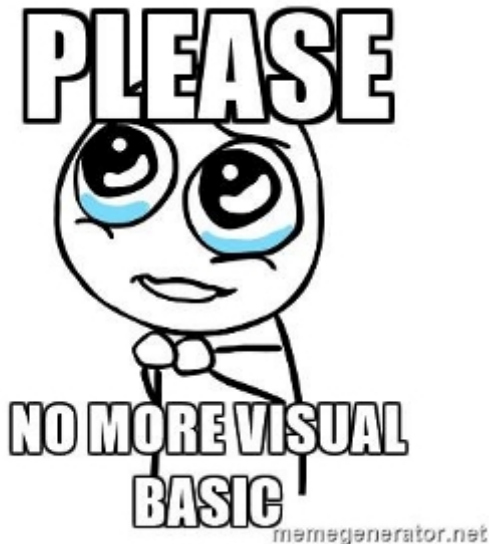


Defeating the VB5 Packer



JUNE 7, 2017 JUNE 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: 730b761f833a0f0636b04ba315c5ceedd0baaba

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).

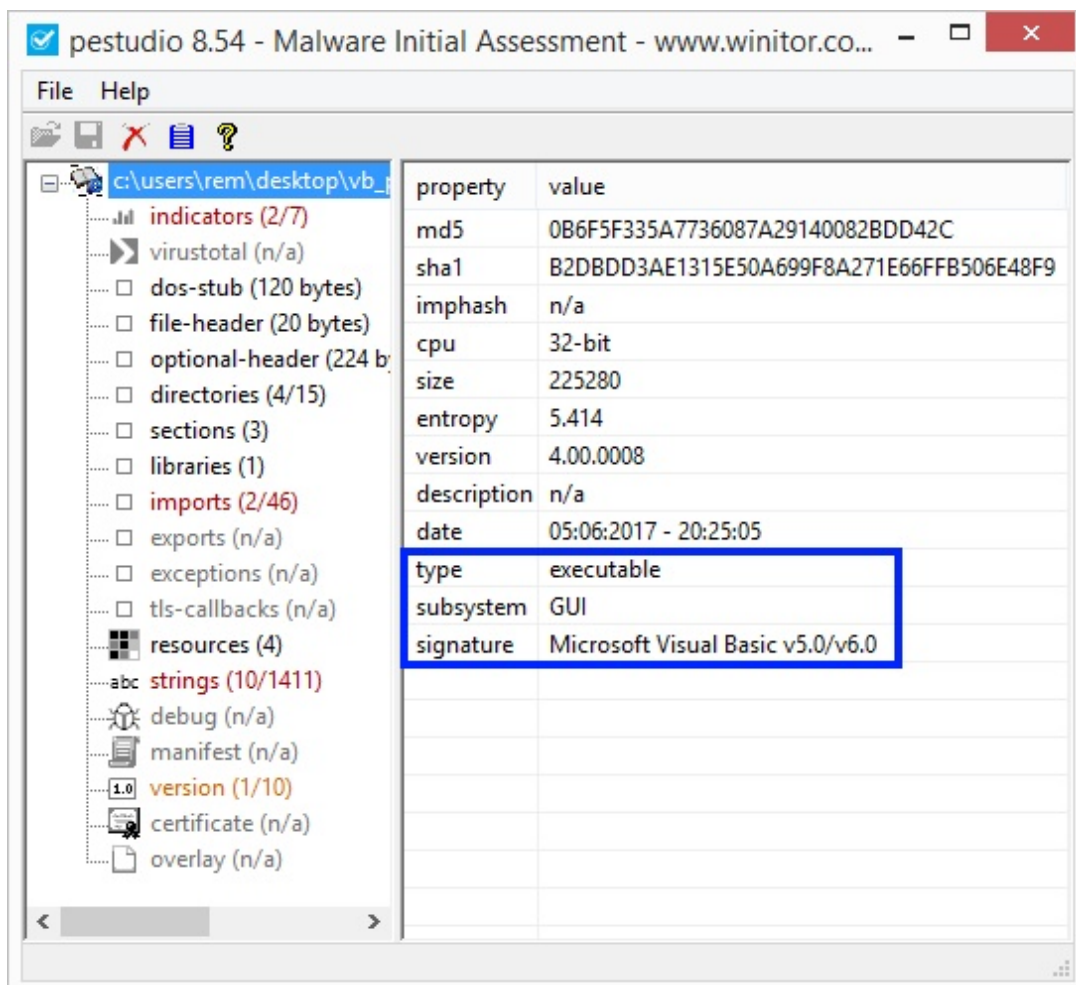


Figure 1: PESTudio depicting sample is a VB Executable

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: worldDCoin (<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):

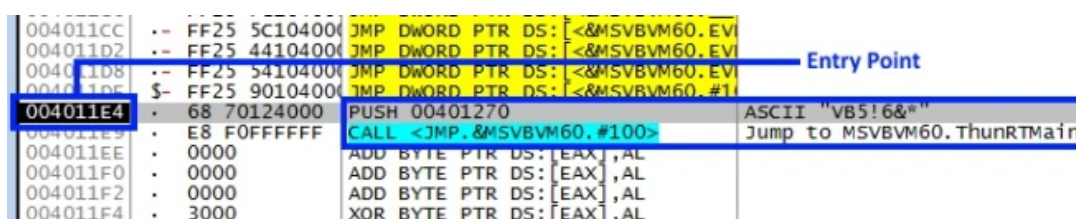


Figure 2: Sitting at Entry Point of the sample

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 00 B0 27 43 00	° 'C
004012A0	1C 18 40 00 00 F0 30 00 00 FF FF FF 08 00 00 00	↑@ 80 é yyy
004012B0	01 00 00 00 03 00 00 00 E9 00 00 00 00 00 13 40 00	↑@ 80 é yyy
004012C0	2C 12 40 00 00 F0 11 40 00 78 00 00 00 81 00 00 00	↑@ 80 é yyy
004012D0	88 00 00 00 89 00 00 00 00 00 00 00 00 00 00 00	↑@ 80 é yyy
004012E0	00 00 00 00 00 00 00 00 56 69 74 61 6D 69 6E 61	↑@ 80 é yyy
004012F0	00 54 61 69 6E 73 79 00 00 43 6F 64 6F 6E 73 00	↑@ 80 é yyy

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 00 00]	
BackupLanguageDLL	[7E 00 00 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).

004327B0	55	PUSH EBP	
004327B1	8B EC	MOV EBP,ESP	
004327B3	83 EC 08	SUB ESP,8	
004327B6	68 D6104000	PUSH <JMP.&MSVBVM60.__vbaExcept	Jump to MSVBVM60.__vbaExceptionHandler
004327BB	64:A1 000000	MOV EAX,DWORD PTR FS:[0]	
004327C1	50	PUSH EAX	
004327C2	64:8925 0000	MOV DWORD PTR FS:[0],ESP	Installs SE handler 4010D6
004327C9	83 EC 44	SUB ESP,44	
004327CC	53	PUSH EBX	
004327CD	56	PUSH ESI	
004327CE	57	PUSH EDI	
004327CF	8965 F8	MOV DWORD PTR SS:[EBP-8],ESP	
004327D2	C745 FC C010	MOV DWORD PTR SS:[EBP-4],004010	
004327D9	A1 10404300	MOV EAX,DWORD PTR DS:[434010]	
004327DE	33 FF	XOR EDI,EDI	
004327E0	3B C7	CMP EAX,EDI	
004327E2	897D E0	MOV DWORD PTR SS:[EBP-20],EDI	
004327E5	897D D0	MOV DWORD PTR SS:[EBP-30],EDI	
004327E8	897D C0	MOV DWORD PTR SS:[EBP-40],EDI	
004327EB	897D BC	MOV DWORD PTR SS:[EBP-44],EDI	
004327EE	75 10	JNE SHORT 00432800	

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).

7FFDC000	00	DB 00	InheritedAddressSpace = 0
7FFDC001	00	DB 00	ReadImageFileExecOptions = 0
7FFDC002	01	DB 01	BeingDebugged = TRUE
7FFDC003	00	DB 00	SpoolError = FALSE
7FFDC004	FFFFFFF	DD FFFFFFFF	Mutant = INVALID_HANDLE_VALUE
7FFDC008	00004000	DD OFFSET vitamina_Packed.<STRUCT IMAGE_DOS_HEADER>	ImageBaseAddress = 00400000
7FFDC00C	40947677	DD OFFSET ntdll.77769440	LoaderData = ntdll.77769440
7FFDC010	90103000	DD 00301090	ProcessParameters = 301090
7FFDC014	00000000	DD 00000000	SubsystemData = NULL
7FFDC018	00003000	DD 00300000	ProcessHeap = 00300000
7FFDC01C	A0937677	DD OFFSET ntdll.777693A0	FastPeblock = ntdll.777693A0
7FFDC020	00000000	DD 00000000	FastPeblockRoutine = 00000000
7FFDC024	00000000	DD 00000000	FastPebunlockRoutine = 00000000
7FFDC028	01000000	DD 00000001	EnvironmentUpdateCount = 1
7FFDC02C	204BC076	DD USER32.76C04B20	KernelCallbackTable = 76C04B20
7FFDC030	00000000	DD 00000000	Reserved = 0
7FFDC034	00000000	DD 00000000	Thunksoroptions = 0
7FFDC038	00000300	DD 00030000	FreeList = 30000
7FFDC03C	00000000	DD 00000000	TlsExpansionCounter = 0
7FFDC040	E0937677	DD OFFSET ntdll.777693E0	TlsBitmap = ntdll.777693E0
7FFDC044	FF000100	DD 000100FF	TlsBitmapbits[2] = 100FF
7FFDC048	00000000	DD 00000000	ReadOnlySharedMemoryBase = 7FEB0000
7FFDC04C	0000E87F	DD 7FEB0000	ReadOnlySharedMemoryHeap = NULL
7FFDC050	00000000	DD 00000000	ReadOnlyStaticServerData = 7FEB04A0
7FFDC054	A004E87F	DD 7FEB04A0	AnsiCodePageData = 7FEB0000
7FFDC058	0000F87F	DD 7FEB0000	OemCodePageData = 7FFC0224
7FFDC05C	2402FC7F	DD 7FFC0224	UnicodeCaseTableData = 7FFD0648
7FFDC060	4806FD7F	DD 7FFD0648	NumberOfProcessors = 2
7FFDC064	02000000	DD 00000002	NTGlobalFlag = 112.
7FFDC068	70000000	DD 00000070	Reserved = 0
7FFDC06C	00000000	DD 00000000	CriticalSectionTimeoutLo = 7988000
7FFDC070	00809807	DD 07988000	CriticalSectionTimeoutHi = -1793
7FFDC074	60E8FFFF	DD FFFFE86D	HeapSegmentReserve = 1048576.
7FFDC078	00001000	DD 00100000	HeapSegmentCommit = 8192.
7FFDC07C	00200000	DD 00002000	HeapCommitTotalFreeThreshold = 65536.
7FFDC080	00000100	DD 00010000	HeapCommitFreeLockThreshold = 4096.
7FFDC084	00100000	DD 00001000	NumberOfHeaps = 5
7FFDC088	05000000	DD 00000005	MaximumNumberOfHeaps = 16.
7FFDC08C	10000000	DD 00000010	ProcessHeaps = 77768520
7FFDC090	20857677	DD OFFSET ntdll.77768520	GdiSharedHandleTable = 00440000
7FFDC094	00004400	DD 00440000	ProcessStarterHelper = NULL
7FFDC098	00000000	DD 00000000	GdiDCAAttributeList = 14
7FFDC09C	14000000	DD 00000014	LoaderLock = 777653D0
7FFDC0A0	D0537677	DD OFFSET ntdll.777653D0	OSMajorVersion = 6
7FFDC0A4	06000000	DD 00000006	OSMinorVersion = 3
7FFDC0A8	03000000	DD 00000003	OSBuildNumber = 9600.
7FFDC0AC	8025	DW 2580	OSCSDVersion = 0
7FFDC0AE	0000	DW 0	OSPlatformId = 2
7FFDC0B0	02000000	DD 00000002	ImageSubsystem = 2
7FFDC0B4	02000000	DD 00000002	ImageSubsystemMajorVersion = 4
7FFDC0B8	04000000	DD 00000004	ImageSubsystemMinorVersion = 0
7FFDC0BC	00000000	DD 00000000	ImageProcessAffinityMask = 3
7FFDC0C0	03000000	DD 00000003	GdiHandleBuffer[34.] = 0
7FFDC0C4	00000000	DD 00000000	

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).

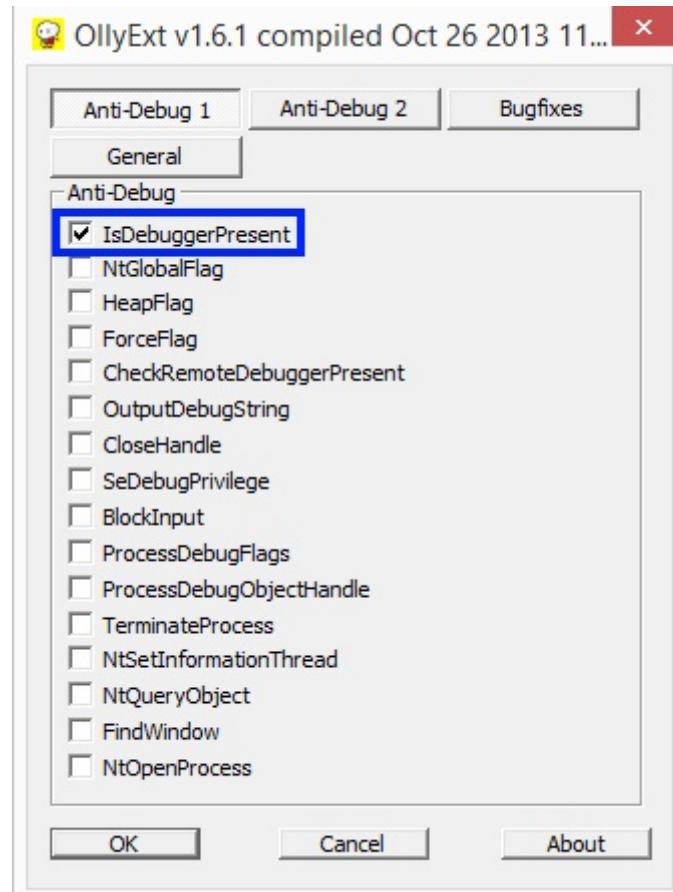


Figure 7: OllyDBG's OllyExt IsDebuggerPresent bypass

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):

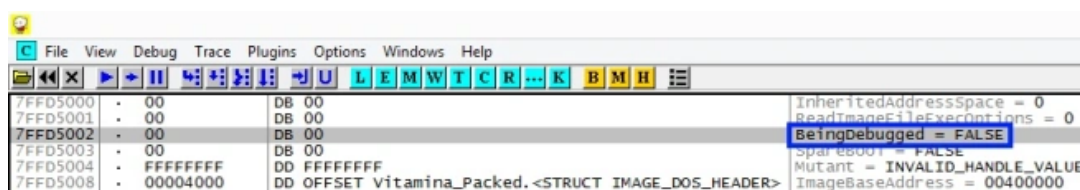


Figure 8: PEB.BeingDebugged flag after enabling OllyExt bypass

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).

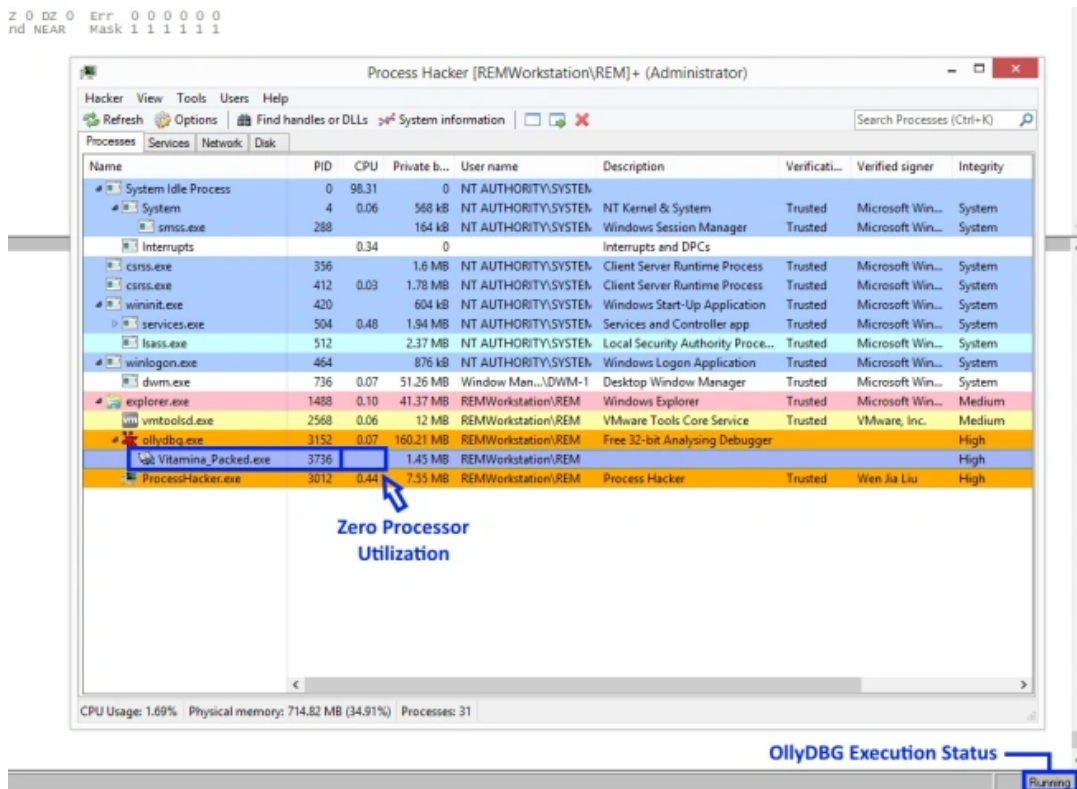


Figure 9: Process Hacker – CPU Utilization before OllyExt bypass

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

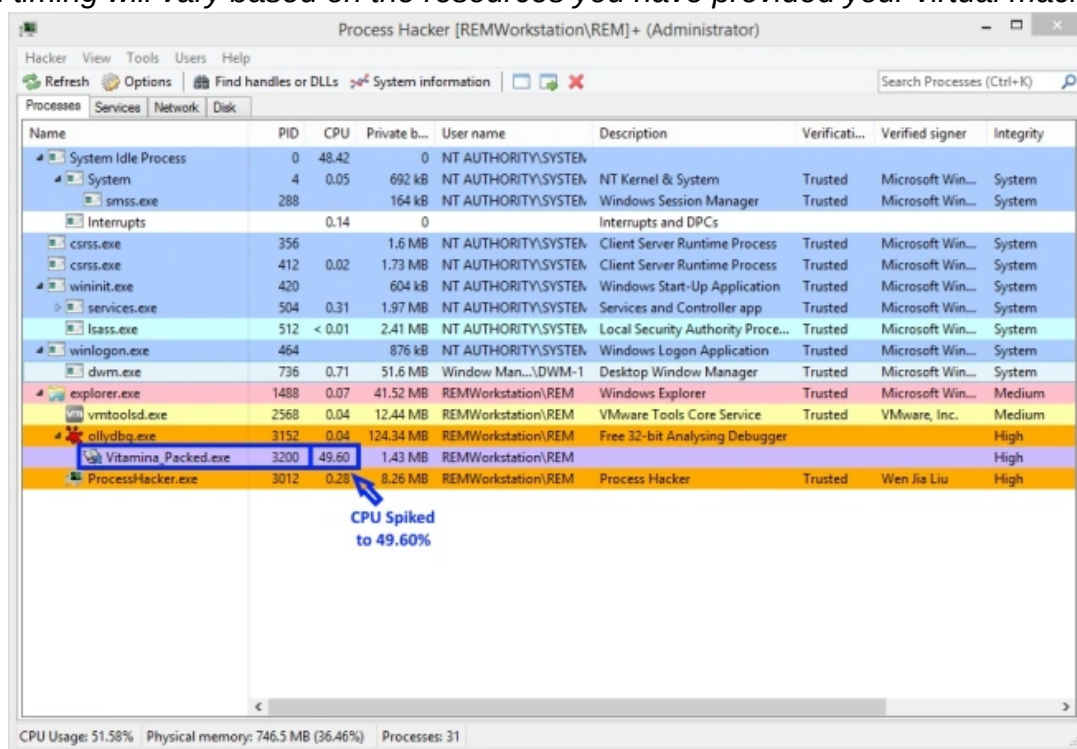


Figure 10: Process Hacker – CPU Utilization after OllyExt bypass

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.

011C01AF	54	PUSH ESP	USER32.GetCursorPos
011C01B0	FFD0	CALL EAX	
011C01B2	83F8 00	CMP EAX,0	
011C01B5	0F84 AF1F0000	JE 011C216A	
011C01B8	0F6F0424	MOVQ MM0,QWORD PTR SS:[ESP]	move cursor coordinates into MM0 (ST0)
011C01BF	6A 01	PUSH 1	Arg1 for KERNEL32.Sleep = 1ms
011C01C1	FF95 94000000	CALL DWORD PTR SS:[EBP+94]	KERNEL32.Sleep
011C01C7	54	PUSH ESP	
011C01C8	FF95 08010000	CALL DWORD PTR SS:[EBP+108]	USER32.GetCursorPos
011C01CE	F8 00	CMP EAX,0	
011C01D1	0F84 931F0000	JE 011C216A	
011C01D7	0F6F0C24	MOVQ MM1,QWORD PTR SS:[ESP]	move cursor coordinates into MM1 (ST1)
011C01DB	0F76C8	PCMPEQD MM1,MM0	compare MM0 to MM1. if equal, MM1=-1, else MM1=0
011C01DE	0F7EC9	MOVD ECX,MM1	
011C01E1	83F9 00	CMP ECX,0	
011C01E4	75 D9	JNE SHORT 011C01BF	jump if cursor has not moved
011C01E6	81C4 00020000	AND ESP,2000	
011C01EC	EB 48	JMP SHORT 011C0236	

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]

011C0236	64:8B1D 300000	MOV EBX,DWORD PTR FS:[30]	access PEB
011C023D	8A58 68	MOV BL,BYTE PTR DS:[EBX+68]	access NtGlobalFlag
011C0240	80E3 70	AND BL,70	
011C0243	80FB 70	CMP BL,70	check if lower 8-bits equals 0x70
011C0246	0F84 1E1F0000	JNE 011C216A	jump if flags are found
011C024C	88 01000000	MOV EAX,1	
011C0251	0FA2	CPUID	
011C0253	89D0	MOV EAX,EDX	
011C0255	C1E8 17	SHR EAX,17	
011C0258	83E0 01	AND EAX,00000001	
011C025B	83F8 01	CMP EAX,1	
011C025E	0F85 061F0000	JNE 011C216A	
011C0264	31C9	XOR ECX,ECX	
011C0266	898D 24010000	MOV DWORD PTR SS:[EBP+124],ECX	
011C026C	E9 411D0000	JMP 011C1FB2	
011C0271	59	POP ECX	
011C0272	894D 68	MOV DWORD PTR SS:[EBP+68],ECX	
011C0275	E9 461D0000	JMP 011C1FC0	
011C027A	5A	POP EDX	
011C027B	E8 1D040000	CALL 011C069D	
011C0280	8945 64	MOV DWORD PTR SS:[EBP+64],EAX	
011C0283	8B4D 04	MOV ECX,DWORD PTR SS:[EBP+4]	
011C0286	E9 B31C0000	JMP 011C1F3E	
011C028B	5A	POP EDX	
011C028C	E8 0C040000	CALL 011C069D	
011C0291	8945 50	MOV DWORD PTR SS:[EBP+50],EAX	
011C0294	E9 F61C0000	JMP 011C1F8F	
011C0299	59	POP ECX	
011C029A	894D 18	MOV DWORD PTR SS:[EBP+18],ECX	
011C029D	E9 641D0000	JMP 011C2006	
011C02A2	5A	POP EDX	
011C02A3	E8 F5030000	CALL 011C069D	
011C02A8	8945 44	MOV DWORD PTR SS:[EBP+44],EAX	
011C02AB	8B4D 04	MOV ECX,DWORD PTR SS:[EBP+4]	
011C02AE	E9 7C1C0000	JMP 011C1F2F	
011C02B3	5A	POP EDX	
011C02B4	E8 E4030000	CALL 011C069D	
011C02B9	8945 4C	MOV DWORD PTR SS:[EBP+4C],EAX	
011C02BC	EB 56	JMP SHORT 011C0314	

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 \(https://www.aldeid.com/wiki/PEB-Process-Environment-Block\)](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.

7FFDD000	. 00	DB 00	InheritedAddressSpace = 0
7FFDD001	. 00	DB 00	ReadImageFileExecOptions = 0
7FFDD002	. 00	DB 00	BeingDebugged = FALSE
7FFDD003	. 00	DB 00	SpareBool = FALSE
7FFDD004	. FFFFFFFF	DD FFFFFFFF	Mutant = INVALID_HANDLE_VALUE
7FFDD008	. 00004000	DD OFFSET vitamina_Packed.<STRU	ImageBaseAddress = 00400000
7FFDD00C	. 40947677	DD OFFSET ntdll.77769440	LoaderData = ntdll.77769440
7FFDD010	. 90101E00	DD 001E1090	ProcessParameters = 1E1090
7FFDD014	. 00000000	DD 00000000	SubsystemData = NULL
7FFDD018	. 00001E00	DD 001E0000	ProcessHeap = 001E0000
7FFDD01C	. A0937677	DD OFFSET ntdll.777693A0	FastPebLock = ntdll.777693A0
7FFDD020	. 00000000	DD 00000000	FastPebLockRoutine = 00000000
7FFDD024	. 00000000	DD 00000000	FastPebUnlockRoutine = 00000000
7FFDD028	. 01000000	DD 00000001	EnvironmentUpdateCount = 1
7FFDD02C	. 2048C076	DD USER32.76C04B20	KernelCallbackTable = 76C04B20
7FFDD030	. 00000000	DD 00000000	Reserved = 0
7FFDD034	. 00000000	DD 00000000	ThunksOrOptions = 0
7FFDD038	. 00000300	DD 00030000	FreeList = 30000
7FFDD03C	. 00000000	DD 00000000	TlsExpansionCounter = 0
7FFDD040	. E0937677	DD OFFSET ntdll.777693E0	TlsBitmap = ntdll.777693E0
7FFDD044	. FF030100	DD 000103FF	TlsBitmapBits[2] = 103FF
7FFDD048	. 00000000	DD 00000000	
7FFDD04C	. 0000EB7F	DD 7FEB0000	ReadOnlySharedMemoryBase = 7FEB0000
7FFDD050	. 00000000	DD 00000000	ReadOnlySharedMemoryHeap = NULL
7FFDD054	. A040EB7F	DD 7FEB04A0	ReadOnlyStaticServerData = 7FEB04A0
7FFDD058	. 0000FB7F	DD 7FFB0000	AnsicodePageData = 7FFB0000
7FFDD05C	. 2402FC7F	DD 7FFC0224	OemCodePageData = 7FFC0224
7FFDD060	. 4806FD7F	DD 7FFD0648	UnicodeCaseTableData = 7FFD0648
7FFDD064	. 02000000	DD 00000002	NumberOfProcessors = 2
7FFDD068	. 70000000	DD 00000070	NtGlobalFlag = 112.
7FFDD06C	. 00000000	DD 00000000	Reserved = 0
7FFDD070	. 00809B07	DD 07988000	CriticalSectionTimeout.Lo = 7988000
7FFDD074	. 6DE8FFFF	DD FFFFE86D	CriticalSectionTimeout.Hi = -1793
7FFDD078	. 00001000	DD 00100000	HeapSegmentReserve = 1048576.
7FFDD07C	. 00200000	DD 00002000	HeapSegmentCommit = 8192.
7FFDD080	. 00000100	DD 00010000	HeapDeCommitTotalFreeThreshold = 65536.
7FFDD084	. 00100000	DD 00001000	HeapDeCommitFreeBlockThreshold = 4096.
7FFDD088	. 07000000	DD 00000007	NumberOfHeaps = 7
7FFDD08C	. 10000000	DD 00000010	MaximumNumberOfHeaps = 16.
7FFDD090	. 20857677	DD OFFSET ntdll.77768520	ProcessHeaps = 77768520
7FFDD094	. 00004400	DD 00440000	GdipSharedHandleTable = 00440000
7FFDD098	. 00000000	DD 00000000	ProcessStarterHelper = NULL
7FFDD09C	. 14000000	DD 00000014	GdiDCAAttributeList = 14
7FFDD0A0	. D0537677	DD OFFSET ntdll.777653D0	LoaderLock = 777653D0
7FFDD0A4	. 06000000	DD 00000006	OSMajorVersion = 6
7FFDD0A8	. 03000000	DD 00000003	OSMinorVersion = 3
7FFDD0AC	. 8025	DW 2580	OSBuildNumber = 9600.
7FFDD0AE	. 0000	DW 0	OSCDVersion = 0
7FFDD0B0	. 02000000	DD 00000002	OSPlatformId = 2
7FFDD0B4	. 02000000	DD 00000002	ImageSubsystem = 2
7FFDD0B8	. 04000000	DD 00000004	ImageSubsystemMajorVersion = 4
7FFDD0BC	. 00000000	DD 00000000	ImageSubsystemMinorVersion = 0
7FFDD0C0	. 03000000	DD 00000003	ImageProcessAffinityMask = 3

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).

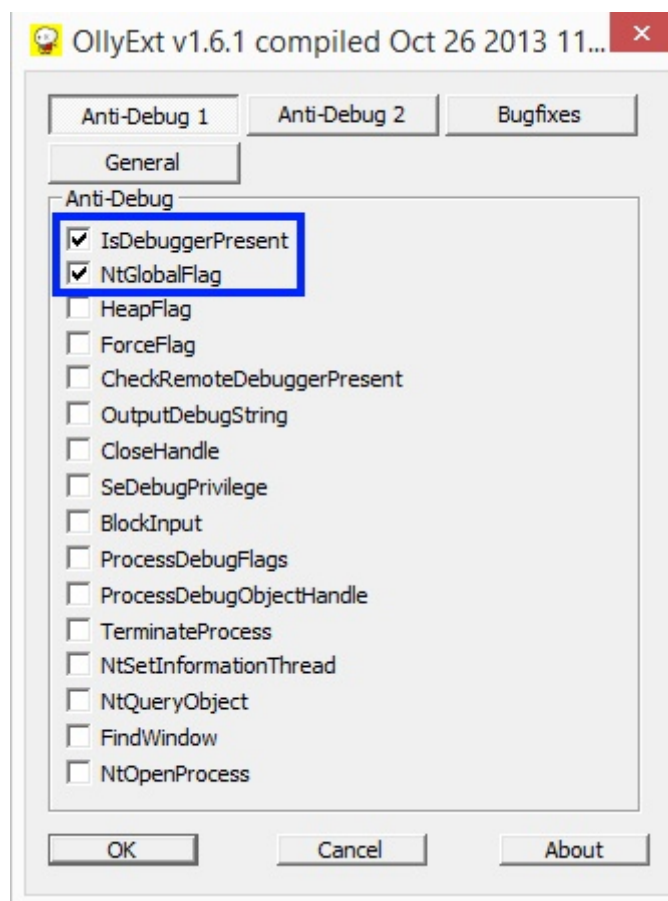


Figure 14: OllyDBG's OllyExt IsDebuggerPresent and NtGlobalFlag bypass

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.

7FFDB000	. 00	DB 00	InheritedAddressSpace = 0
7FFDB001	. 00	DB 00	ReadImageFileExecOptions = 0
7FFDB002	. 00	DB 00	BeingDebugged = FALSE
7FFDB003	. 00	DB 00	Spare001 = FALSE
7FFDB004	. FFFFFFFF	DD FFFFFFFF	Mutant = INVALID_HANDLE_VALUE
7FFDB008	. 00004000	DD OFFSET vitamina_Packed.<STRU	ImageBaseAddress = 00400000
7FFDB00C	. 40947677	DD OFFSET ntdll.77769440	LoaderData = ntdll.77769440
7FFDB010	. 60102C00	DD 002C1060	ProcessParameters = 2C1060
7FFDB014	. 00000000	DD 00000000	SubsystemData = NULL
7FFDB018	. 00002C00	DD 002C0000	ProcessHeap = 002C0000
7FFDB01C	. A0937677	DD OFFSET ntdll.777693A0	FastPebLock = ntdll.777693A0
7FFDB020	. 00000000	DD 00000000	FastPebLockRoutine = 00000000
7FFDB024	. 00000000	DD 00000000	FastPebUnlockRoutine = 00000000
7FFDB028	. 01000000	DD 00000001	EnvironmentUpdateCount = 1
7FFDB02C	. 204BC076	DD USER32.76C04B20	KernelCallbackTable = 76C04B20
7FFDB030	. 00000000	DD 00000000	Reserved = 0
7FFDB034	. 00000000	DD 00000000	ThunksOrOptions = 0
7FFDB038	. 00000300	DD 00030000	FreeList = 30000
7FFDB03C	. 00000000	DD 00000000	TlsExpansionCounter = 0
7FFDB040	. E0937677	DD OFFSET ntdll.777693E0	TlsBitmap = ntdll.777693E0
7FFDB044	. FF000100	DD 000100FF	TlsBitmapBits[2] = 100FF
7FFDB048	. 00000000	DD 00000000	
7FFDB04C	. 0000E87F	DD 7FE80000	ReadOnlySharedMemoryBase = 7FE80000
7FFDB050	. 00000000	DD 00000000	ReadOnlySharedMemoryHeap = NULL
7FFDB054	. A004E87F	DD 7FEB04A0	ReadOnlyStaticServerData = 7FEB04A0
7FFDB058	. 0000F87F	DD 7FFB0000	AnsiCodePageData = 7FFB0000
7FFDB05C	. 2402FC7F	DD 7FFC0224	OemCodePageData = 7FFC0224
7FFDB060	. 4806FD7F	DD 7FFD0648	UnicodeCaseTableData = 7FFD0648
7FFDB064	. 02000000	DD 00000002	NumberOfProcessors = 2
7FFDB068	. 00000000	DD 00000000	
7FFDB06C	. 00000000	DD 00000000	NtGlobalFlag = 0
7FFDB070	. 00809807	DD 079B8000	Reserved = 0
			CriticalSectionTimeout_Lo = 79B8000

Figure 15: Bypassed BeingDebugged and NtGlobalFlag values within the PEB

With checks for both PEB.BeingDebugged and PEB.NtGlobalFlag 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 won't 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](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](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](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*):

```

[RETURN to 01A707AE
ApplicationName = "C:\Users\REM\Desktop\vitamina_Packed.exe"
CommandLine = ""C:\Users\REM\Desktop\vitamina_Packed.exe""
pProcessSecurity = NULL
pThreadSecurity = NULL
InheritHandles = FALSE
CreationFlags = CREATE_SUSPENDED
pEnvironment = NULL
CurrentDirectory = NULL
pStartupInfo = 01C40048 -> STARTUPINFO {Size=0, Reserved1=NULL, Desktop=NULL, Title=NULL}
pProcessInformation = 01C4008C -> PROCESS_INFORMATION {hProcess=NULL, hThread=NULL, Pro

```

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):

Address	Hex dump
01C4008C	34 01 00 00 30 01 00 00 08 07 00 00 A8 05 00 00
01C4009C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

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):

Structure PROCESS_INFORMATION at 01C4008C			
Address	Hex dump	Decoded data	Comments
01C4008C	34 01 00 00	DD 00000134	hProcess = 00000134
01C40090	30 01 00 00	DD 00000130	hThread = 00000130
01C40094	08 07 00 00	DD 00000708	ProcessID = 708 (1800.)
01C40098	A8 05 00 00	DD 000005A8	ThreadID = 5A8

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.

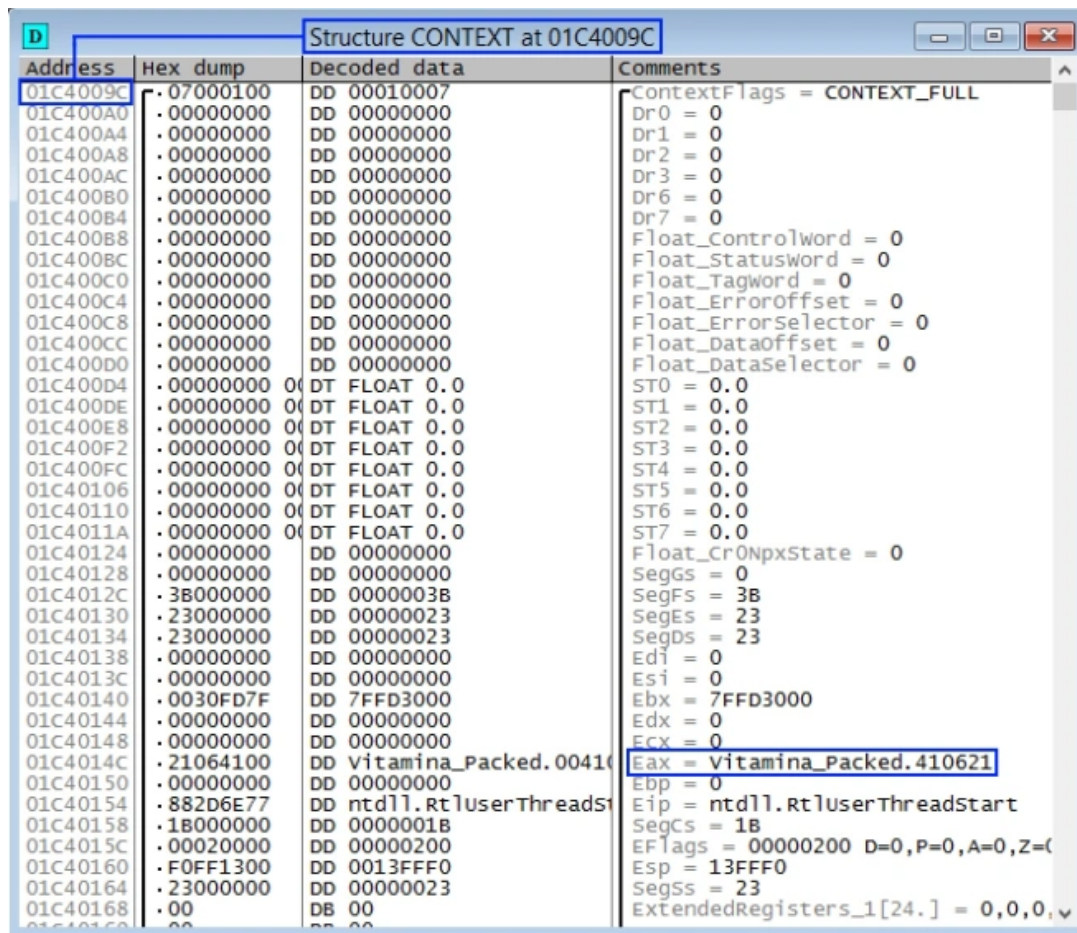
NtSetContextThread

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):



Address	Hex dump	Decoded data	Comments
01C4009C	.07000100	DD 00010007	ContextFlags = CONTEXT_FULL
01C400A0	.00000000	DD 00000000	Dr0 = 0
01C400A4	.00000000	DD 00000000	Dr1 = 0
01C400A8	.00000000	DD 00000000	Dr2 = 0
01C400AC	.00000000	DD 00000000	Dr3 = 0
01C400B0	.00000000	DD 00000000	Dr6 = 0
01C400B4	.00000000	DD 00000000	Dr7 = 0
01C400B8	.00000000	DD 00000000	Float_Controlword = 0
01C400BC	.00000000	DD 00000000	Float_Statusword = 0
01C400C0	.00000000	DD 00000000	Float_Tagword = 0
01C400C4	.00000000	DD 00000000	Float_ErrorOffset = 0
01C400C8	.00000000	DD 00000000	Float_ErrorSelector = 0
01C400CC	.00000000	DD 00000000	Float_DataOffset = 0
01C400D0	.00000000	DD 00000000	Float_DataSelector = 0
01C400D4	.00000000	DD 00000000	ST0 = 0.0
01C400DE	.00000000	DD 00000000	ST1 = 0.0
01C400E8	.00000000	DD 00000000	ST2 = 0.0
01C400F2	.00000000	DD 00000000	ST3 = 0.0
01C400FC	.00000000	DD 00000000	ST4 = 0.0
01C40106	.00000000	DD 00000000	ST5 = 0.0
01C40110	.00000000	DD 00000000	ST6 = 0.0
01C4011A	.00000000	DD 00000000	ST7 = 0.0
01C40124	.00000000	DD 00000000	Float_Cr0NpxState = 0
01C40128	.00000000	DD 00000000	SegGs = 0
01C4012C	.3B000000	DD 0000003B	SegFs = 3B
01C40130	.23000000	DD 00000023	SegEs = 23
01C40134	.23000000	DD 00000023	SegDs = 23
01C40138	.00000000	DD 00000000	Edi = 0
01C4013C	.00000000	DD 00000000	Esi = 0
01C40140	.0030FD7F	DD 7FFD3000	Ebx = 7FFD3000
01C40144	.00000000	DD 00000000	Edx = 0
01C40148	.00000000	DD 00000000	Ecx = 0
01C4014C	.21064100	DD Vitamina_Packed.00410621	Eax = Vitamina_Packed.410621
01C40150	.00000000	DD 00000000	Ebp = 0
01C40154	.882D6E77	DD ntdll.RtlUserThreadStart	Eip = ntdll.RtlUserThreadStart
01C40158	.1B000000	DD 0000001B	SegCs = 1B
01C4015C	.00020000	DD 00000200	EFlags = 00000200 D=0,P=0,A=0,Z=0
01C40160	.F0FF1300	DD 0013FFF0	Esp = 13FFF0
01C40164	.23000000	DD 00000023	SegSs = 23
01C40168	.00	DD 00	ExtendedRegisters_1[24.] = 0,0,0,0

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):

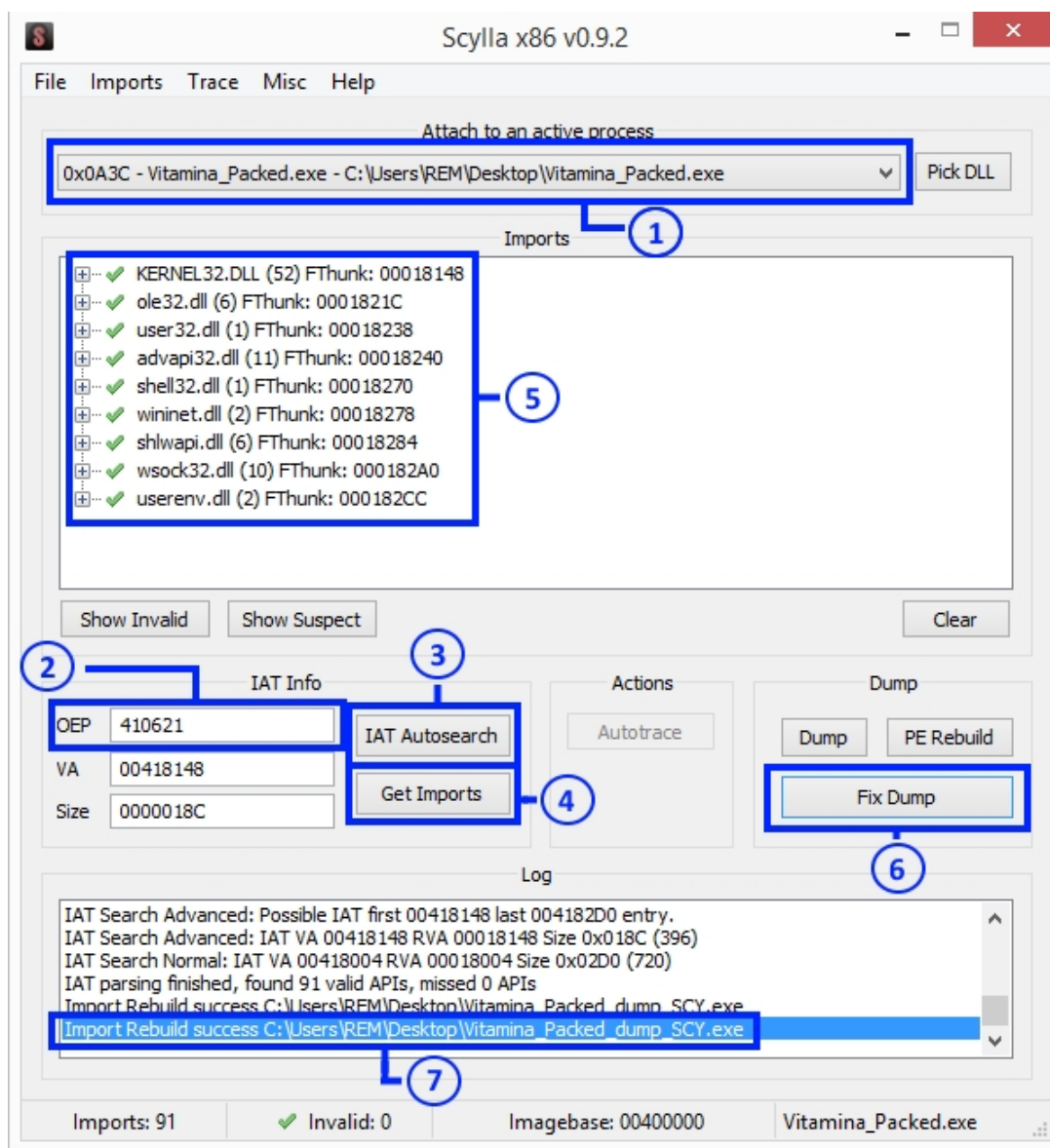


Figure 21: Fixing corrupt Import Address Table of dumped process using Scylla

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>)