Unpacking Visual Basic Packers – IcedID

zero2auto.com/2020/06/22/unpacking-visual-basic-packers/

0verfl0wz2a

June 22, 2020

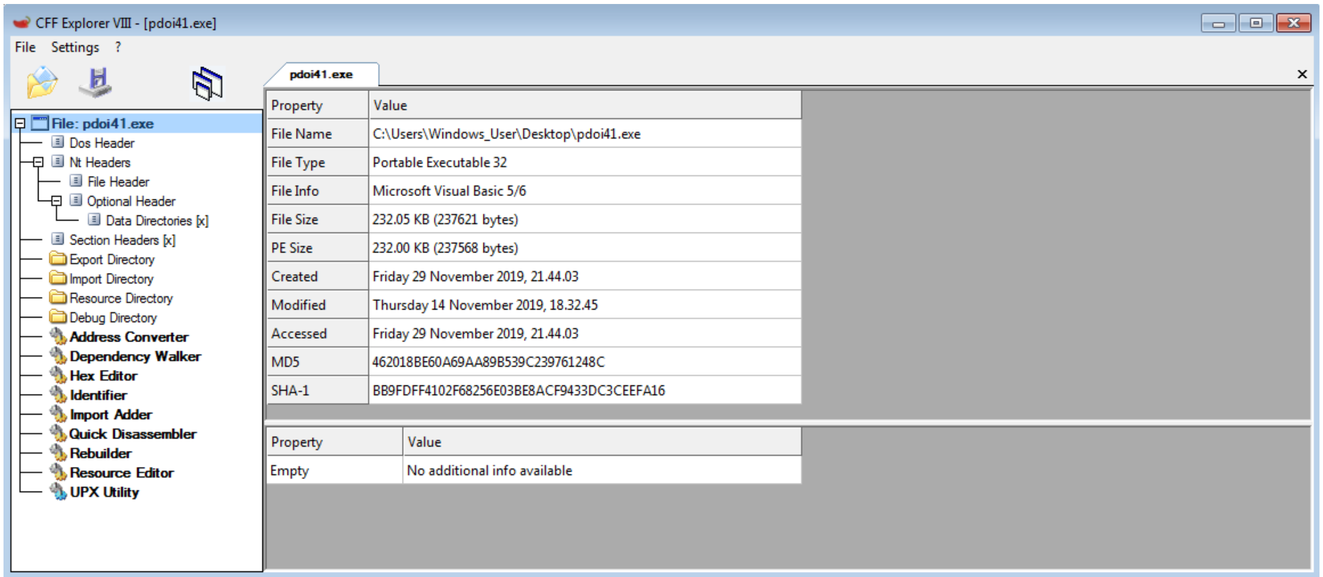
Despite the fact that VisualBasic is an age-old programming language, it is still being used to develop malicious software – specifically packers – to this day. As a result, you will often encounter VisualBasic based packers used in a lot of “script-kiddie” malware, such as keyloggers and remote access tools being sold on forums, and more recently, IcedID! For those of you not aware, IcedID (AKA BokBot) is a banking trojan that has been around for a couple years now, which was quite infamous in the malware analysis community due to it’s novelty process injection technique of API hooking certain calls to execute code inside of a spawned svchost.exe process. If you haven’t heard of this injection technique before, don’t worry! We will be covering it in **Section 3** of the Zero2Automated course, along with other injection techniques – but, if you would like to check out a few articles about IcedID to get some context before reading this one, feel free to! Anyway, let’s jump straight into the analysis!

Indicator Of Compromise:

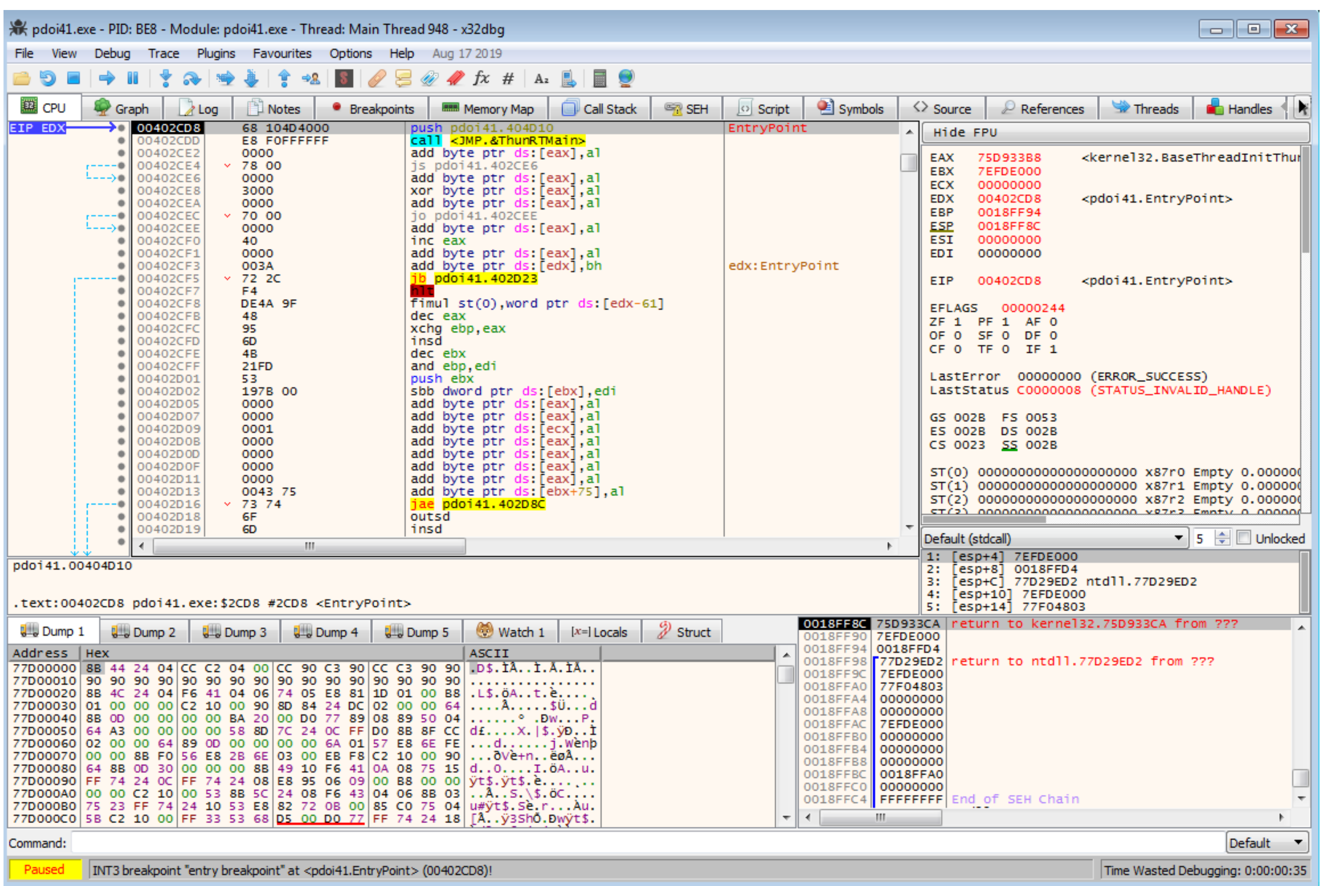
MD5 Hash of Packed Sample: *462018be60a69aa89b539c239761248c*

Initial Analysis:

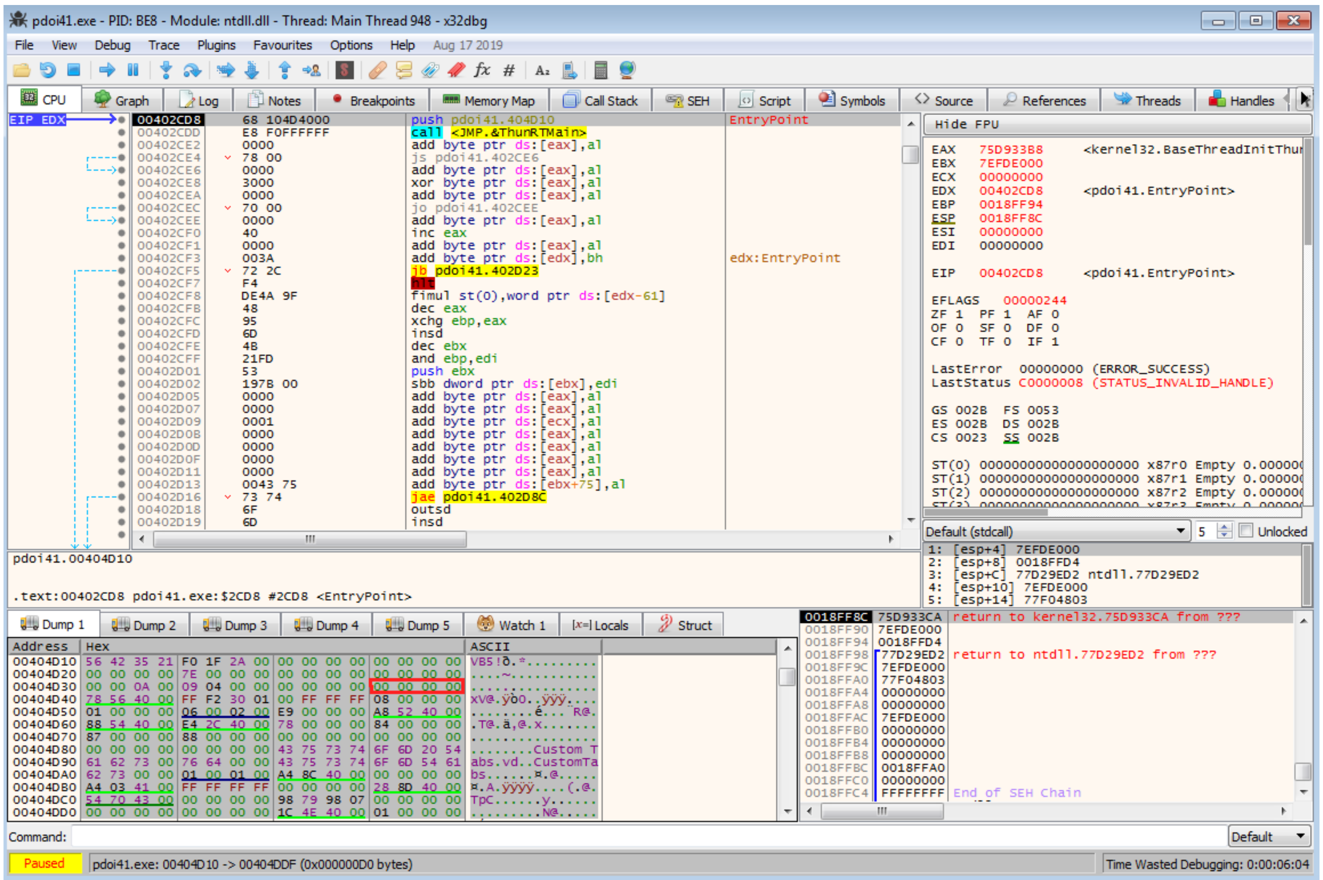
So, while I do know that this sample is packed as I have looked at it before, let’s approach it as if we had no idea what it was. First things first, we want to open it up in a PE analysis tool, which in this case I will be using **CFF Explorer**. One of the good things about **CFF Explorer** is the fact that as soon as we open up the malicious executable in it, we can see the **File Info**, which is *Microsoft Visual Basic 5/6,* so automatically we know that VB was used to create the file.



As we don't want to spend too long analyzing the executable, let's go ahead and open it up in x32dbg to begin with. The reason we are opening it up in a debugger to begin with rather than IDA Pro is due to the fact that VB is more commonly used to create malware packers than malware itself, so we are saving ourselves some time here. What we see in the debugger can tell us if the executable is packed or not, so let's check that out first. As you can see in the image below, the entry point is a lot different to the entry points of most executables, as all there is is a push instruction and then a call to **ThunRTMain()**.



When we view the pushed address in the dump, you'll notice a few strings such as *VB5!* and *Custom Tabs*. This is actually a structure used to tell **ThunRTMain()** about the program, including where the entry point of the user code actually is. **Here** you can see a table containing information about the different values inside the structure, and the position of them. The value we want to find is referred to as **aSubMain**, as that is the address called by **ThunRTMain()** once everything has been initialized. The area highlighted in red in the memory dump **should** contain the address of **aSubMain**, however it is completely empty! This indicates that there is some obfuscation going on, or the authors have altered the compilation routine. As a result of this, we will have to rely on setting some breakpoints instead to unpack the sample, rather than statically analyze it.

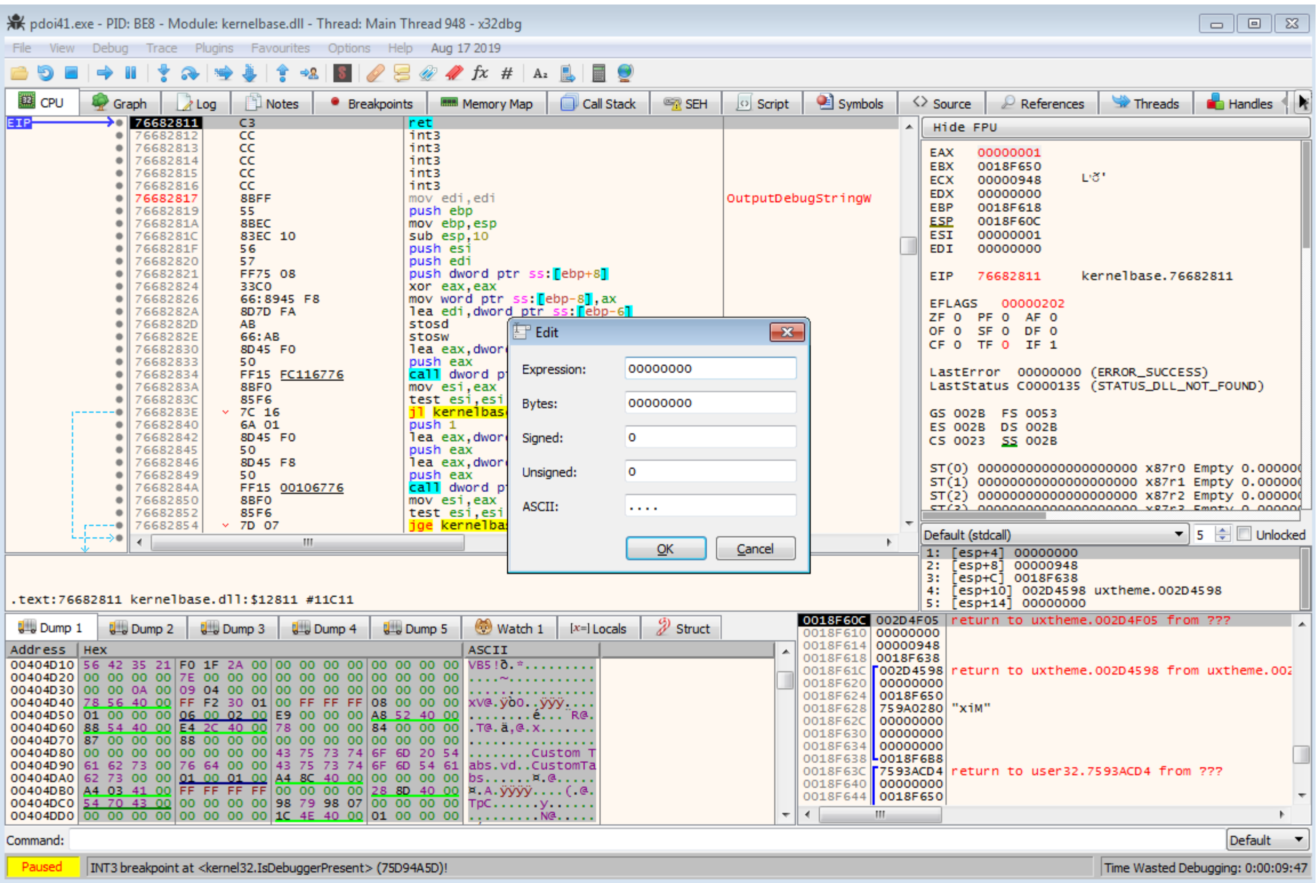


When unpacking VB based packers, I like to put a breakpoint on 4 main API calls:

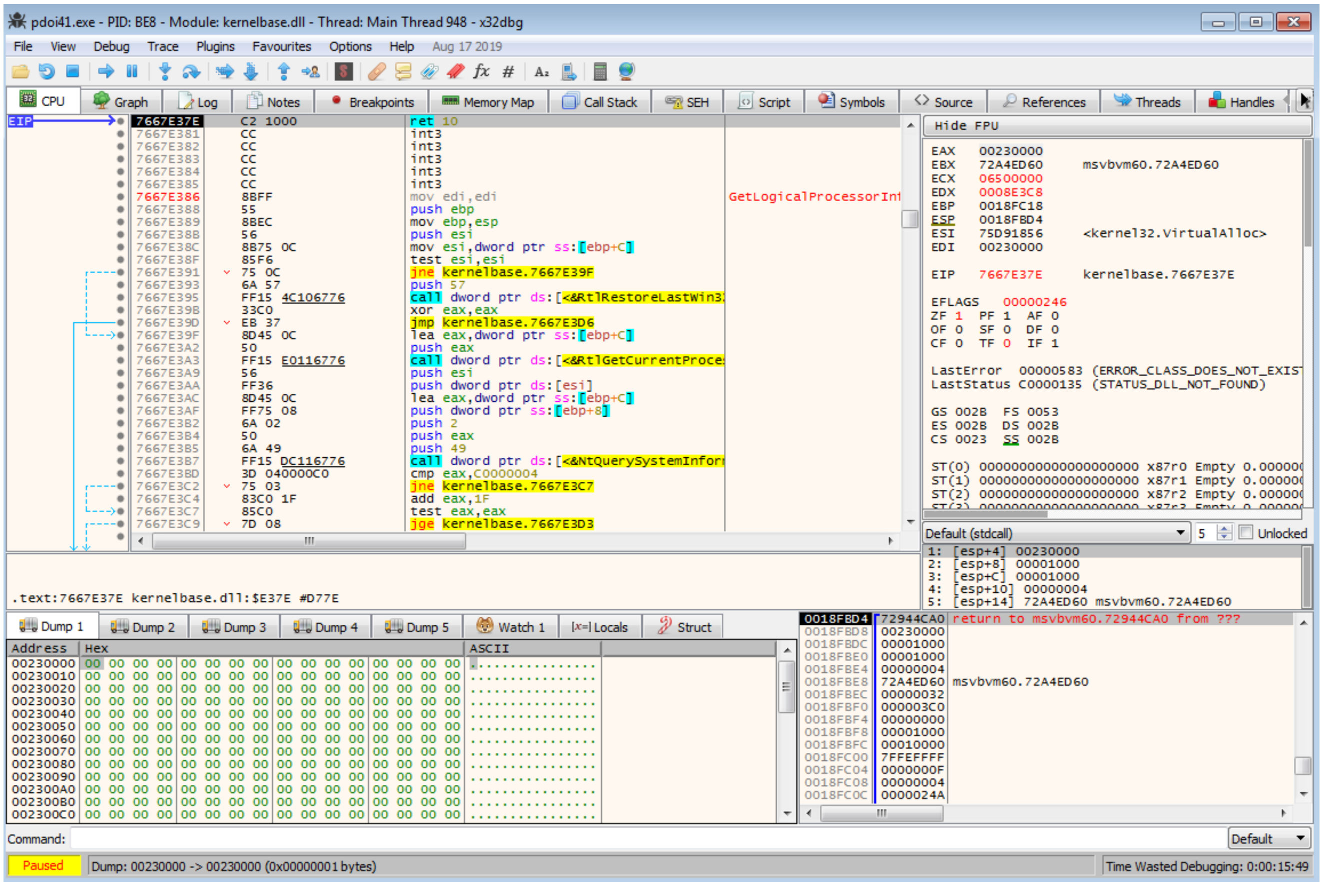
- **VirtualAlloc**
- **VirtualProtect**
- **IsDebuggerPresent**
- **CreateProcessInternalW**

This allows us to; view any allocated regions of memory that may have an executable written into it, prevent any processes from being created, as well as stop the most common anti-debug method from executing, which is quite common in these VB packers. The reason we want to break on it is so we can alter the value to "hide" the debugger from the process. If a debugger is detected by the malware, it might not halt execution as many of you may have thought – it is quite common for the program to continue executing, however it will take a very different path that does nothing. This can lead you to analyze a program for hours, trying to figure out why nothing malicious is actually happening.

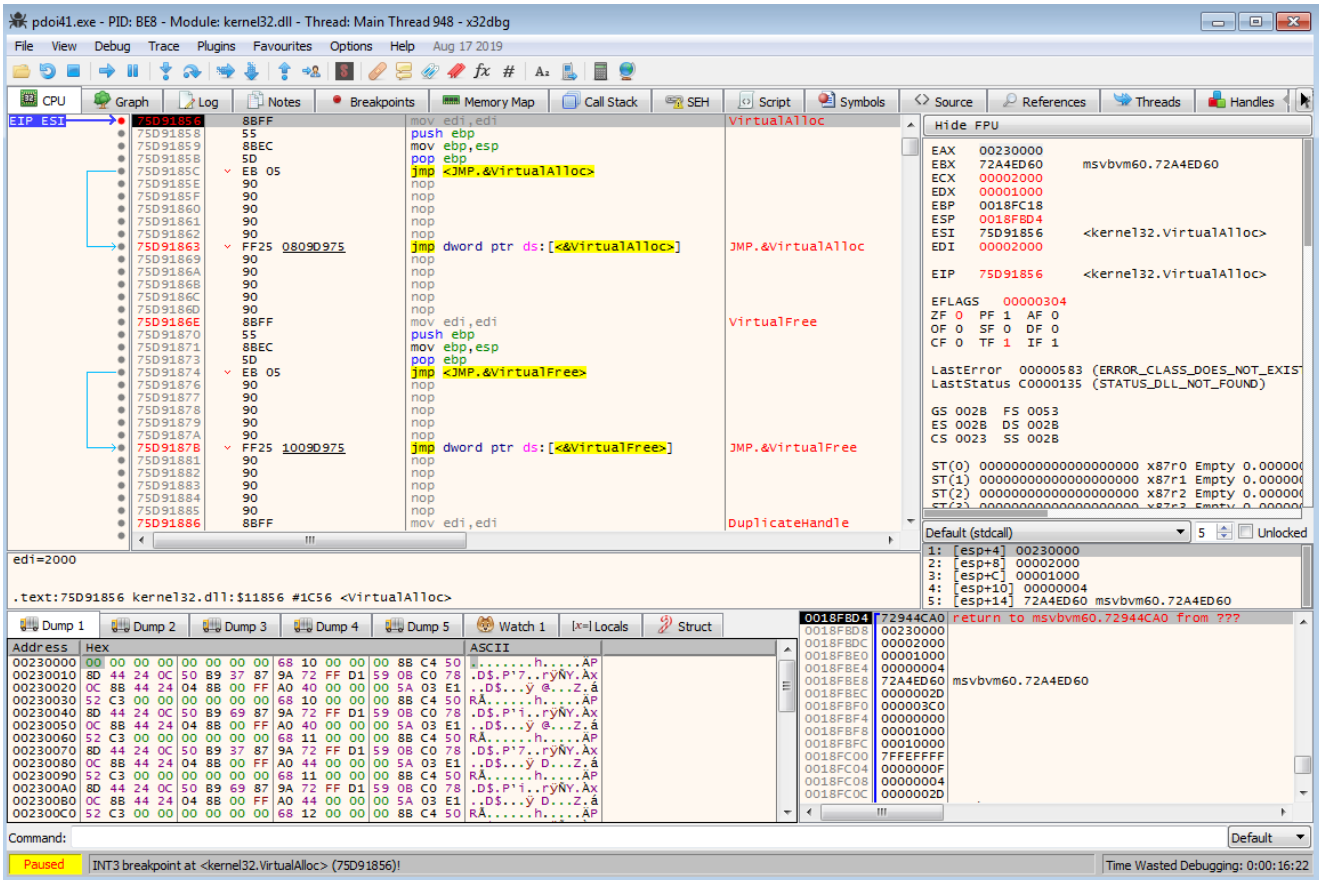
So, putting a breakpoint on these calls and running the debugger, you can see we hit **IsDebuggerPresent** immediately. In order to alter the return value of this call, we want to *Execute To Return* and then alter the value in **EAX** from **1** to **0**. Then go ahead and execute the debugger again, and wait for the next breakpoint to be hit!



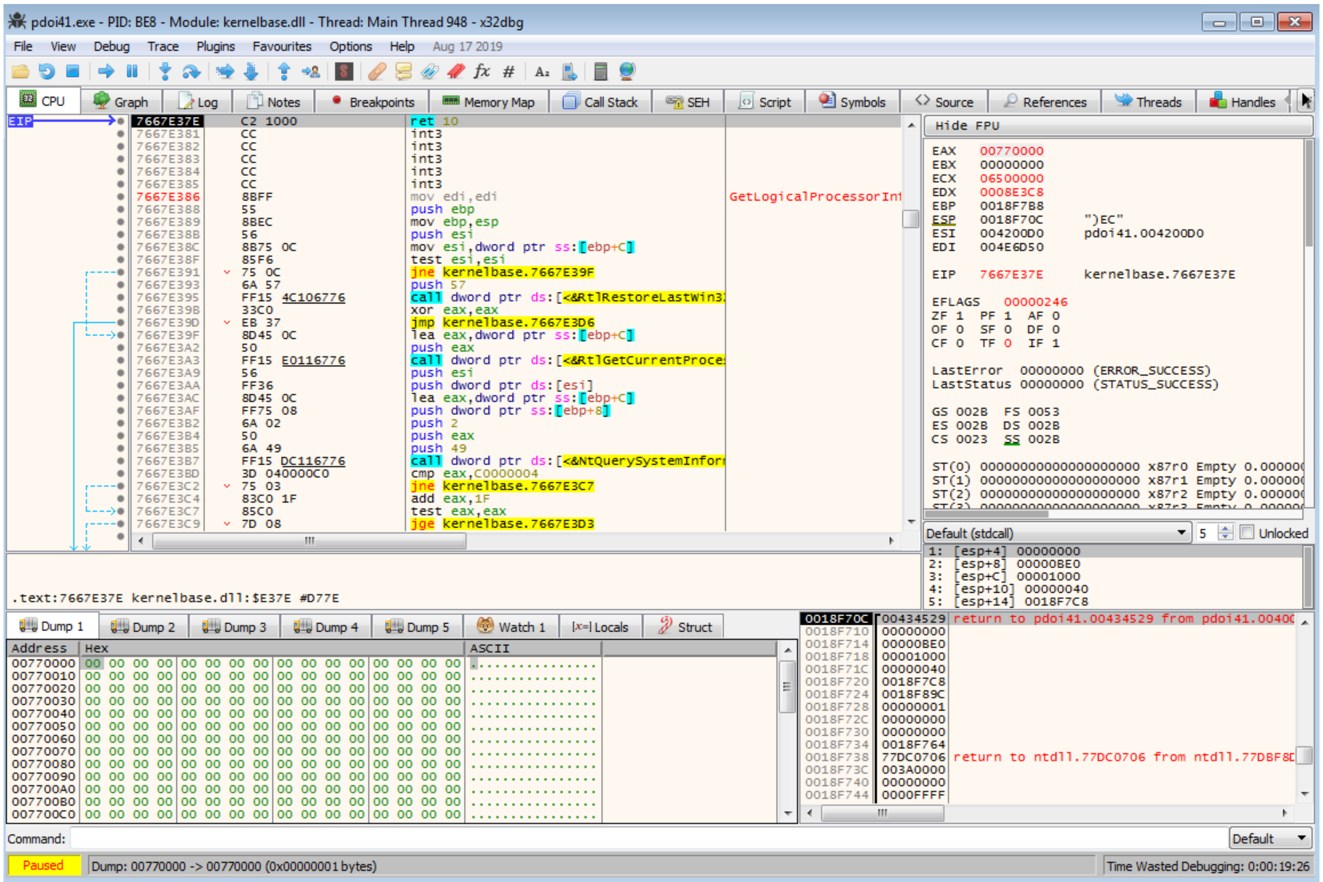
The next breakpoint to be hit is **VirtualAlloc**, so let's execute to return again and follow the value stored in **EAX** in the dump – buuut on the first run you'll notice that there is no option to follow the value in the dump. This is also quite common for some samples – what happens is it first calls **VirtualAlloc** in order to reserve the memory location, and then it will allocate it on the second call. Therefore, go ahead and execute the debugger again and you'll see it breaks on **VirtualAlloc** once more – this time when you execute to return, you'll be able to follow the value in the dump!



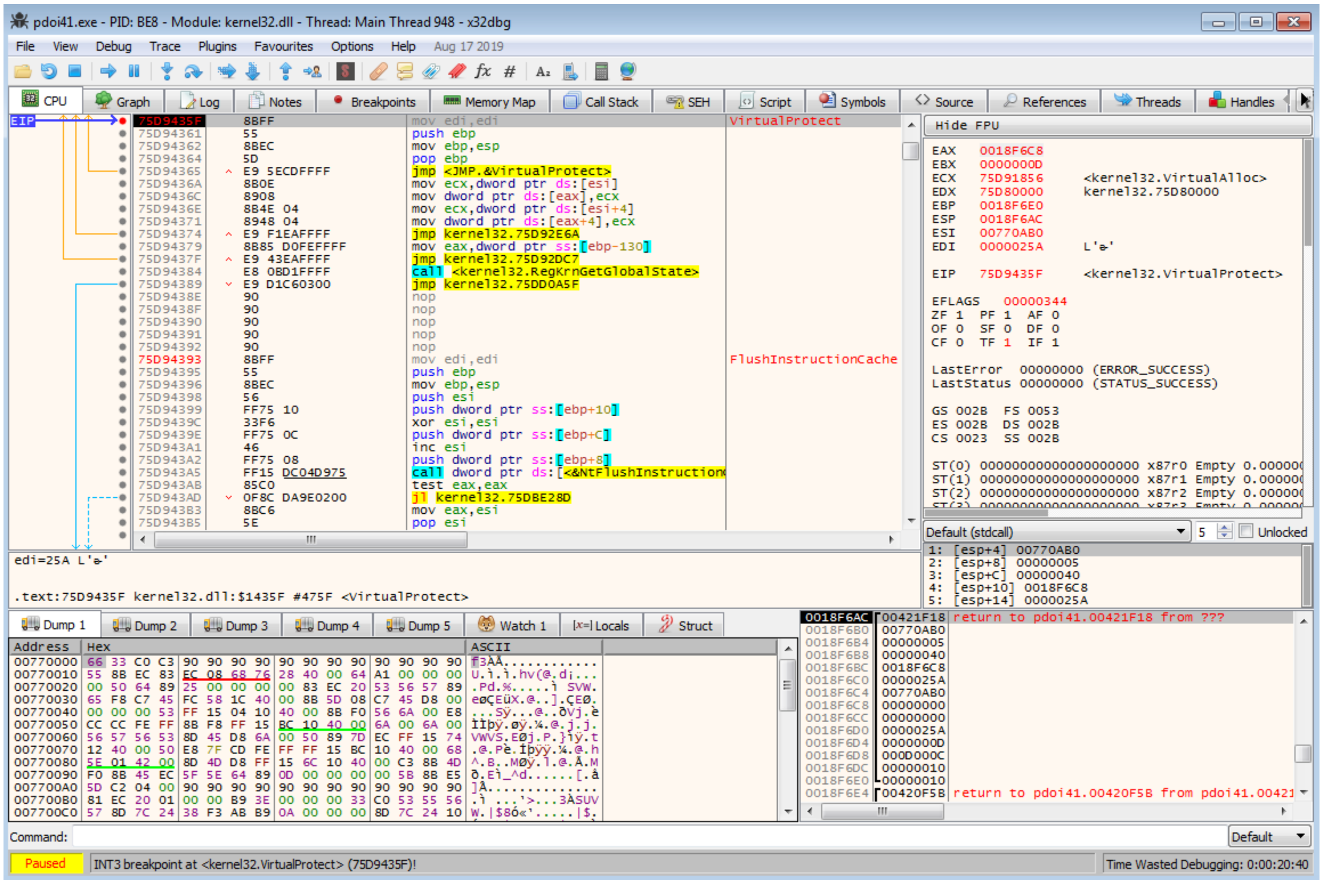
After that last allocate, running the debugger once again, it'll break on **VirtualAlloc**. At this point, you'll notice the previous memory region has been filled in, however it doesn't seem to be very useful and it definitely isn't an executable, so let's continue running the debugger and keep an eye on any allocated regions of memory, until something interesting happens!



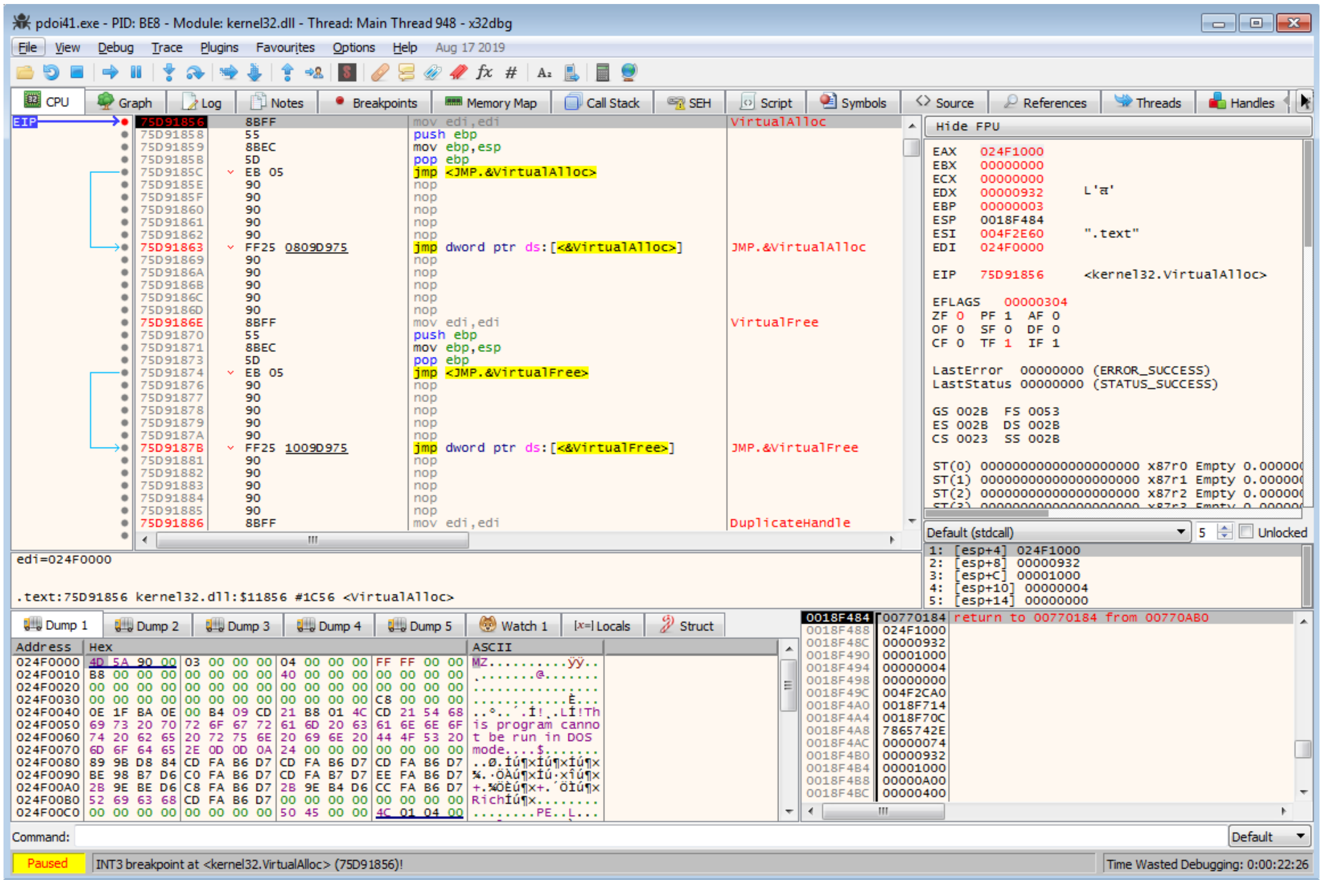
Finally, after ignoring several more useless **VirtualAlloc** and **VirtualProtect** calls, we finally find a new region being allocated at **0x00770000**! Now let’s follow this in the dump and run the debugger once more to see if anything interesting is copied over.

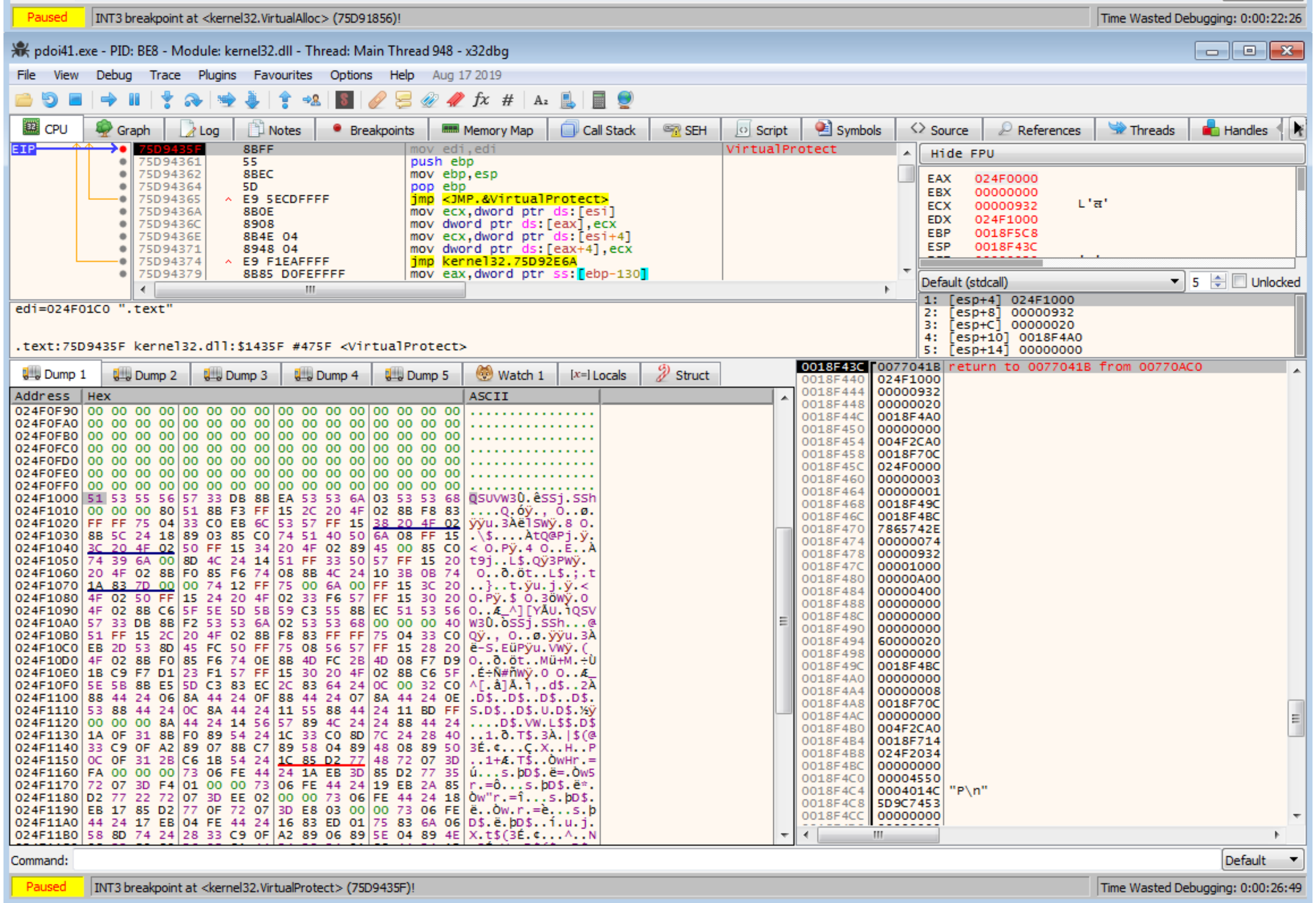


The next breakpoint that is hit is a call to **VirtualProtect**, which is changing the protection of the region of memory just allocated – which has also been filled in with what seems to be shellcode. If you watched the initial video in the section regarding packers, you'll remember that packers often use shellcode to decrypt the executable and overwrite the packer (in memory) with the decrypted executable, so that could be what is happening here. Let's go ahead and run the debugger again, until we see an executable being written to an allocated region of memory!

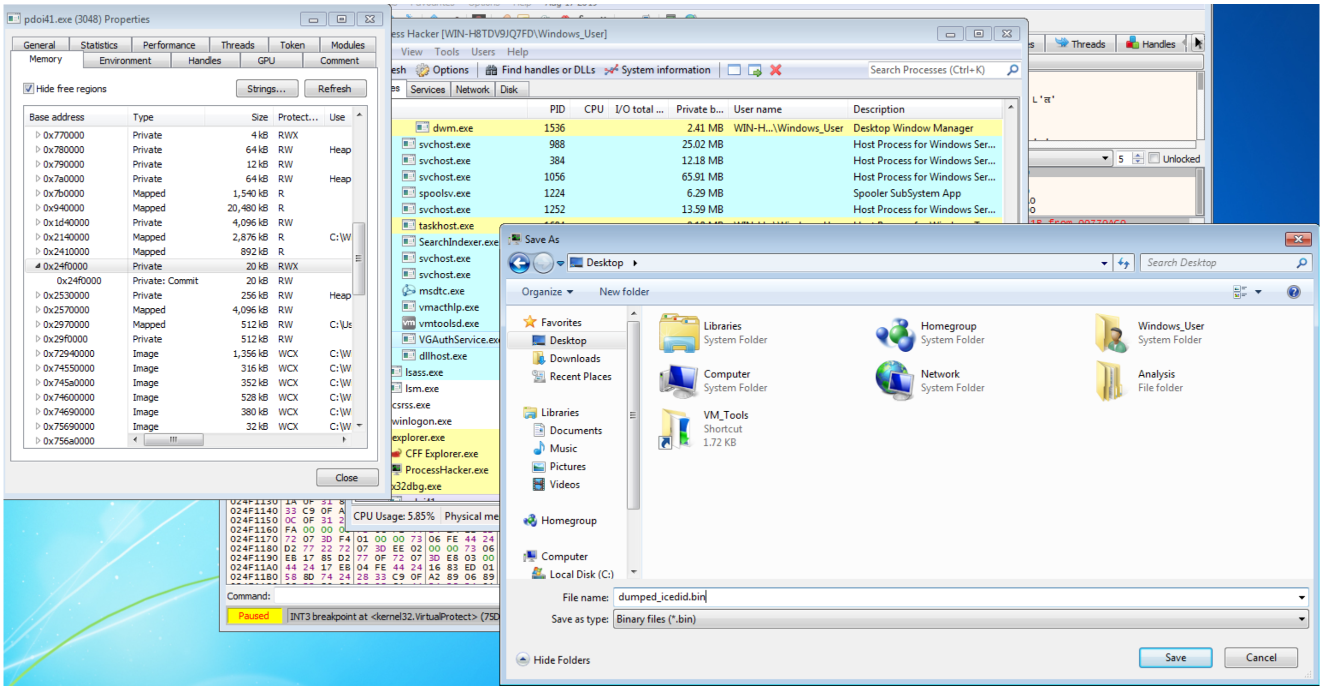


Sure enough, after a few more triggered breakpoints, we reach a call to **VirtualAlloc** that allocates a memory region, which shortly after has an executable written to it! This isn't the whole executable however, if you scroll down you'll notice only the header has been written, so we can't dump it out just yet. The reason why it hasn't been fully written into memory yet is due to the fact that the executable must be mapped into memory in order to execute, so the packer will go ahead and allocate memory at **0x024F1000** and write the **.text** section to it, then allocate memory at **0x024F2000** and write the **.rdata** section to it, and so on. Therefore, let's go ahead and run the debugger until we hit a call to **VirtualProtect**, as this is when the packer begins to change the protection of the different regions of memory in the executable – such as changing the protection on the **.text** section to **RWX** (read-write-execute).

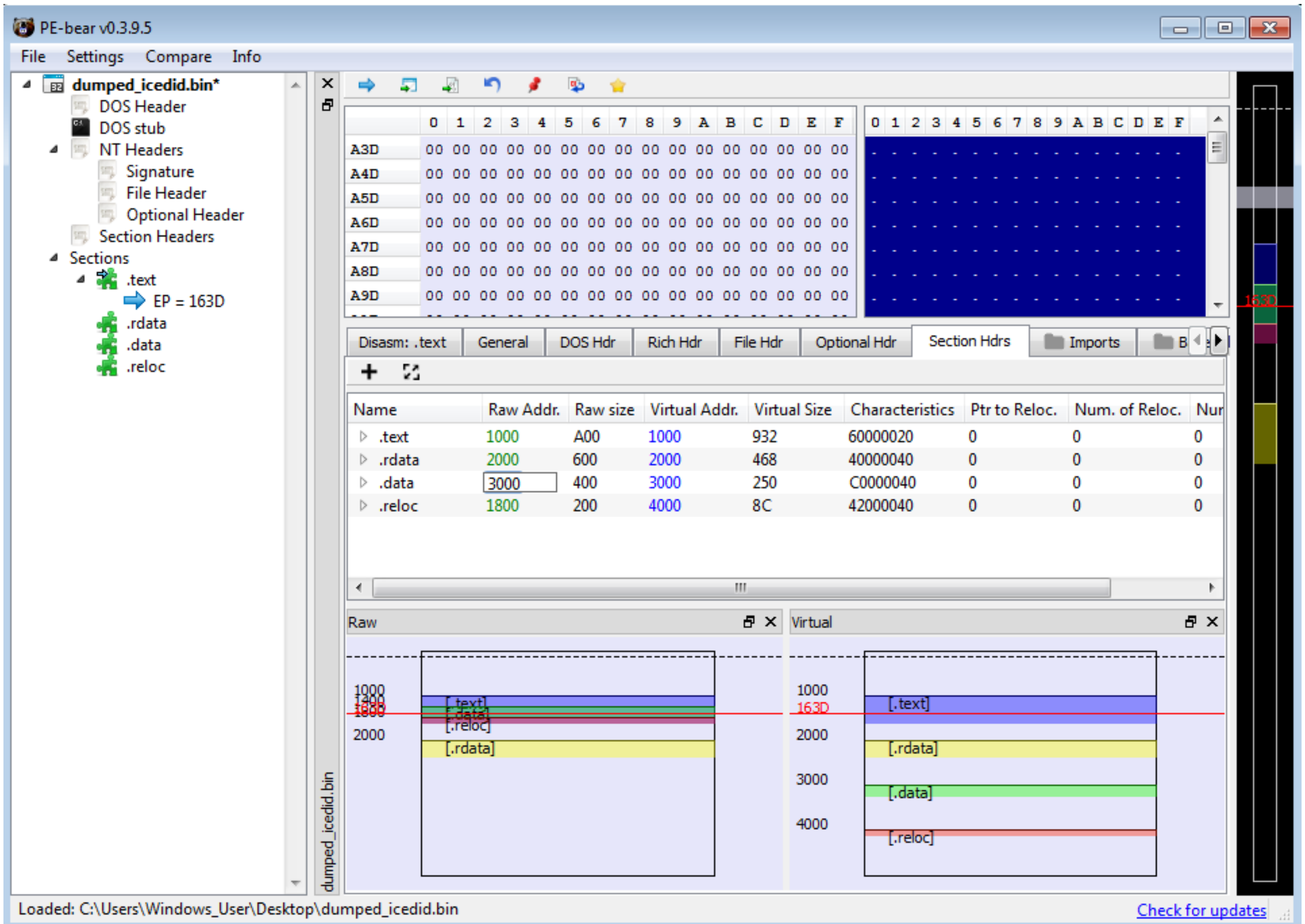




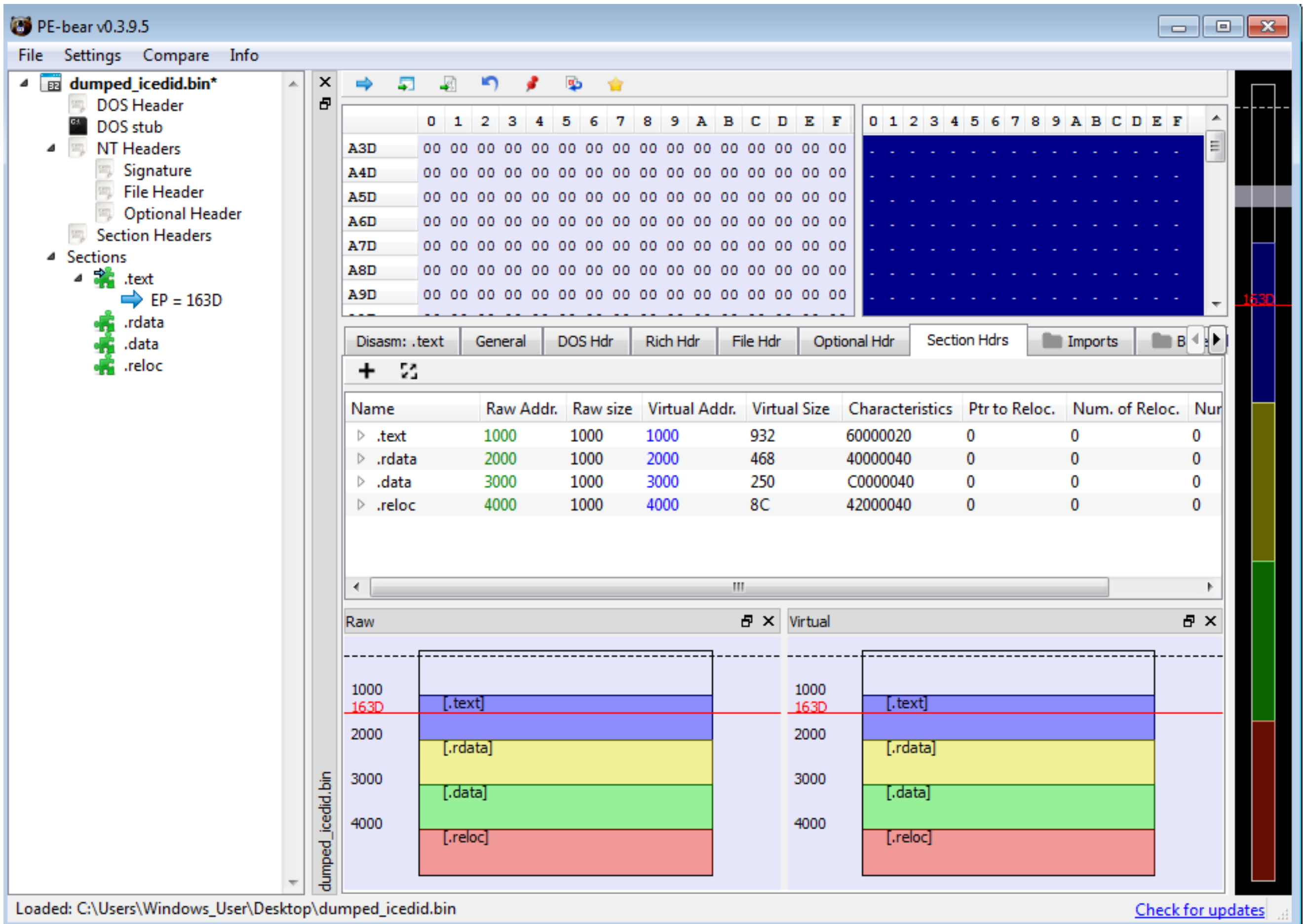
Once we've hit **VirtualProtect**, we can go ahead and dump out the executable from memory, as this is the fully unpacked **IcedID** loader! One of my favourite tools to do this is **Process Hacker 2**, but you can use whatever tool you like, or even use the inbuilt x32dbg memory dump functionality.

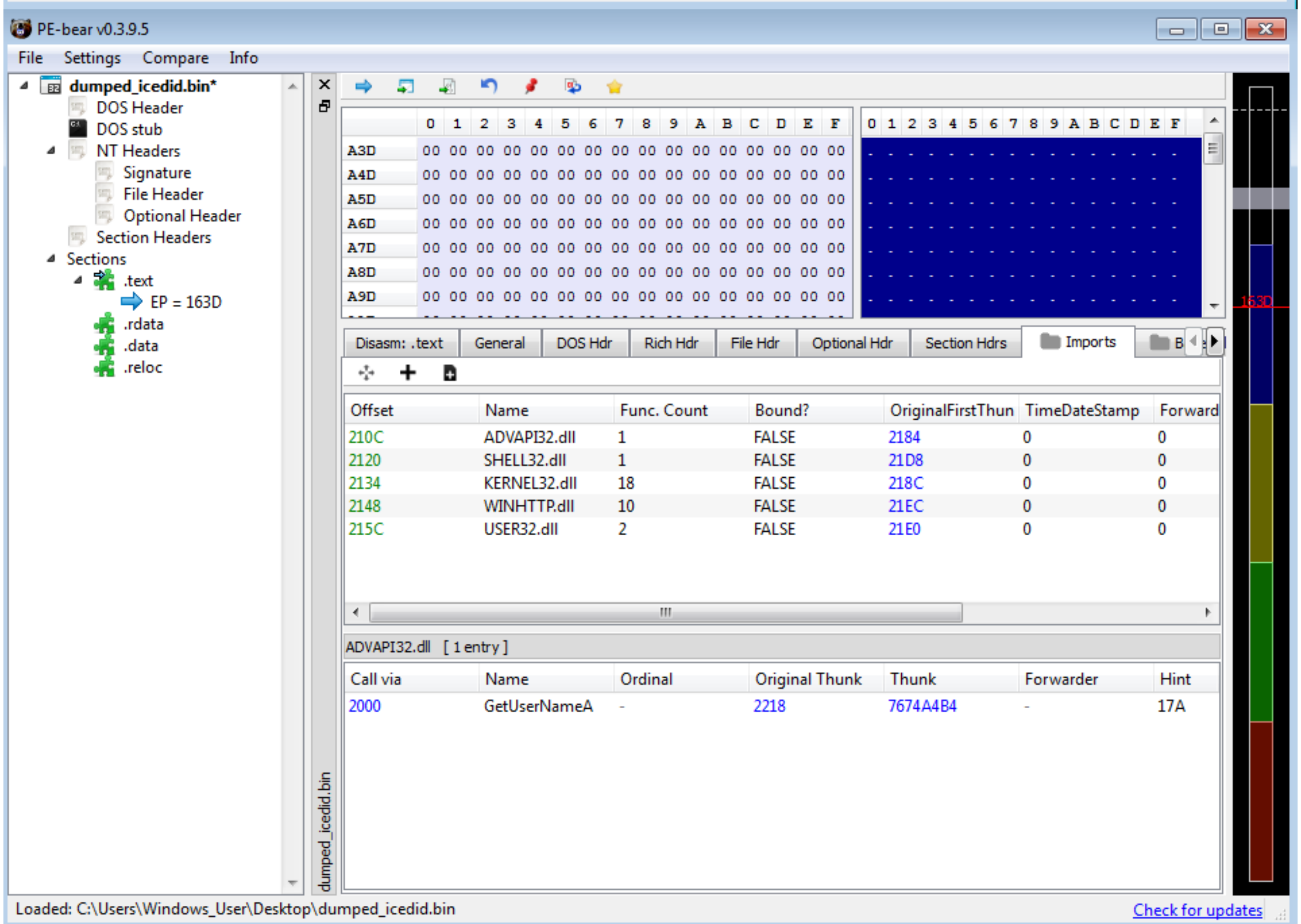


With the executable dumped from memory, we now need to unmap it. As I said before, the packer maps it into memory, and so if we were to open the program up in IDA Pro, it would try and resolve values as if the executable was unmapped – so instead of finding the **.text** section at **0x024F1000**, it would look at **0x00040400** (as an example). Therefore, we need to unmap it. The best tool to do this is **PE-Bear**, and upon opening the program up in it, we want to go to the *Section Hdrs* tab, and change the **Raw Addr.** tab to match the values shown in the **Virtual Addr.** tab, and then change the **Raw Size** values so they match up as well (this is the difference between the sections in terms of size) – so let's go ahead and change that!



If everything has gone as planned, you'll now have something that looks like the image below! If you check the *Imports* tab, you'll notice that they have also been filled in, so now we can view all the imports that the malware uses at some point in it's execution! Go ahead and save the fixed dump by right clicking the filename in the top left and choosing to *Save Executable As…*and congratulations! You have successfully unpacked this VB packed sample of an IcedID Loader!


PE-bear v0.3.9.5
File Settings Compare Info
dumped_icedid.bin*
DOS Header
DOS stub
NT Headers
Signature
File Header
Optional Header
Section Headers
Sections
.text
EP = 163D
.rdata
.data
.reloc
Disasm: .text General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports
Name Raw Addr. Raw size Virtual Addr. Virtual Size Characteristics Ptr to Reloc. Num. of Reloc.
.text 1000 1000 1000 932 60000020 0 0 0
.rdata 2000 1000 2000 468 40000040 0 0 0
.data 3000 1000 3000 250 C0000040 0 0 0
.reloc 4000 1000 4000 8C 42000040 0 0 0
Raw
Virtual
1000
163D
2000
3000
4000
[.text]
[.rdata]
[.data]
[.reloc]
dumped_icedid.bin
Loaded: C:\Users\Windows_User\Desktop\dumped_icedid.bin
Check for updates


PE-bear v0.3.9.5
File Settings Compare Info
dumped_icedid.bin*
DOS Header
DOS stub
NT Headers
Signature
File Header
Optional Header
Section Headers
Sections
.text
EP = 163D
.rdata
.data
.reloc
Disasm: .text General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports
Offset Name Func. Count Bound? OriginalFirstThun TimeDateStamp Forward
210C ADVAPI32.dll 1 FALSE 2184 0 0
2120 SHELL32.dll 1 FALSE 21D8 0 0
2134 KERNEL32.dll 18 FALSE 218C 0 0
2148 WINHTTP.dll 10 FALSE 21EC 0 0
215C USER32.dll 2 FALSE 21E0 0 0
ADVAPI32.dll [1 entry]
Call via Name Ordinal Original Thunk Thunk Forwarder Hint
2000 GetUserNameA - 2218 7674A4B4 - 17A
dumped_icedid.bin
Loaded: C:\Users\Windows_User\Desktop\dumped_icedid.bin
Check for updates