

Unpacking Visual Basic Packers – IcedID

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

0verfl0wz2a

June 22, 2020

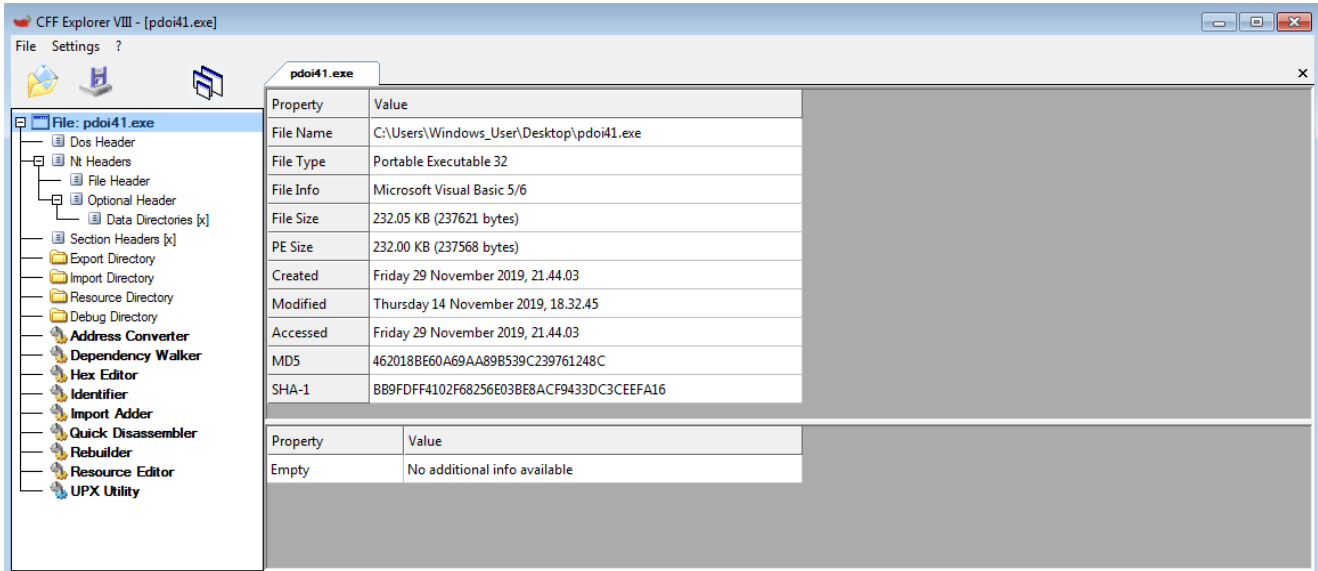
Despite the fact that VisualBasic is an age-old programming language, it is still being used to develop malicious software – specifically packers – to this day. As a result, you will often encounter VisualBasic based packers used in a lot of “script-kiddie” malware, such as keyloggers and remote access 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:

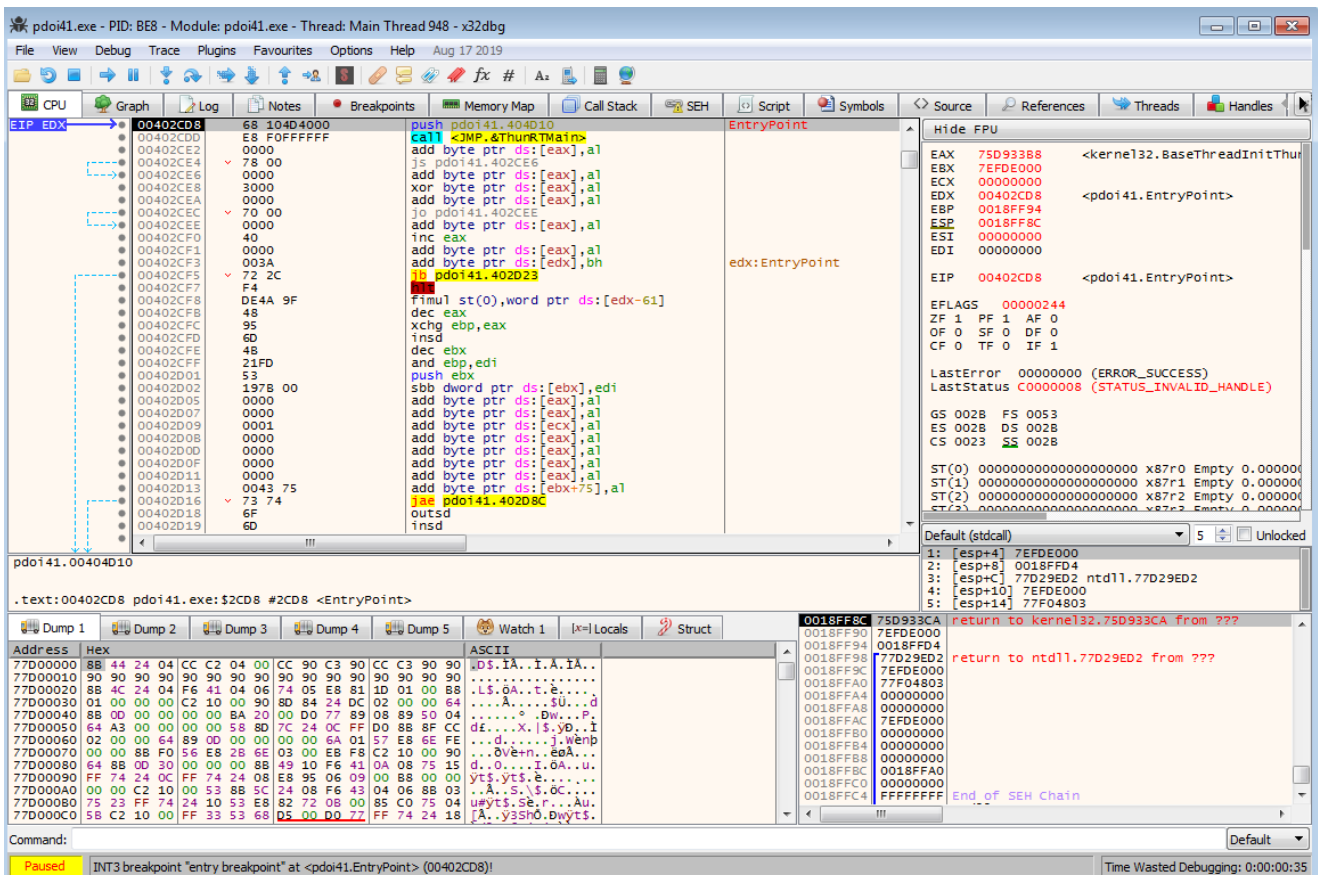
MD5 Hash of Packed Sample: `462018be60a69aa89b539c239761248c`

Initial Analysis:

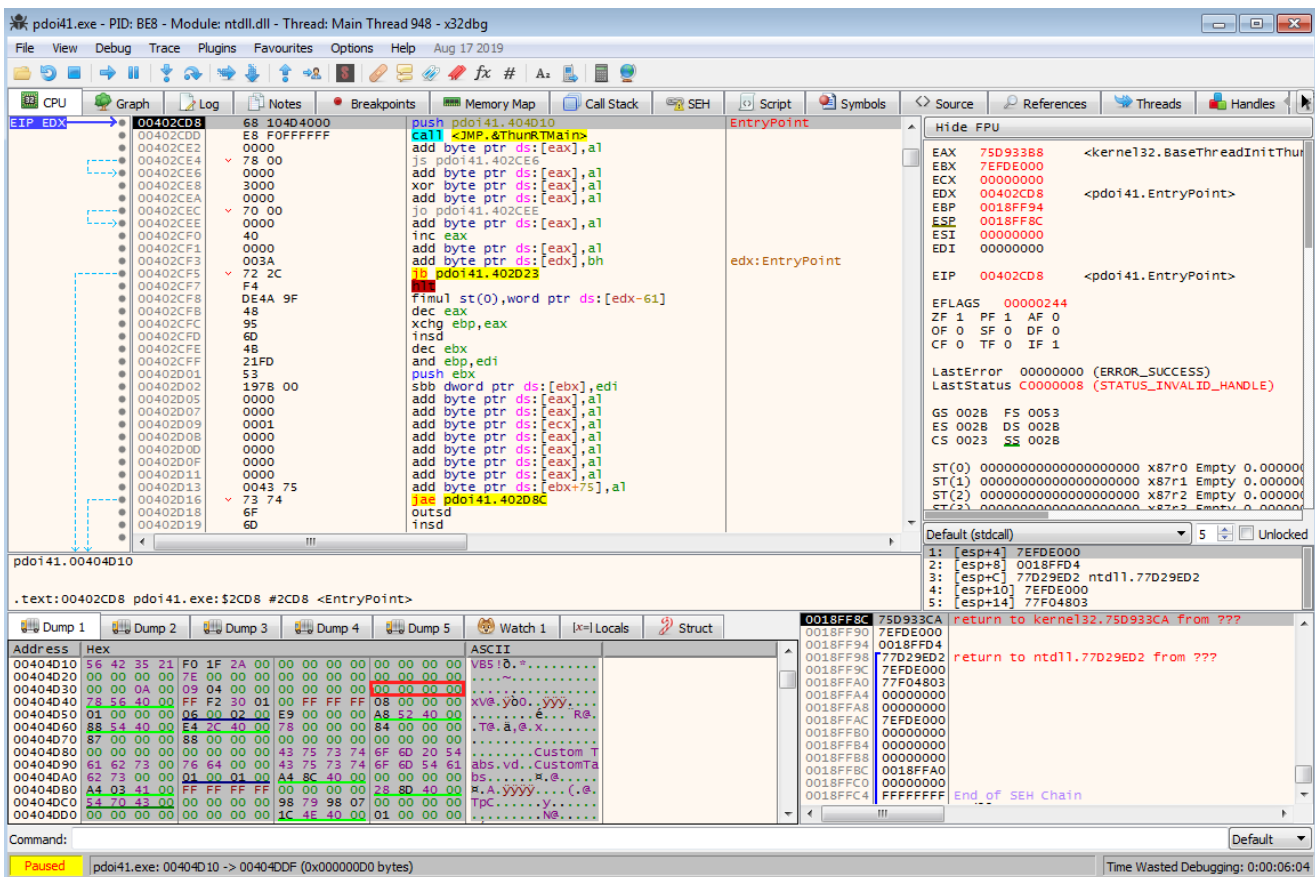
So, while I do know that this sample is packed as I have looked at it before, let’s approach it as if we had no idea what it was. First things first, we want to open it up in a PE analysis tool, which in this case I will be using **CFF Explorer**. One of the good things about **CFF Explorer** is the 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.



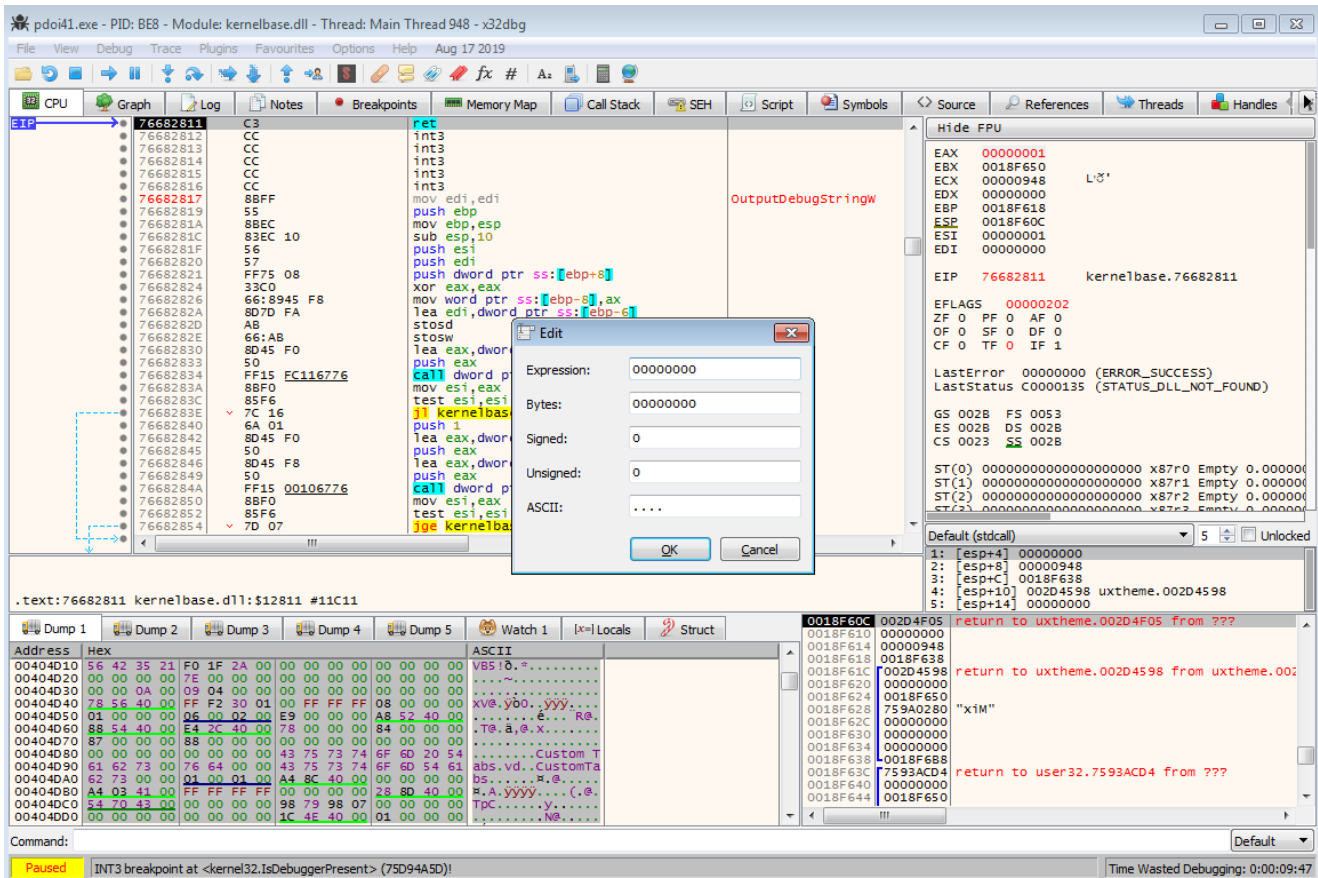
When unpacking VB based packers, I like to put a breakpoint on 4 main API calls:

- **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!

The screenshot displays the Immunity Debugger interface for the process `pdoi41.exe`. The main assembly window shows the following instructions:

```

7667E37E C2 1000 ret 10
7667E381 CC int3
7667E382 CC int3
7667E383 CC int3
7667E384 CC int3
7667E385 CC int3
7667E386 8BFF mov edi,edi
7667E388 55 push ebp
7667E389 8BEC mov ebp,esp
7667E38B 56 push esi
7667E38C 8B75 0C mov esi,dword ptr ss:[ebp+C]
7667E38F 85F6 test esi,esi
7667E391 75 0C jne kernelbase.7667E39F
7667E393 6A 57 push 57
7667E395 FF15 4C106776 call dword ptr ds:[<K&Rt>RestoreLastWin3
7667E39B 33C0 xor eax,eax
7667E39D EB 37 jmp kernelbase.7667E3D6
7667E39F 8D45 0C lea eax,dword ptr ss:[ebp+C]
7667E3A2 50 push eax
7667E3A3 FF15 E0116776 call dword ptr ds:[<K&Rt>GetCurrentProce
7667E3A9 56 push esi
7667E3AA FF36 push dword ptr ds:[esi]
7667E3AC 8D45 0C lea eax,dword ptr ss:[ebp+C]
7667E3AF FF75 08 push dword ptr ss:[ebp+8]
7667E3B2 6A 02 push 2
7667E3B4 50 push eax
7667E3B5 6A 49 push 49
7667E3B7 FF15 DC116776 call dword ptr ds:[<K&Rt>QuerySystemInfor
7667E3BD 3D 040000C0 cmp eax,C0000004
7667E3C2 75 03 jne kernelbase.7667E3C7
7667E3C4 83C0 1F add eax,1F
7667E3C7 85C0 test eax,eax
7667E3C9 7D 08 jge kernelbase.7667E3D3

```

The registers window shows the following values:

```

EAX 00230000
EBX 72A4ED60
ECX 06500000
EDX 0008E3C8
EBP 0018FC18
ESP 0018FBD4
ESI 75D91856
EDI 00230000

```

The memory dump window shows a dump of 0x00000001 bytes at address 00230000.

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!

pdoint1.exe - PID: BE8 - Module: kernel32.dll - Thread: Main Thread 948 - x32dbg

File View Debug Trace Plugins Favourites Options Help Aug 17 2019

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles

EIP ESI
 75D91856 88FF mov edi,edi
 75D91858 55 push ebp
 75D91859 88EC mov ebp,esp
 75D9185B 5D pop ebp
 75D9185C EB 05 jmp <JMP.&VirtualAlloc>
 75D9185E 90 nop
 75D9185F 90 nop
 75D91860 90 nop
 75D91861 90 nop
 75D91862 90 nop
 75D91863 FF25 0809D975 jmp dword ptr ds:[<&VirtualAlloc>] JMP.&VirtualAlloc
 75D9186A 90 nop
 75D9186B 90 nop
 75D9186C 90 nop
 75D9186D 90 nop
 75D9186E 88FF mov edi,edi VirtualFree
 75D91870 55 push ebp
 75D91871 88EC mov ebp,esp
 75D91873 5D pop ebp
 75D91874 EB 05 jmp <JMP.&VirtualFree>
 75D91876 90 nop
 75D91877 90 nop
 75D91878 90 nop
 75D91879 90 nop
 75D9187A 90 nop
 75D9187B FF25 1009D975 jmp dword ptr ds:[<&VirtualFree>] JMP.&VirtualFree
 75D91881 90 nop
 75D91882 90 nop
 75D91883 90 nop
 75D91884 90 nop
 75D91885 90 nop
 75D91886 88FF mov edi,edi DuplicateHandle

edi=2000
 .text:75D91856 kernel32.dll:11856 #1C56 <VirtualAlloc>

Hide FPU
 EAX 00230000
 EBX 72A4ED60 msvbvm60.72A4ED60
 ECX 00002000
 EDX 00001000
 EBP 0018FBD4
 ESP 0018FBD4
 ESI 75D91856 <kernel32.VirtualAlloc>
 EDI 00002000
 EIP 75D91856 <kernel32.VirtualAlloc>
 EFLAGS 0000304
 ZF 0 PF 1 AF 0
 OF 0 SF 0 DF 0
 CF 0 TF 1 IF 1
 LastError 00000583 (ERROR_CLASS_DOES_NOT_EXIST)
 LastStatus C0000135 (STATUS_DLL_NOT_FOUND)
 GS 002B FS 0053
 ES 002B DS 002B
 CS 0023 SS 002B
 ST(0) 000000000000000000000000 x87r0 Empty 0.00000
 ST(1) 000000000000000000000000 x87r1 Empty 0.00000
 ST(2) 000000000000000000000000 x87r2 Empty 0.00000
 ST(3) 000000000000000000000000 x87r3 Empty 0.00000

Default (stdcall) 5 Unlocked
 1: [esp+4] 00230000
 2: [esp+8] 00002000
 3: [esp+C] 00001000
 4: [esp+10] 00000004
 5: [esp+14] 72A4ED60 msvbvm60.72A4ED60

0018FBD4 72944CA0 return to msvbvm60.72944CA0 from ???
 0018FBD8 00230000
 0018FBD0 00002000
 0018FBE0 00001000
 0018FBF4 00000004
 0018FBE8 72A4ED60 msvbvm60.72A4ED60
 0018FBE4 0000002D
 0018FBF0 0000003C
 0018FBF4 00000000
 0018FBF8 00001000
 0018BFC 00010000
 0018FC00 7FFFFFFF
 0018FC04 0000000F
 0018FC08 00000004
 0018FC0C 0000002D

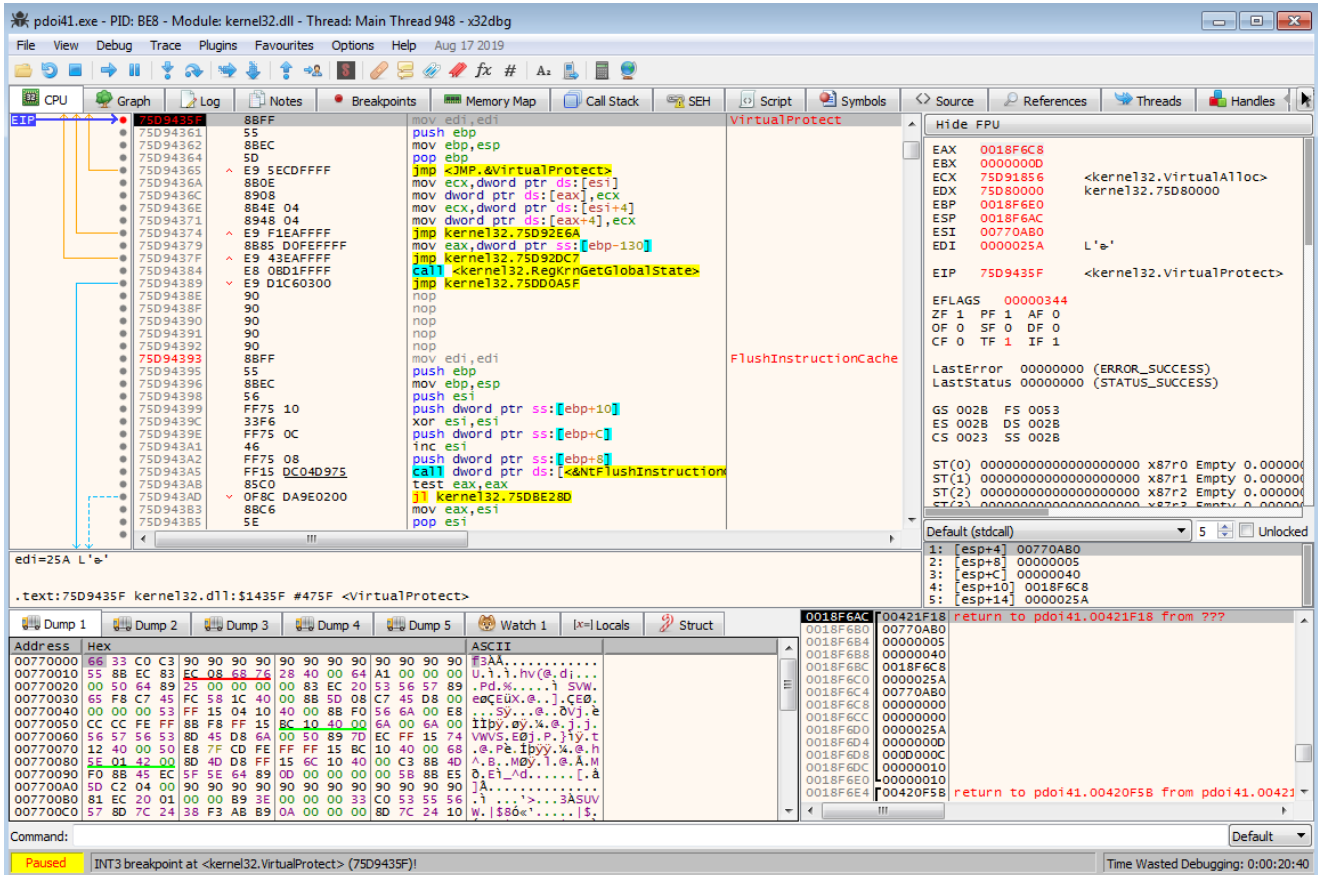
Address	Hex	ASCII
00230000	00 00 00 00 00 00 00 68 10 00 00 00 8B C4 50h....AP
00230010	8D 44 24 0C 50 B9 37 87 9A 72 FF D1 59 0B C0 78	.DS.P17..ryNY,Ax
00230020	0C 88 44 24 04 88 00 FF A0 40 00 00 5A 03 E1	..DS...y@...Z,ã
00230030	52 C3 00 00 00 00 00 68 10 00 00 00 8B C4 50	RÄ.....h....AP
00230040	8D 44 24 0C 50 B9 37 87 9A 72 FF D1 59 0B C0 78	.DS.P17..ryNY,Ax
00230050	0C 88 44 24 04 88 00 FF A0 40 00 00 5A 03 E1	..DS...y@...Z,ã
00230060	52 C3 00 00 00 00 00 68 11 00 00 00 8B C4 50	RÄ.....h....AP
00230070	8D 44 24 0C 50 B9 37 87 9A 72 FF D1 59 0B C0 78	.DS.P17..ryNY,Ax
00230080	0C 88 44 24 04 88 00 FF A0 44 00 00 5A 03 E1	..DS...y D...Z,ã
00230090	52 C3 00 00 00 00 00 68 11 00 00 00 8B C4 50	RÄ.....h....AP
002300A0	8D 44 24 0C 50 B9 37 87 9A 72 FF D1 59 0B C0 78	.DS.P17..ryNY,Ax
002300B0	0C 88 44 24 04 88 00 FF A0 44 00 00 5A 03 E1	..DS...y D...Z,ã
002300C0	52 C3 00 00 00 00 00 68 12 00 00 00 8B C4 50	RÄ.....h....AP

Command: [Paused] INT3 breakpoint at <kernel32.VirtualAlloc> (75D91856) Time Wasted Debugging: 0:00:16:22

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 screenshot shows a debugger window for 'pdoi41.exe - PID: BE8 - Module: kernelbase.dll - Thread: Main Thread 948 - x32dbg'. The CPU window displays assembly instructions, including several calls to kernelbase functions like `RestoreLastWin32`, `GetCurrentProcess`, and `QuerySystemInformation`. The registers window shows the state of various registers, including EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. The dump window shows a memory dump starting at address 00770000. The command window shows the current command: `Dump: 00770000 -> 00770000 (0x00000001 bytes)`.

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).

pdci41.exe - PID: BE8 - Module: kernel32.dll - Thread: Main Thread 948 - x32dbg

File View Debug Trace Plugins Favourites Options Help Aug 17 2019

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles

EIP: 75D91856 8BFF mov edi,edi
 75D91858 55 push ebp
 75D91859 8BEC mov ebp,esp
 75D9185D 5D pop ebp
 75D9185C EB 05 jmp <JMP.&VirtualAlloc>
 75D9185E 90 nop
 75D9185F 90 nop
 75D91860 90 nop
 75D91861 90 nop
 75D91862 90 nop
 75D91863 FF25 0809975 jmp dword ptr ds:[<&VirtualAlloc>]
 75D91864 90 nop
 75D91865 90 nop
 75D91866 90 nop
 75D91867 90 nop
 75D91868 90 nop
 75D91869 90 nop
 75D9186A 90 nop
 75D9186B 90 nop
 75D9186C 90 nop
 75D9186D 90 nop
 75D9186E 8BFF mov edi,edi
 75D91870 55 push ebp
 75D91871 8BEC mov ebp,esp
 75D91873 5D pop ebp
 75D91874 EB 05 jmp <JMP.&VirtualFree>
 75D91876 90 nop
 75D91877 90 nop
 75D91878 90 nop
 75D91879 90 nop
 75D9187A 90 nop
 75D9187B 90 nop
 75D9187C 90 nop
 75D9187D 90 nop
 75D9187E 90 nop
 75D9187F 90 nop
 75D91880 90 nop
 75D91881 FF25 1009975 jmp dword ptr ds:[<&VirtualFree>]
 75D91882 90 nop
 75D91883 90 nop
 75D91884 90 nop
 75D91885 90 nop
 75D91886 8BFF mov edi,edi

VirtualAlloc
 JMP.&VirtualAlloc
 VirtualFree
 JMP.&VirtualFree
 DuplicateHandle

Hide FPU
 EAX 024F1000
 EBX 00000000
 ECX 00000000
 EDX 00000932 L'at'
 EBP 00000003
 ESP 0018F484
 ESI 004F2E60 ".text"
 EDI 024F0000
 EIP 75D91856 <kernel32.VirtualAlloc>
 EFLAGS 00000304
 ZF 0 PF 1 AF 0
 OF 0 SF 0 DF 0
 CF 0 TF 1 IF 1
 LastError 00000000 (ERROR_SUCCESS)
 LastStatus 00000000 (STATUS_SUCCESS)
 GS 002B FS 0053
 ES 002B DS 002B
 CS 0023 SS 002B
 ST(0) 000000000000000000000000 x87r0 Empty 0.000000
 ST(1) 000000000000000000000000 x87r1 Empty 0.000000
 ST(2) 000000000000000000000000 x87r2 Empty 0.000000
 ST(3) 000000000000000000000000 x87r3 Empty 0.000000

Default (stdcall) 5 Unlocked
 1: [esp+4] 024F1000
 2: [esp+8] 00000932
 3: [esp+C] 00001000
 4: [esp+10] 00000004
 5: [esp+14] 00000000

return to 00770184 from 00770A80

Command: [Paused] INT3 breakpoint at <kernel32.VirtualAlloc> (75D91856) Time Wasted Debugging: 0:00:22:26

pdci41.exe - PID: BE8 - Module: kernel32.dll - Thread: Main Thread 948 - x32dbg

File View Debug Trace Plugins Favourites Options Help Aug 17 2019

CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles

EIP: 75D9435F 8BFF mov edi,edi
 75D94361 55 push ebp
 75D94362 8BEC mov ebp,esp
 75D94364 5D pop ebp
 75D94365 E9 5ECCDFFF jmp <JMP.&VirtualProtect>
 75D9436A 8B0E mov ecx,dword ptr ds:[esi]
 75D9436C 8908 mov dword ptr ds:[ecx],ecx
 75D9436E 8B4E 04 mov ecx,dword ptr ds:[esi+4]
 75D94371 8948 04 mov dword ptr ds:[ecx+4],ecx
 75D94374 E9 5FAEFFFF jmp kernel32.75D92E6A
 75D94379 mov ecx,dword ptr ss:[ebp-130]

VirtualProtect

Hide FPU
 EAX 024F0000
 EBX 00000000
 ECX 00000932 L'at'
 EDX 024F1000
 EBP 0018F5C8
 ESP 0018F43C

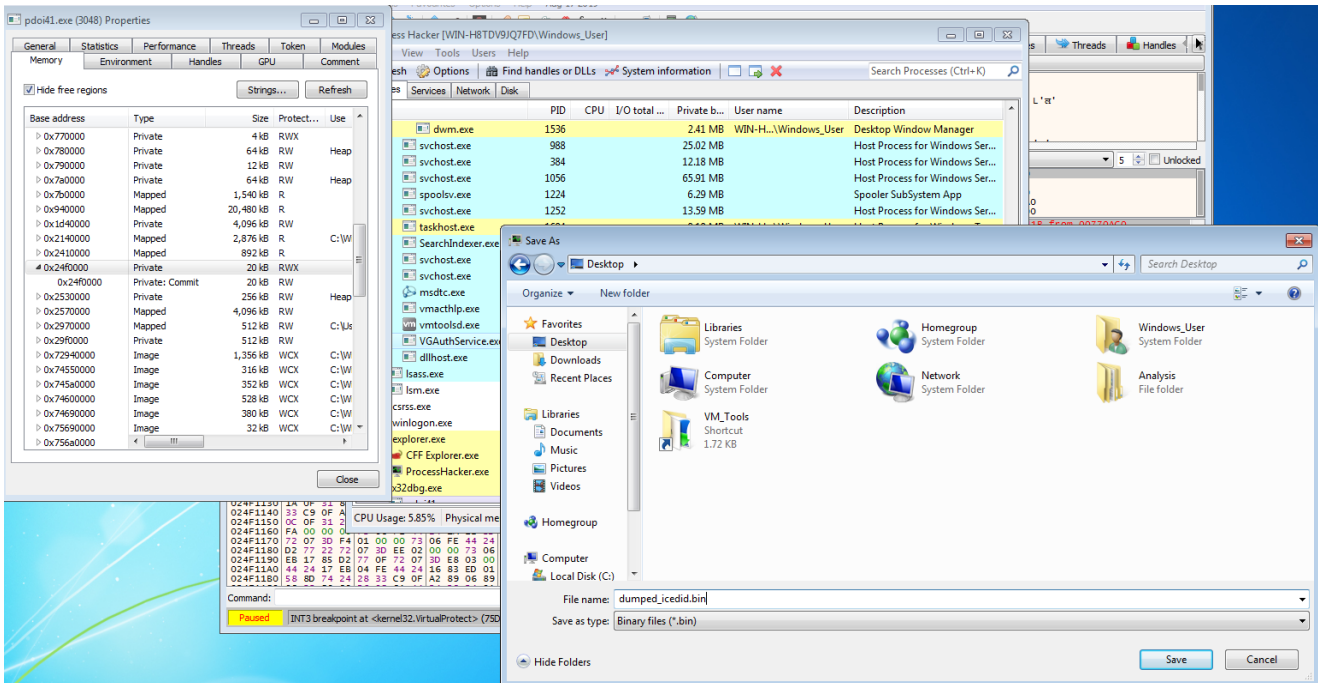
Default (stdcall) 5 Unlocked
 1: [esp+4] 024F1000
 2: [esp+8] 00000932
 3: [esp+C] 00000020
 4: [esp+10] 0018F4A0
 5: [esp+14] 00000000

return to 00770418 from 00770A0C

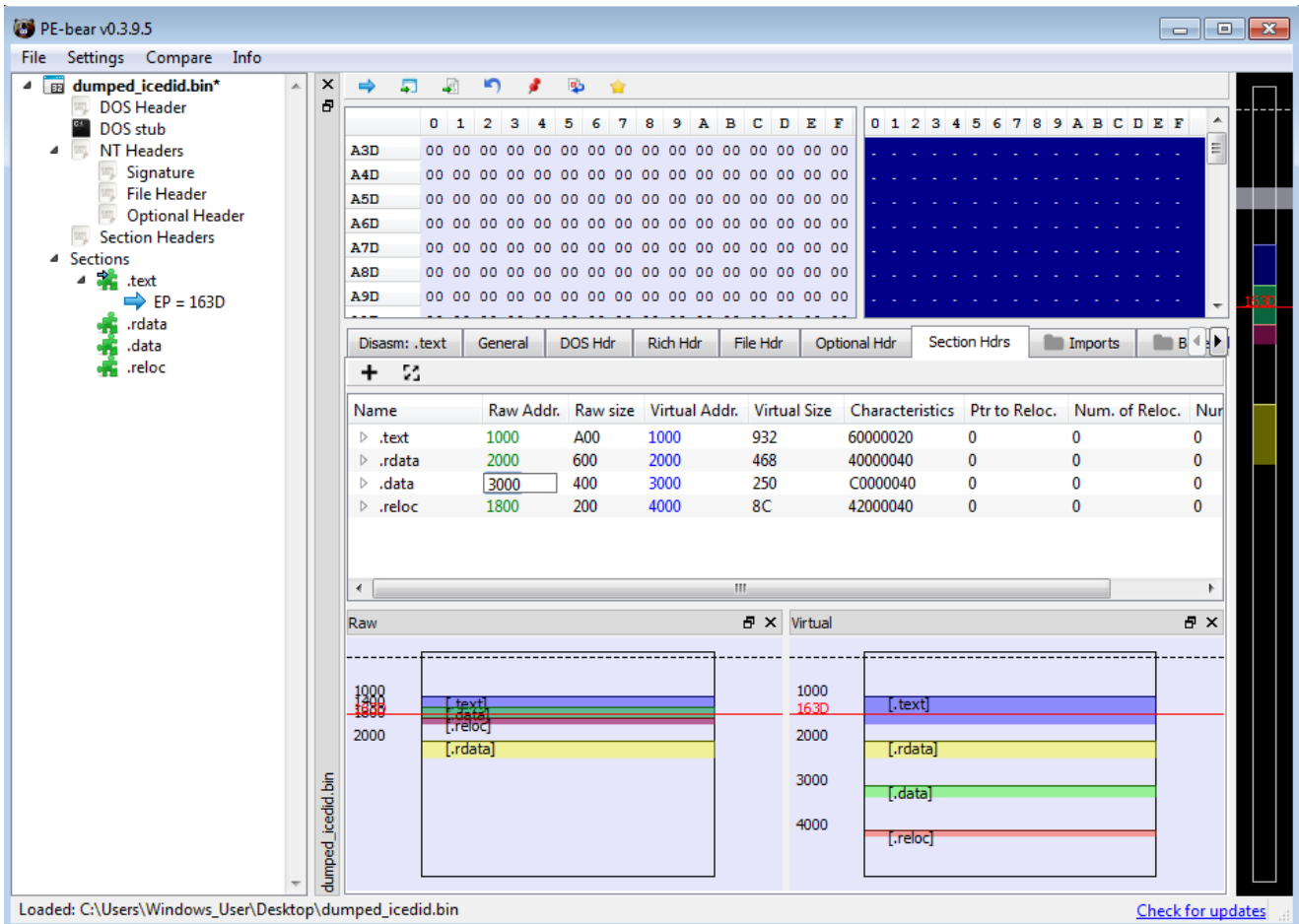
Command: [Paused] INT3 breakpoint at <kernel32.VirtualProtect> (75D9435F) Time Wasted Debugging: 0:00:26:49

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 its 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 B

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Nur
▷ .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

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 B

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeStamp	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

Loaded: C:\Users\Windows_User\Desktop\dumped_icedid.bin [Check for updates](#)