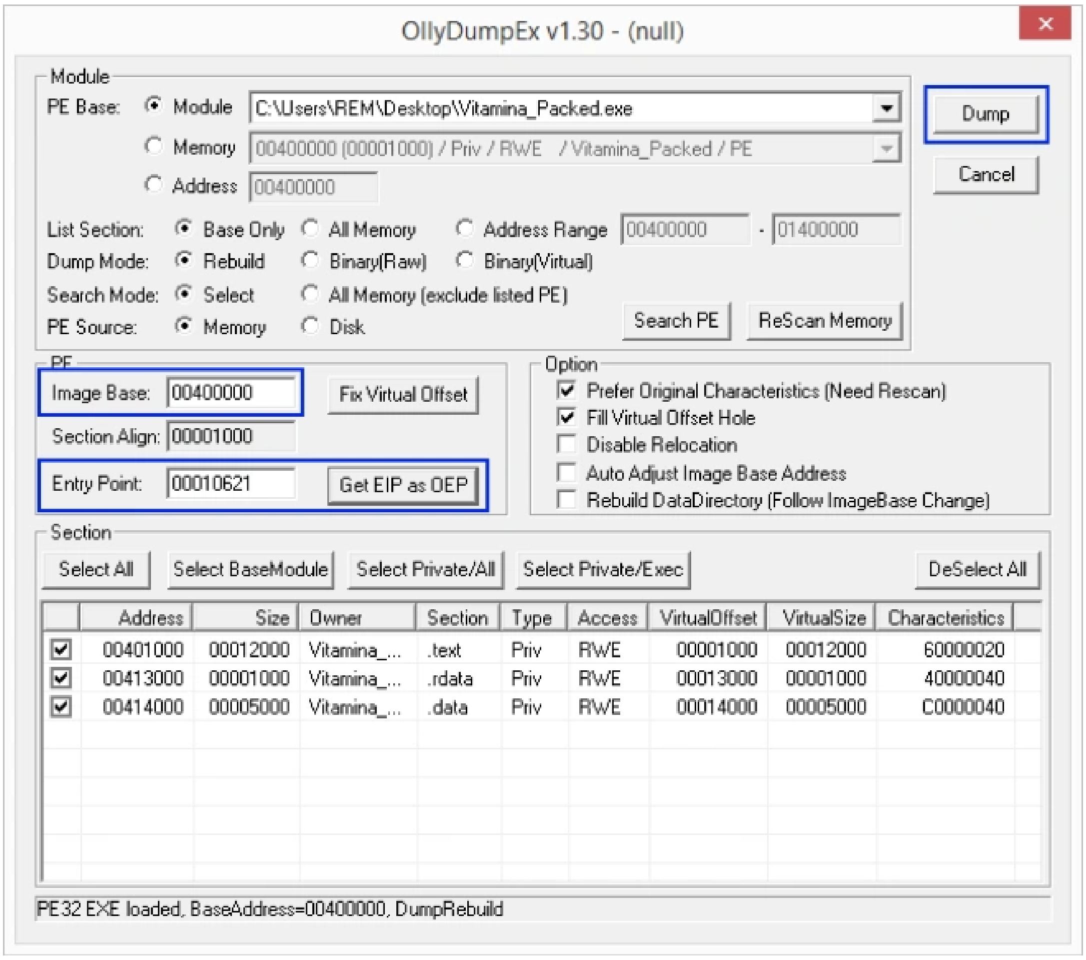


Figure 20: Dumping suspended process to disk using OllyDBG's OllyDumpEx plugin

Fix the Imports

Because we dumped this process from memory, the resolutions within the Import Address Table (IAT) are all messed up. The final step that we need to take, in order to make this dumped file a fully functional unpacked executable, is to fix the imports of this dumped file using Scylla Imports Reconstructor.

With the subprocess still running in a suspended state (Figure 21):

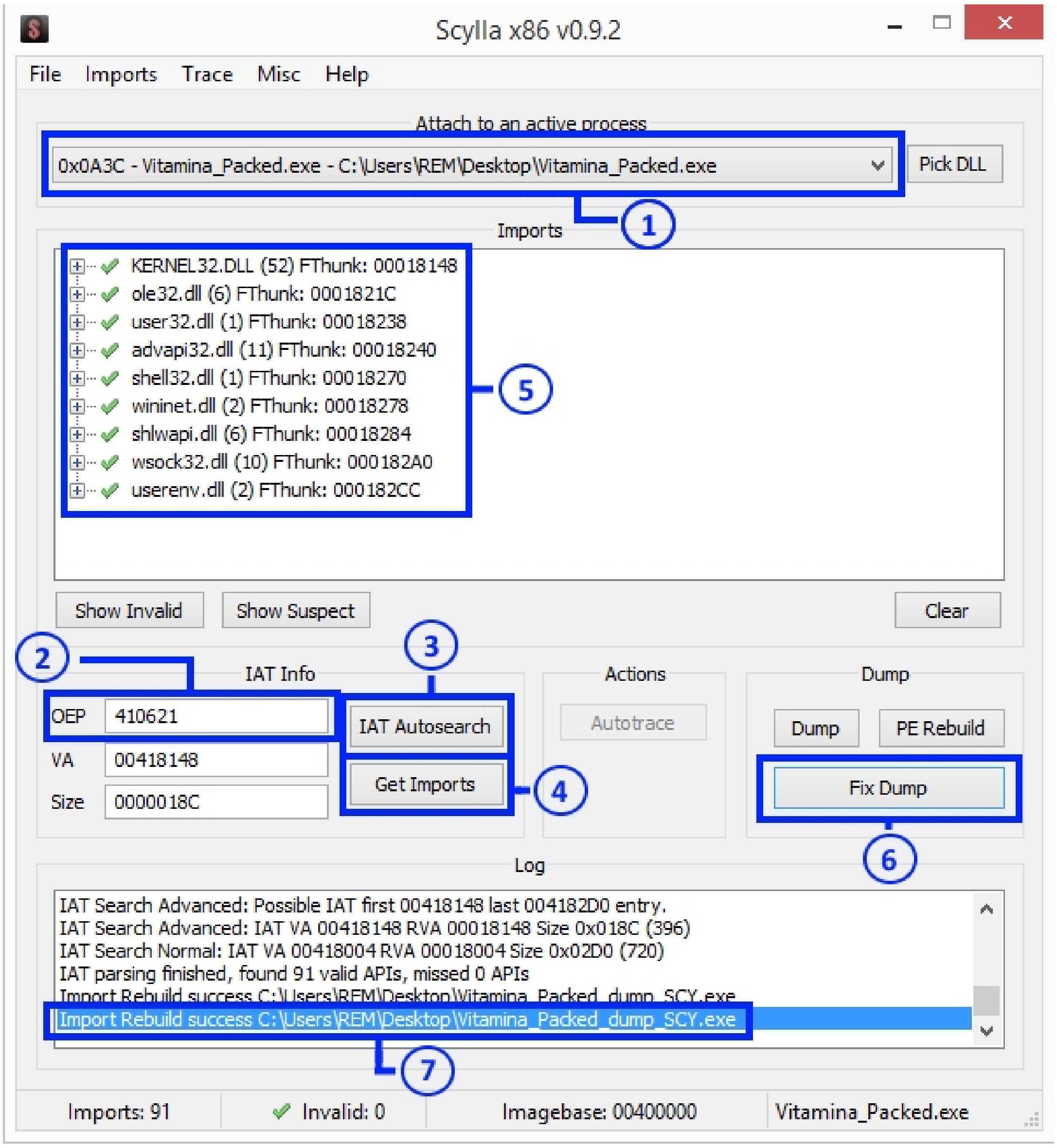


Figure 21: Fixing corrupt Import Address Table of dumped process using Scylla

1. Open Scylla and attach it to the suspended subprocess using the drop down menu at the top.
2. Enter the Entry Point address that we identified in the NtSetContextThread section (410621) into the “OEP” field.
3. Click the “IAT Autosearch” button.
4. Click the “Get Imports” button.
5. This should populate the Imports section with a list of DLLs with green check marks. If you see a bunch of red X’s, something is wrong and you’ll need to fix this issue before you can move on to the next step.
6. Click the “Fix Dump” button and choose the path and filename of the unpacked non-functional executable that you dumped from OllyDBG.
7. This should produce a new executable named after the original dumped file but with “_SCY” appended to it. In my instance, this was “Vitamina_Packed_dump_SCY.exe”.

This new SCY file should be your fully functional unpacked executable. You can now take this unpacked executable and perform static/dynamic/code level analysis without any issues; assuming the sample wasn’t packed multiple times.

Resources

- http://waleedassar.blogspot.com/2012/03/visual-basic-malware-part-1.html (http://waleedassar.blogspot.com/2012/03/visual-basic-malware-part-1.html)
- https://www.vb-decompiler.org/pcode_decompiling.htm (https://www.vb-decompiler.org/pcode_decompiling.htm)
- http://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf (http://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf)
- https://www.aldeid.com/wiki/PEB-Process-Environment-Block (https://www.aldeid.com/wiki/PEB-Process-Environment-Block)
- https://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged (https://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged)
- https://www.aldeid.com/wiki/PEB-Process-Environment-Block/NtGlobalFlag (https://www.aldeid.com/wiki/PEB-Process-Environment-Block/NtGlobalFlag)
- http://www.autosectools.com/Process-Hollowing.pdf (http://www.autosectools.com/Process-Hollowing.pdf)
- https://forum.tuts4you.com/files/file/576-scylla-imports-reconstruction/ (https://forum.tuts4you.com/files/file/576-scylla-imports-reconstruction/)
- https://vimeo.com/204733748 (https://vimeo.com/204733748)