## Analysing ISFB – The First Loader

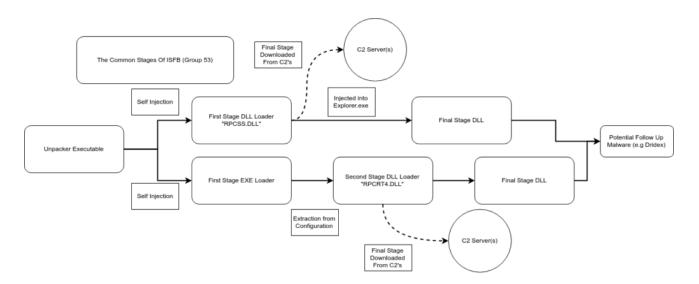
Offset.net/reverse-engineering/malware-analysis/analysing-isfb-loader

13 March 2019

- <u>overflow</u>\_
- 13th March 2019
- 2 Comments

I'm finally getting round to writing this post – for the past few months I have been analysing different versions of ISFB/Ursnif/Gozi to gain a deeper understanding in the functionality of this specific malware. In this post, I will be detailing how to unpack and then analyse the first stage loader executable, and then use that information to extract the second stage loader DLL, known as **rpcrt4.dll**, which we will analyse in a later post.

In a nutshell, ISFB is a banking trojan used to steal financial information from unsuspecting victims. It utilizes several methods to do so, from stealing saved passwords to injecting JavaScript into predetermined websites. This specific sample of ISFB is version 2.14.60, and can be attributed to a specific ISFB v2 group based on the infection routine used – specifically the macros that execute a powershell command that is simply Base64 encoded. The group behind this sample also reuse the encryption key for different campaigns (the default key), making their samples easily identifiable compared to other large groups utilizing ISFB. I have been unable to locate specific threat actor names, and as a result, I will be referring to this group as **Group 53**, based off of **this** presentation by FireEye.



Similarly to other groups utilizing ISFB for financial gain or simply as a stager, **Group 53** gains a foothold on the target's system using a malicious Word document containing embedded macros, which in turn lead to the execution of a Powershell script responsible for downloading the first stage executable. Certain groups "partner" with other groups that are

able to distribute malicious spam (malspam) on a large scale, such as the group behind Hancitor. This could potentially result in larger infection numbers, compared to those groups that are relying on their own distribution methods. I would assume "renting" a spot from the group behind Hancitor would be quite expensive as a result of its enormous outreach, which is why a lot of groups, including **Group 53**, have to distribute their own malicious documents. I will be focusing on the unpacker executable and the first stage executable loader in this post, rather than the Word Document itself, as its functionality is quite straightforward. As always, the samples have been uploaded to <u>VirusBay</u>. So, let's crack it open!

**But**, before I do I'd like to thank all of the people who helped me out in analysing the different samples of ISFB, including <u>@VK\_Intel</u>, <u>@Nazywam</u>, and <u>@Maciekkotowicz</u> (for his great papers on ISFB). Anyway, let's get on with the reversing!

## Part 1: Unpacking the first stage executable

```
MD5 of First Stage Executable: bc72604061732a9280edbe5e2c1db33b
```

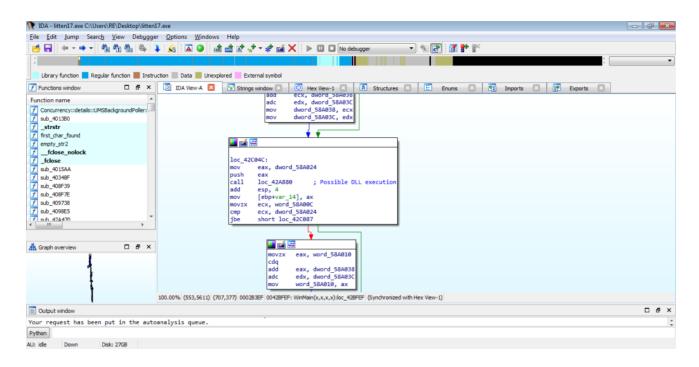
Typically I would open it up in PEStudio or perhaps perform some static analysis, however at this point I have already determined that it is highly likely to be a sample of ISFB, based on the Word Macro, although we still want to unpack the first EXE to be 100% sure that it is in fact ISFB. First, let's open it in IDA to try and find a call (or jmp) to a memory region or register – this could possibly be a call to the unpacked stage.



It definitely looks like there is some unpacking going on here (based on the length and intricate flow shown in the image above), and upon looking at the strings in the binary we can see that there aren't many that are legible, or meaningful. Normally when unpacking a

sample, I start at the bottom and work my way up – most unpackers exit once the file has been unpacked – although this depends. In this case, the unpacker performs **Self-Injection**, and overwrites itself with the unpacked file. This is not unusual for ISFB, and if you analyse some other samples (even from other groups), you will most likely find this occurring too. This means that the unpacker does not exit until the unpacked file does, although we can assume that the last function called will transfer execution over to the unpacked executable.

In this specific file, there is no call or jmp to a register or memory region – however, there is a call to **loc\_42A880**. It's the last call in WinMain, so as mentioned before, this function is most likely responsible for transferring execution over to the EXE.



When we jump into this function, it is clear that it hasn't been converted to a function yet, and so we are unable to view it as a graph. Sometimes, you are able to turn it into a function by pressing "P", although it doesn't seem to work in this case. So, we will have to deal with the text mode. It is definitely possible to follow the jumps and conditional jumps to try and find the call or jump to our executable, but we can speed this up a bit. Locate a **call** instruction and click it – this should highlight the instruction and other instances of it. This doesn't work for every sample, but it is worth a try to speed things up. Simply scroll down or up from where you are, looking for other instances of **call**. The last call I located took **lpAddress** as the first argument, so let's take a look at this. **lpAddress** indicates that it contains an address to a region of memory, meaning it could contain the address of our unpacked executable.

le Edit Jump Search View Debugge	r Options Windows Help			
🖆 🔚 🗢 🕶 🕶 🦛 🦓 🦓 🐧	* * * * 🏠 🙆 🥥 🛋 😥 🕴	🖬 🗙 🕨 💷	🔲 No debugger 🔹 🤹 💽 👔 🕈 😭	
				•
				·
Library function 📃 Regular function 📗 Instru	iction 📗 Data 📕 Unexplored 📒 External symb	loi		
Functions window 🛛 🗗 🗙	IDA View-A 🛛 🔂 Strings window	🛛 🚺 Hex Vie	-1 🗶 🖪 Structures 🗶 🗮 Enums 🗵 🛐 Imports 🗵 📝 Exports	X
inction name	.text:0042BBCA	push	eax	
	.text:00428BCB	mov	ecx, dword_58A028	
Concurrency::details::UMSBackgroundPoller:	.text:0042BBD1	push	ecx	
sub_401380	.text:0042BBD2	call	sub_42C900	
_strstr	.text:0042BBD7	add	esp, 8	
first_char_found	.text:0042BBDA	mov	edx, dword_58A024	
empty_str2	.text:0042BBE0	mov	eax, dword_58A028	
fclose_nolock	.text:0042BBE5	lea	ecx, [eax+edx-6]	
fclose	.text:0042BBE9	mov	dword_58A018, ecx	
sub_4015AA	.text:0042BBEF	movzx	edx, word ptr [ebp-24h]	
sub_40348F	.text:00428BF3	push	edx eax, lpAddress	
sub_408F39	.text:004288F4 .text:004288F9	mov	eax, <u>ipeddress</u> eax	
sub_408F7E	.text:0042BBFA	push call	sub 42A470	
	.text:004288FF	add	esp, 8	
sub_409738	.text:00428C02	xor	ecx, ecx	
sub_4098E5	.text:00428C04	mov	dword 58A038, eax	
sub_42A470	.text:00428C09	mov	dword 58A03C, ecx	
WinMain(x,x,x,x)	.text:0042BC0F	mov	ecx, dword 58A038	
sub_42C900	.text:00428C15	add	ecx, 3763h	
sub_42C9E0	.text:00428C18	mov	esi, dword 58A03C	
sub_42CE48	.text:0042BC21	adc	esi, 0	
sub_42CE4D	.text:0042BC24	mov	edx, dword 58A024	
sub 42CE64	.text:0042BC2A	xor	eax, eax	
operator delete(void *)	.text:0042BC2C	sub	ecx, edx	
	0002AFE9 0042BE9: .text:0042	BBE9 (Synchronized	with Hex View-1)	
	•		III.	
Output window				
ur request has been put in the auto	anaiysis queue.			

Inside this function, if we jump straight to the end, we can see a call to **ESI**. Hit space to jump from the Graph view back to Text view in order to get the address of this call. This could be the call to our unpacked EXE, and so we need the address of it to view it in x32dbg and put a breakpoint on it.

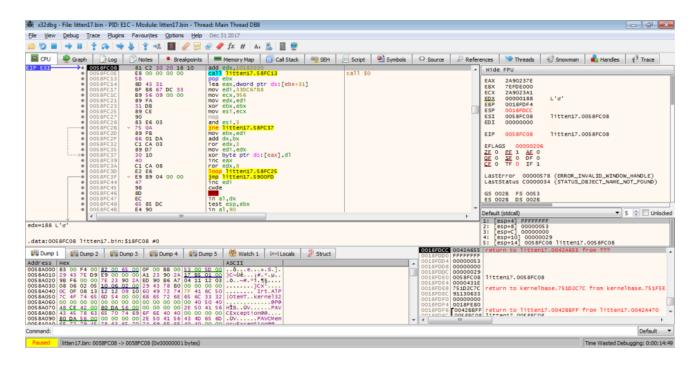
	1 🌯 🗣 🖡 🔏	🔺 🎱 📾 📾 🖬 🖑 - 🛠 🗃 🗡		No debugger 💽 🐑 🗊 🕈 🕅	 -
					-
Library function 📃 Regular fun	ction 📕 Instruction 📗	Data 📕 Unexplored 📒 External symbol			
Functions window	0 8 × [] 1	DA View-A 🛛 🔝 Strings window 🗵	Hex Vie	v-1 🗶 🖪 Structures 🗶 🗮 Enums 🗶 🛐 Imports 🗶 📝 Exports 🗶	
Function name	*	.text:0042A5F4	mov	edx, [ebp+arg_4]	
Concurrency::details::UMSBackgr	oundPoller:	.text:0042A5F7	add	edx, 3763h	
f sub_401380	oundroller.	.text:0042A5FD	sub	edx, dword_58A024	
		.text:0042A603	add	edx, fl0ldProtect	
		.text:0042A609	add	edx, fl0ldProtect	
f first_char_found		.text:0042A60F .text:0042A615	mov	floldProtect, edx eax, [ebp+arg 0]	
f empty_str2		.text:0042A615	mov	eax, [ebp+arg_0] [ebp+var 14], eax	
fclose_nolock		.text:0042A618	mov	ecx, dword 58A024	
ffclose		.text:0042A621	mov	edx, [ebp+arg_4]	
f sub_4015AA		.text:0042A624	lea	eax, [edx+ecx-6]	
📶 sub_40348F		.text:0042A628	mov	ecx, floldProtect	
f sub_408F39		.text:0042A62E	sub	ecx, eax	
f sub_408F7E		.text:0042A630	mov	floldProtect, ecx	
f sub_409738		.text:0042A636	mov	edx, [ebp+var 18]	
f sub_4098E5		.text:0042A639	mov	eax, dword_58A024	
f sub_42A470		.text:0042A63E	lea	ecx, [eax+edx-6]	
f WinMain(x,x,x,x)		.text:0042A642	mov	dword_58A018, ecx	
f sub_42C900		.text:0042A648	mov	esi, [ebp+var_14]	
		.text:0042A648	inc	edx	
f sub_42C9E0		.text:0042A64C	add	edx, 15Dh	
f_sub_42CE48		.text:0042A652	inc	edx	
f sub_42CE4D		.text:0042A653	call	esi ; Interesting Call	
f sub_42CE64		.text:0042A655	retn		
f operator delete(void *)	-	.text:0042A655 sub_42A470	endp		
	P.	000299FD 0042A5FD: sub_42A470+18	) (Synchroni	ed with Hex View-1)	
		•		m	Þ
Output window					
our request has been put :		1			

So the memory address to this call is **0x0042A653**, so let's now open this up in x32dbg and jump to this address. Simply push CTRL-G in x32dbg and type in **0042A653**, and then hit enter – this will jump to that address, allowing you to put a breakpoint on it. Whilst

attempting to unpack a sample, I prefer to open up Process Hacker alongside it, so that if I put a breakpoint on the wrong address and the unpacked process executes, I can easily detect it, either through the **Network** tab or through the **Processes** tab.

je View Debug Trace Plugins Favourites Options Help Dec 31 2017	Process Hacker [Reversing\RE]				l	- 0 💌
	Hacker View Tools Users He					
	🕉 Refresh 🎲 Options 🛛 🋗 Find	d handles or [	OLLs 😽 Syster	n information	Search Process	es (Ctrl+K)
🕮 CPU 🤹 Graph 📝 Log 🖄 Notes 🔹 Breakpoints 📟 Memory Map 🧊 Call Stack 🧠 SEH 🧕	Processes Services Network Disk					
<ul> <li>0042A5F7</li> <li>0042A5FD</li> <li>0042A5FD&lt;</li></ul>	Name	PID	CPU I/O tota	I Private b	User name	Descri 4
0042A603     03 15 1C A0 58 00     add edx,dword ptr ds: 58A01C     0042A609     03 15 1C A0 58 00     add edx,dword ptr ds: 58A01C	msiexec.exe	2972		9.55 MB		Winde
<ul> <li>0042A60F</li> <li>89 15 1C A0 58 00 0042A615</li> <li>88 45 08</li> <li>mov dword ptr ds:[58A01C],edx</li> <li>mov dword ptr ss:[ebp+8]</li> </ul>	Isass.exe	508		4.41 MB		Local
0042A618     89 45 EC     mov dword ptr sst ebp-14 eax	Ism.exe	516		2.38 MB		Local
0042A618     88 0D 24 A0 58 00     0042A621     88 55 0C     mov ecx,dword ptr ds:[58A024]     mov edx,dword ptr ss:[ebp+C]	CSTSS.exe	416	0.06	2.34 MB		Client
O042A624     SD 44 0A FA     lea eax,dword ptr ds:[edx+ecx-6]     mov ecx,dword ptr ds:[58A01C]	😭 winlogon.exe	480		2.78 MB		Winde
0042A62E     25 C8     sub ecx,eax     mov dword ptr ds:[58A01C],ecx	svchost.exe	756		4.39 MB		Host F
0042A636     88 55 E8     mov edx,dword ptr ss:[ebp-18]	4 🖭 svchost.exe	840		18.14 MB		Host F
O042A639     A1 24 A0 58 00     mov eax,dword ptr ds:[58A024]     lea ecx,dword ptr ds:[eax+edx-6]	audiodg.exe	824		15.43 MB	1	Windo
0042A642     89 00 <u>18 A0 58 00</u> mov dword ptr ds:[ <u>58A018</u> ],ecx     0042A648     88 75 EC mov esi,dword ptr ss:[ebp-14]	4 💷 svchost.exe	888		97.59 MB		Host F
0042A648     42     1nc edx     add edx,150	dwm.exe	1120		1.74 MB	Reversing\RE	Deskte
• 0042A652 42 inc edx	svchost.exe	924		22.13 MB		Host F
COLLASS FF D6 Call esi     O042A655 C3 ret	svchost.exe	332		6.51 MB		Host F
0042A656     88 15 24 A0 58 00     mov edx,dword ptr ds:[58A024]     imul edx,edx,50	svchost.exe	920	0.01	27.66 MB		Host F
0042A65F     03 55 0C     add edx,dword ptr ss:[ebp+C]     mov dword ptr ds:[58A01C],edx	spoolsv.exe	1172		6.08 MB		Spool
<pre>4 0042862] 89 15 16 A0 58 00 [mov dword per 05: [Seware], eax</pre>	svchost.exe	1220		11.35 MB		Host F
i=litten17.0058FC08	OfficeClickToRun.exe	1312		32.75 MB		Micro
	svchost.exe	1408		5.76 MB		Host F
text:0042A653 litten17.bin:\$2A653 #29A53	taskhost.exe	2008		8.02 MB	Reversing\RE	Host F
🗒 Dump 1 🚛 Dump 2 🚛 Dump 3 🚛 Dump 4 🚛 Dump 5 🛞 Watch 1 🛛 🕼 Locals 🖉 Struct	4 🧊 explorer.exe	1392	0.11	42.65 MB	Reversing\RE	Winde
Idress Hex ASCII	😼 VBoxTray.exe	1828	28	B/s 2.63 MB	Reversing\RE	Virtua
70C0000 88 44 24 04 CC C2 04 00 CC 90 C3 90 CC C3 90 90 .0\$.1Å1Å1Å 70C0010 90 90 90 90 90 90 90 90 90 90 90 90 90	⊿ 😹 x32dbg.exe	3452	2.88	43.64 MB	Reversing\RE	x64db
0C0020 88 4C 24 04 F6 41 04 06 74 05 E8 81 1D 01 00 88 .LS.8AL.E 0C0030 01 00 00 0C 2 10 00 90 80 84 24 DC 02 00 00 64ASUd	🗾 litten17.bin	3612	0.02	1.26 MB	Reversing\RE	Tuber
0C0040 8B 0D 00 00 00 00 BA 20 00 0C 77 89 08 89 50 04	👧 ida.exe	3688	0.18	151.72 MB	Reversing\RE	The Ir
0C0050 64 A3 00 00 00 00 58 8D 7C 24 0C FF D0 88 8F CC dfX. \$.ÿD1 0C0060 02 00 00 64 89 0D 00 00 00 00 6A 01 57 E8 6E FEdj.wenp	ProcessHacker.exe	4020	0.63	9.69 MB	Reversing\RE	Proce
0C0070 00 00 88 F0 56 E8 28 6E 03 00 E8 F8 C2 10 00 90ðvè+neøÅ 0C0080 64 88 00 30 00 00 08 84 9 10 F6 41 0A 08 75 15 d0I.öAu.	SearchIndexer.exe	2292		40.18 MB		Micro
0C0090 FF 74 24 0C FF 74 24 08 E8 95 06 09 00 B8 00 00 ÿt\$.ÿt\$.è						
mmand:		•				•
Paused INT3 breakpoint at litten 17.0042A653 (0042A653)	CPU Usage: 4.40% Physical memor	v: 1.2 GB (29.9	5%) Processes	37		

Once the breakpoint has been hit, step into **ESI** and in this sample there is what seems to be a second unpacking "stage" – this code was not here originally, in fact if we try and view it in IDA, you will simply see a lot of question marks and the variable **unk\_58FCo8**.



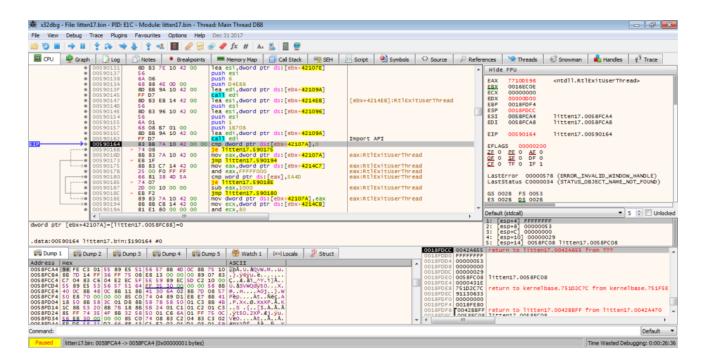
<u>Eile E</u> dit Jump Searc <u>h V</u> iew Deb <u>u</u> gger	Options Windows Help	
🛃 🖛 🕶 👻 🦓 🦓 🐴	😠 🔺 🖬 📾 🖈 🖈 🛪 📹 🗙 🕨 💷 🗖 No debugger 🔹 🔹 💽 🗊 😭 🎬	
		•
Library function 📃 Regular function 📗 Instruc	ion 📗 Data 📕 Unexplored 📃 External symbol	
Functions window	🗓 IDA View-A 🗵 🚺 Hex View-1 🛛 🖪 Structures 🛛 🗒 Enums 🗵 🛐 Imports 🖾	Exports 🗵
unction name	data:0058FC04 db ?;	
Concurrency::details::UMSBackgroundPoller:	.data:0058FC05 db ?;	
f sub_401380	.data:0058FC06 db ? ;	
f _strstr	<pre>.data:0058FC07</pre>	
f first_char_found	. data:0058FC00 db ? :	
empty_str2	data 0958FC0A db 2 :	
ffclose_nolock	data:0058FC08 db ?;	
f close	<pre>.data:0058FC0C db ?;</pre>	
sub_4015AA	db ?;	
sub_40348F	.data:0058FC0E db ?;	
f sub_408F39	data:0058FC0F db ?;	
	.data:0058FC10 db ?; .data:0058FC11 db ?;	
f sub_408F7E	data:0058FC12 db ?:	
f sub_409738	data:0058FC13 db ? :	
f sub_4098E5	data:0958FC14 db 2 :	
f_sub_42A470	• .data:0058FC15 db	
f WinMain(x,x,x,x)	• .data:0058FC16	
f_ sub_42C900	.data:0058FC17 db ?;	
f sub_42C9E0	.data:0058FC18 db ?;	
f_sub_42CE48	.data:0058FC19 db ?;	
f sub_42CE4D	.data:0058FC1A db ?;	
f sub_42CE64	data:0058FC18 db ?;	
operator delete(void *)	.data:0058FC1C db ?; .data:0058FC1D db ?;	
	data:0056FCID db r;	
	UNKNOWN 0058FC08: .data:unk_58FC08 (Synchronized with Hex View-1)	
Output window		_ <i>a</i> ×
ne initial autoanalysis has been fini	, , , , , , , , , , , , , , , , , , ,	1
ython	20154 -	,

As it is difficult to statically analyse this section without dumping it and opening it up in IDA, I will be jumping over functions, rather than stepping through them manually. This will also help to speed things up, but make sure your network adapter is not attached, as one of these functions may execute the executable.

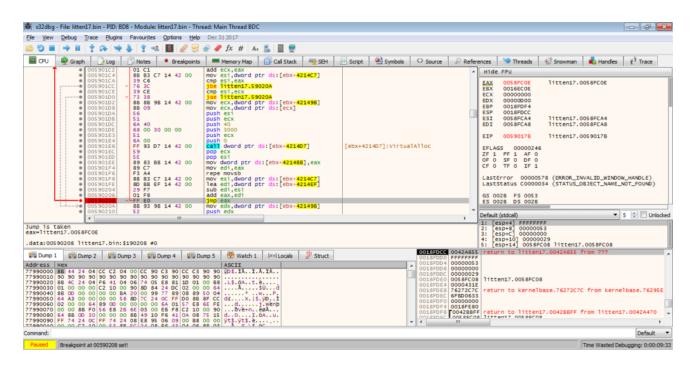
In the GIF below, you can see there is some form of loop going on, with XOR being used to XOR a byte at the memory address stored in EAX, with the value in DL. You might also notice that the assembly is changing as each loop goes on – this is another example of the second unpacking stage. We can simply put a breakpoint on the jmp and then run the program until it hits it.



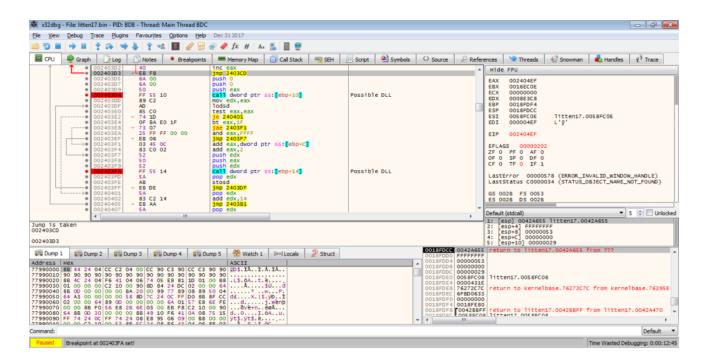
Once we follow the jump, we are met with several calls to **[ebx+xxxxx]**. Each of these could jump to the unpacked EXE, however as we progress further on, it is clear that these are simply calls to Windows API functions. Notice the call to **EDI**? EDI is pointing to a function that dynamically imports these APIs so that they can be called by the unpacking stub. The result is stored in EAX, and as seen in the image below, this specific call imported the API **RtlExitUserThread**.



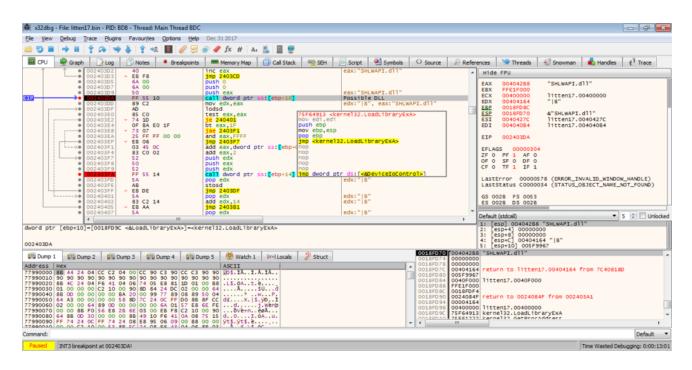
Scrolling down a bit, we can see a **jmp eax**, so let's put a breakpoint on that and run to it, and then follow the jump.

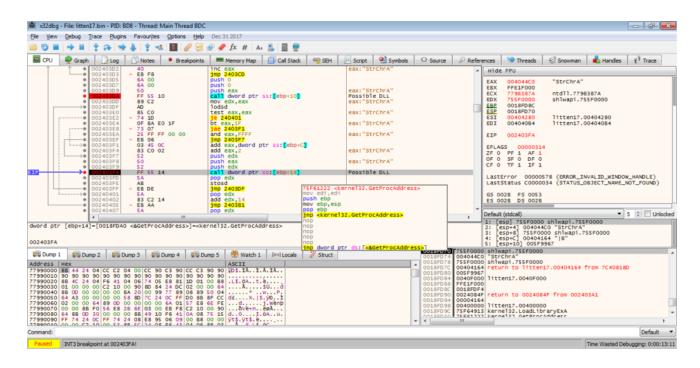


This jump takes us to a newly allocated region of memory, with more code. I didn't want to examine each function, so I scrolled down until the **ret** instruction, and started examining the (**local** – not API calls) functions backwards. The last function did not seem likely to be the executable "executor", as there was no call or jump instruction to a different region of memory, however the second last function had some calls to **[ebp+14]** and **[ebp+C]**, so let's put a few breakpoints on those.



However, upon executing the program and hitting the breakpoint, it is clear that they are simply calling **LoadLibrary** and **GetProcAddress**.





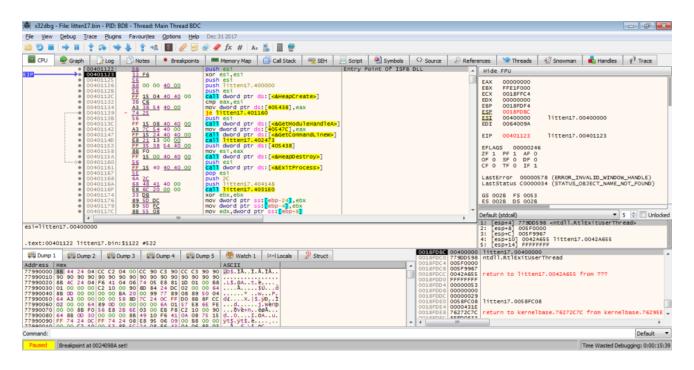
Therefore, we can simply return from this function, as it is just importing API's, which isn't extremely important to us yet. After exiting the function, there are several calls to API's, in particular **VirtualProtect**. **VirtualProtect** is responsible for changing the permissions/protection of different memory regions, so that they are readable, writable, or executable (can be one, two, or all three). In this case, we can see that **VirtualProtect** is being used to change the protection of the memory region at **0x00406000**, which is the **.BSS** section. As I mentioned before, this unpacker overwrites itself with the unpacked executable, so it is safe to say that we are extremely close to the unpacked EXE. We could dump it now, but there could be some extra unpacking going on, so let's wait until we jump to the **0x00400000** memory region.

豪 x32dbg - File: litten17.bin - PID: BD8 - Thread: Main Thread BDC	
File View Debug Trace Plugins Favourites Options Help Dec 31 2017	
a 🗊 🖬   🕆 🗛   👳 🎍   🛊 🕫 📓 🖉 🖉 🖉 🌮 🏌 #   🗛 🎉   🗒 💇	
🗱 CPU 👷 Graph 🔂 Log 🗋 Notes 🔹 Breakpoints 🛲 Memory Map 🚺 Call Stack 🧠 SEH 💿 Script 🐏 Symbols 🗘 Source	🔎 References 🖙 Threads 🕄 Snowman 🔒 Handles 🦸 Trace
Image: Control of the set of the se	Hide FPU           EX         00406000         litten17.00406000           EX         FFE1F000           EX         00406000         litten17.00406000           EX         FFE1F000           EX         0001000           EX         0001000           EX         0001100           EX         0001100           EX         0001100           EX         0001100           EX         0001100           EX         0000020           EX         00000004           EX         00000004           EX         00000246           ZF         0000034           LastError         0000034 (STATUS_OBJECT_NAME_NOT_FOUND)           GS         0028         FS           Default(dtal)          S © Unlocks
word ptr [ebx+4214DB]=[002404DB <&VirtualProtect>]= <kernel32.virtualprotect></kernel32.virtualprotect>	2: [esp+4] 00001000 3: [esp+8] 00000004 4: [esp+6] 00000004
002408F7	5: [esp+10] 00000003
Constraint         Constra	00456000 11tten17.00406000 00001000 00001000 00579957 00424655 return to 11tten17.00424655 from 777 04244555 return to 11tten17.00424655 from 777 000000000 000000000 11tten17.00424655 from 777
	Default

At this point, there isn't much going on in the function, so we can simply put a breakpoint on the **ret** instruction and return from this function.

* x32dbg - File: litten17.bin - PID: BD8 - Thread: Main Thread BDC								- 6 <b>X</b>
	ec 31 2017							
	🐙 J% # A1 👞 🔛 💇					1 - 0		
		Script Symbols	Source	P Refere	nces 🔄 😒 Th	rreads 🛛 🐨 Snowman	💼 Handles	₹ <sup>7</sup> Trace
0 0240948         89         48         1C         mm           0 0240948         80         85         7A         10         20         00           0 0240954         80         85         7A         10         20         00           0 0240957         80         85         16         11         42         00         mm           0 0240950         80         85         87         7A         10         42         00         mm           0 0240950         80         85         7A         10         42         00         mm         02           0 0240950         80         85         7A         10         42         00         mm         02         409         00 <td< th=""><th></th><th>[ebx+4214E8]:RtlExit</th><th>User Thread</th><th></th><th>ZF 1 PF 1 OF 0 SF 0 CF 0 TF 0 LastError LastStatus GS 0028 F ES 0028 D Default (stdcal)</th><th>F000 FFC4 0000 FFC4 FDF4 FDF4 FDF4 FDF4 FDF4 FDF4 III23 Iitten17 000246 AF0 DF0 IF1 00000578 (ERROR_ C0000034 (STATUS S 0028 S 0028</th><th>_OBJECT_NAME_I</th><th>NOT_FOUND)</th></td<>		[ebx+4214E8]:RtlExit	User Thread		ZF 1 PF 1 OF 0 SF 0 CF 0 TF 0 LastError LastStatus GS 0028 F ES 0028 D Default (stdcal)	F000 FFC4 0000 FFC4 FDF4 FDF4 FDF4 FDF4 FDF4 FDF4 III23 Iitten17 000246 AF0 DF0 IF1 00000578 (ERROR_ C0000034 (STATUS S 0028 S 0028	_OBJECT_NAME_I	NOT_FOUND)
0024098A					2: [esp+8] 3: [esp+C] 4: [esp+10]	005F0000 005F9967 0042A655 litten: FFFFFFF		eau>
Unite         Unit         Unite         Unite <thu< th=""><th>ASCII 90 DS.1A.1A.1A.1A. 90 DS.1A.1A.1A. 64</th><th>• •</th><th>0018FDC0 0018FDC4 0018FDC8 0018FDC0 0018FDD0 0018FDD4 0018FDD4 0018FDD2 0018FDDC 0018FDE0 0018FDE4</th><th>7790D598 005F0000 005F9967 0042A655 FFFFFFFF 00000053 00000000 00000029 0058FC08 0000431E 76272C7C</th><th>return to litten17.0 return to</th><th>xitUserThread</th><th></th><th>1base. 76295E</th></thu<>	ASCII 90 DS.1A.1A.1A.1A. 90 DS.1A.1A.1A. 64	• •	0018FDC0 0018FDC4 0018FDC8 0018FDC0 0018FDD0 0018FDD4 0018FDD4 0018FDD2 0018FDDC 0018FDE0 0018FDE4	7790D598 005F0000 005F9967 0042A655 FFFFFFFF 00000053 00000000 00000029 0058FC08 0000431E 76272C7C	return to litten17.0 return to	xitUserThread		1base. 76295E
Command:								Default
Paused Breakpoint at 0024098A set!							Time Wasted De	bugging: 0:00:15:2

Upon exiting the function, you'll notice the memory address is in the **0x00401000** region. This is the unpacked program! Now we can dump it out, so make sure you have Process Hacker open, although you can dump it from x32dbg.



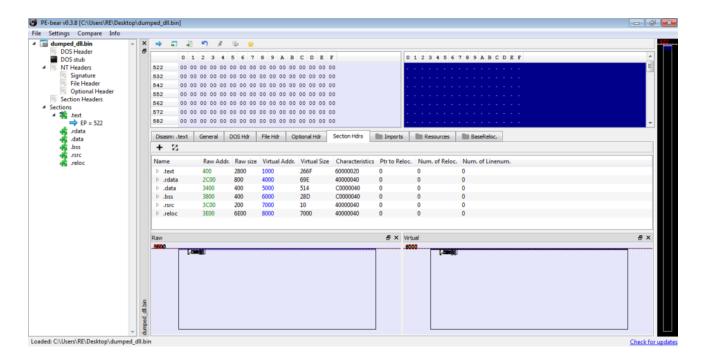
To dump it from memory, simply double click the process in Process Hacker, and right click the **0x400000** region of memory, and select **Save**.

🛠 x32dbg - File: litten17.bin - PID: BD8	- Thread: Main Thread	BDC			Process Hacker [Re	eversing\RE]						X
File <u>View D</u> ebug <u>Trace</u> Plugins	Favourites Options H	Help Dec 31 2	017		Hacker View Too	le Heln						
😑 🔁 🖬 🔶 🖩 🍷 🔿 🐋 -	🖌 🛊 🦗 📓 🦉	😑 🛷 🥒 f	x # 🗛 👗 🗐 朢		S Refresh (2) Opti	and the second sec	adles or		Curtan infe	omation	* Search Proces	ses (Ctrl+K)
🕮 CPU 🧠 Graph 🔀 Log	Notes Breakpo	ints Me	1	SE SEH	O Processes Condona		ndies or	ULLS 🕺	* System into	ormation	Search Proces	ses (Ctri+K)
● 00401122 00401123	🔟 litten17.bin (3032) Pr	operties				- • ×	PID	CPU	I/O total	Private b	User name	Descri *
Ø 00401125		•	A Los Los As Maria				456			2.62 MB		Winde
<pre> 00401126 00401128 </pre>	General Statistics Per	rformance   Thre	eads Token Modules Memor	Environm	ent Handles Comment		756			3.46 MB		Host
0040112C     00401132	Hide free regions				Strings	Refresh	836			17.85 MB		
00401134	Inde free regions				Surigs	Reliesit						Host F
• 00401139 • 0040113B	Base address	Type	Size Protect	Use		Total W ^	896			72.53 MB		Host F
0040113C     00401142	≥ 0x250000	Private	512 kB RW	Heap (ID	n	24 k	164				Reversing\RE	Deskti
00401147	> 0x2d0000	Private	256 kB RW	Heap (10 Heap 32-ł		12 k	924			17.96 MB		Host F
0040114D     00401152	> 0x310000	Private	512 kB RW	Heap and	AL (10 0)	4k	368			5.8 MB		Host F
00401158	> 0x3d0000	Private	64 kB RW	Heap 32-ł	vit (TD 5)	24 k	300			12.66 MB		Host F
● 0040115A >● 00401160	≥ 0x400000	Image	1 052 KR WCV		PE\Desktop\litten17.bin	1,676 k	196			6.2 MB		Spool
00401161 00401167	≥ 0x680000	Private	Read/Write memory		(ID 1)	220 k	232			11.58 MB		Host F
00401168	≥ 0x780000	Mapped	Save		100 17	312 k =	384			17.82 MB		Micro
0040116A     0040116F	> 0x8c0000	Private			t (ID 3)	12 k	492			5.13 MB		Host F
00401174	> 0x8f0000	Private	Change protection		(ID 2)	12 k	572			7.79 MB	Reversing\RE	Host F
00401176     00401179	> 0x900000	Mapped	Free		1	24 k	012			6.96 MB	nererangine	Micro
Ø040117C	≥ 0xa90000	Mapped	Decommit			56 k	188	0.05			Reversing\RE	Windo
·	≥ 0xc20000	Mapped				32 k						
si=litten17.00400000	> 0x21d0000	Private	Read/Write address		t (ID 4)	41	788	0.17			Reversing\RE	The Ir
	> 0x70600000	Image	Сору	Ctrl+C	s\SysWOW64\pleacc.dl	441	964	2.46	124 B/s		Reversing\RE	x64db <sup>⊞</sup>
text:00401122 litten17.bin:	▷ 0x70640000	Image	Copy "Type"		s\SysWOW64\msimg32.dll	16 k	032	0.02			Reversing\RE	Tuber
💭 Dump 1 🛛 💭 Dump 2 💭 D	≥ 0x70650000	Image	сору туре		s\winsxs\x86_microsoft	96 k	264	0.37		10.67 MB	Reversing\RE	Proce
ddress Hex	▷ 0x707e0000	Image	324 kB WCX	C:\Windo	ws\SysWOW64\winspool.drv	80 k	608	0.01		28.3 MB		Micro
7990000 88 44 24 04 CC C2 C 7990010 90 90 90 90 90 90 90 9	▷ 0x70840000	Image	92 kB WCX	C:\Windo	ws\SysWOW64\userenv.dl	32 k	500			2.78 MB		.NET F
7990020 8B 4C 24 04 F6 41 0	≥ 0x70860000	Image	484 kB WCX	C:\Windo	vs\SysWOW64\mscms.dl	36 k	500			4.14 MB		.NET F
7990030 01 00 00 00 C2 10 0 7990040 88 00 00 00 00 00 8	▷ 0x708e0000	Image	528 kB WCX	C:\Windo	vs\winsxs\x86_microsoft	68 k	936			62.34 MB		Host F
7990050 64 A3 00 00 00 00 5	▷ 0x70bb0000	Image	76 kB WCX	C:\Windo	ws\SysWOW64\dwmapi.dl	36 k	340			8.36 MB		WMIE
7990060 02 00 00 64 89 00 0 7990070 00 00 88 F0 56 E8 2	▷ 0x70bd0000	Image	512 kB WCX	C:\Windo	ws\SysWOW64\uxtheme.dll	80 k 🖛	380			928 kB		Host
7990080 64 88 0D 30 00 00 0	≥ 0x70d50000	٠	"			+	380			928 KB		Host
7990090 FF 74 24 0C FF 74 2												
ommand:												,
Paused Breakpoint at 0024098A s						Close	(24.98	3%) Pro	cesses: 36			

Now we can open it up in PEBear and unmap the dumped file. Upon doing so, you can see that the imports have not been successfully resolved by PEBear. This is why we need to unmap the file. When a program is about to be executed, it needs to be mapped in memory, so that it can be interpreted correctly. Therefore, PEBear is unable to resolve the imports until we unmap it.

Settings Compare Info dumped_dll.bin	~ ×	c 🛶	5		5	3	R.	, <u>,</u>																				
DOS Header	5	P	r	0 1	2	3 4	5	6	78	9	АВ	с	D	EF						0 1	2 3	4 5	6789	AB	CDEF			Ā
NT Headers		522	c	00 00	00	00 0	0 00	00 0	00 00	00 0	00 00	00 0	00 0	0 00	0													1
Signature		532	c	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	0 00	0													П
File Header		542	0	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	0 00	0													
Optional Header		552	0	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	00 00	0													
Section Headers Sections		562	0	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	00 00	0													
4 🗱 .text		572	0	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	00 00	0													
= EP = 522		582	0	00 00	00	00 0	0 00	00 0	0 00	00 0	00 00	00	00 0	0 00	0													-
🚓 .rdata					-				Le	1					-		4 a [	-	mports		Reso			seReloc				-
-data			sm: .te		Gene	eral	DO	S Hdr		ile Hdr		ptor	al Hdr		Sect	tion He	drs	_	mports		Reso	ources		serieloc				4
.bss		~	+	Ð																								4
rsrc .rsrc		Offs	et	1	Vame			Fur	nc. Co	ount	1	Boun	d?		(	Origi	nalFirs	tThu	Tim	eDate!	Stamp	Forv	varder	N	ameRVA	FirstThunk		•
		2D6						0			F	ALSE			8	83037	200		FE81	01C0		500		α	830372	80FE8101		1
		2D7	J .					0			F	ALSE			7	73000	000		2008	3303		4C88	BC085	14	741824	D62BD18B		
		2D8						0				ALSE				11881				C183		F075			244C89	28247489		
		2DA						0				ALSE				20245				4548B		1424		18		8A14EB00		
		2DB						0				ALSE				1C683			1424				0188		FF3301	8318244C		
		2DC						0				ALSE				10247			FEOE			5E5F			2BC18B	5B5D2424		
		2DD 2DF						0				ALSE				C2140	C483 IABAB		8B55	0008 088A8			83EC		575653 8DF633	D87D8DC0 458BABF0		
		201						0				ALSE				ABAB			1000		•		0000		004590	406BABPU	-	
		Detai																										ā.
		-																										-
		-																										4
																												12
	4 turnood dl hin																											
	-	<u>.</u>																										

To do so, we simply change the **Raw Addr.** so that it matches the **Virtual Addr.**, and then change the **Raw Size** accordingly. This should result in something looking like the image below.



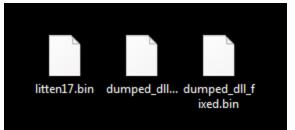
We can check the imports and sure enough, there are 4 imported DLL's, meaning we have the correctly unmapped file. We can now save this to the desktop and congratulations! You have now successfully unpacked the first stage! Let's open it up in IDA for further analysis!

le Settings Compare Info	_^ X	⇒ ₽	J 🔊 🌶	B 🖕							
DOS Header	6			5 6 7 8 9 A	BCDEF		0 1 2 3	456789	ABCDEF		*
<ul> <li>▲ NT Headers</li> <li>Signature</li> <li>File Header</li> <li>Optional Header</li> <li>Section Headers</li> <li>Section S</li> <li>▲ text</li> <li>➡ EP = 1122</li> <li>r.rdata</li> </ul>		532 0 542 0 552 0 562 0 572 0	0 00 00 00 00 00 0 00 00 00 00 0 00 00 0	00         00<	0         00         00         00         00         00           0         00         00         00         00         00         00           0         00         00         00         00         00         00         00           0         00         00         00         00         00         00         00           0         00         00         00         00         00         00         00		Imports In Res		seReloc.		•
.data .bss		·· +		Costia Theria	option ar har a	recourtmenta a			ACREME.		
.rsrc .reloc		Offset	Name	Func. Count	Bound?	OriginalFirstT	hun TimeDateStamp	Forwarder	NameRVA	FirstThunk	
.reloc		4164	SHLWAPLdII	2	FALSE	427C	0	0	42B8	40B4	
		4178	KERNEL32.dll	44	FALSE	4108	0	0	440C	4000	
		418C	USER32.dll	1	FALSE	4288	0	0	4400	40C0	
		41A0	ntdll.dll	9	FALSE	4290	0	0	44A8	40C8	
		Details									
		Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint			
		40B4	StrChrA		44BE	755FC5E6		10F			
		4088	StrRChrA		44B2	755FCCF5		136			
	4 dumped dl.bin										

## Part 2: Analysing the Dumped Executable

MD5 of Dumped Executable: 0063316975e55c765cd12e3d91820478

Upon opening the file in IDA, we can see that the main function only calls one local function and then exits, so it's not too difficult to find the malicious code. If you're not sure if a certain sample is ISFB or not, one telltale sign can be found in the strings window. Most, if not all, ISFB payloads (version 2 – haven't taken a look at version 3 yet so not sure) store a compile/campaign start date in plaintext that is used for string decryption. In this sample, the date is **Jan 28 2019**, so a relatively new sample at the time of writing this. I will go over the string decryption method soon, but first let's take a look at what happens first in the function.

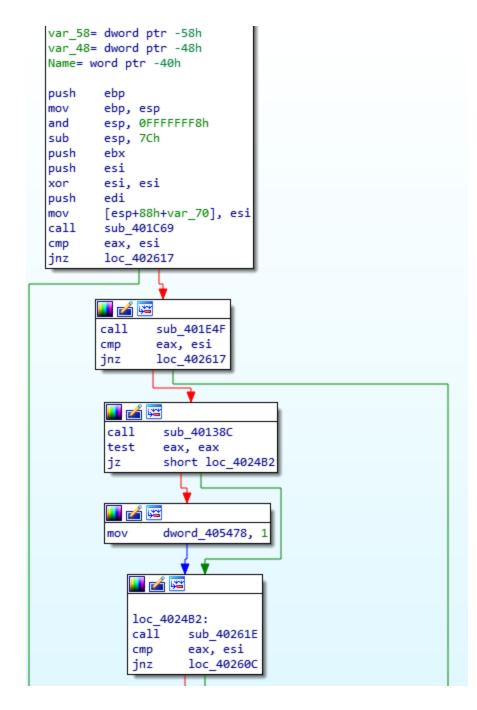


🛉 🔚 🗇 🖛 🕶 🖓	6 6 6 4	k   🔬   🗛   🥥	ं जी की बि	* + + *	🕨 📾 🗙 🕨 🖬 🖬 No debugger 🔹 🐮 💽 🗊 🚏 😭	
•						 -
Library function 📃 Regula				External sy		
Functions window	0 & ×	IDA View-A	🗵 🖹 S	trings windo		
inction name	^	Address	Length	Туре	String	
sub_401000		🖬 .rdata:0040	0000000C	с	Jan 28 2019	
sub_40106E	-	s .rdata:0040	0000009	C	AÜvõQkóxa	
start		rdata:0040	000000C	C	SHLWAPI.dl	
sub_401168		s' .rdata:0040	0000000D	C	KERNEL32.dll	
sub_4012BA		rdata:0040	00000008	C	USER32.dl	
sub_4012CF		s' .rdata:0040	A000000A	С	ntdi.di	
RtlUnwind		.data:00405	. 00000012	C	ÄSUVWATAUAVAWH, FIH	
sub_40133C		.data:00405	. 00000005	C	Q03ÛH	
sub 40138C		.data:00405	. 00000005	С	пн;бн	
sub_4013E1		's' .data:00405.	. 00000005	С	L@ <h< td=""><td></td></h<>	
sub_4014FF		data:00405	. 00000005	С	Ð:Ãuç	
sub_401558		.data:00405	00000007	С	F:3Ò3ÉM	
sub 401570		s' .data:00405	00000009	С	úe≢Htúr\rA	
aub 4015E1	*	's' .data:00405	00000005	С	ÂH;or	
III	•	's' .data:00405	00000006	c	Hath	
		s .data:00405.		C	VbH:0D	
	0 8 ×	's' .data:00405		c	:Äl2H	
raph overview		's' .data:00405		č	rAbi	
		s .data:00405		c	1/#Ubriút	
		s' .data:00405		c	EHFÅ	
		data:00405		č	V8:ÃL	
		s' .data:00405		č	ľ<Ã	
			. 0000003	<u> </u>		 _
		Line 1 of 50				
utput window						8
and "JumpEnter" fai	lad					

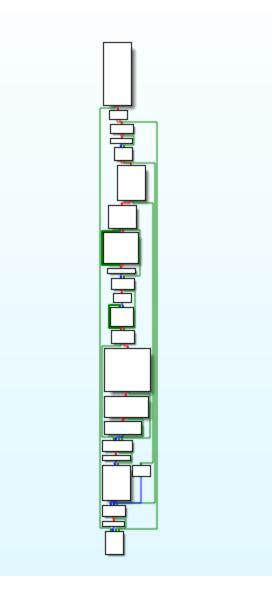
Strings

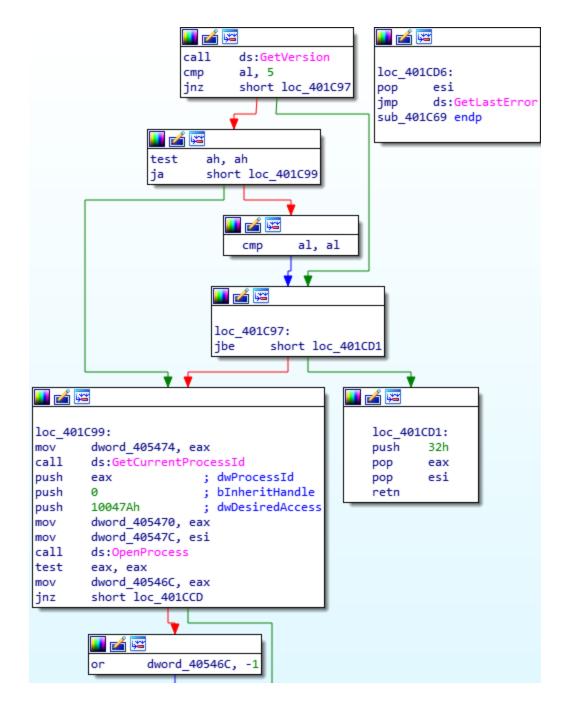
There are several functions called inside this function, so let's take it section by section, so first let's take a look at the first four functions. From the image below, it is clear that the return value of **sub\_401C69** (stored in **eax**) needs to match the value in **esi**, otherwise it will jump to the exit. The second called function – **sub\_401E4F** – seems to do the same thing. The third function seems to be a check for something, as **1** is moved into a DWORD based on the result of a bit-wise AND (**test** performs a bit-wise AND on the two values, however it just sets flags based on the result, which is not stored) on **eax**. Finally, the fourth function seems to the value in **esi**, and it will exit if the conditions are not met. Anyway, enough assuming, let's actually take a look at these functions.

🚺 🗹 🗟	*	
; Attri	butes: noretur	n
public		
	roc near esi	
xor	esi, esi	
	esi	; dwMaximumSize
push	400000h	; dwInitialSize
push	esi	; flOptions
call	ds:HeapCreate	,
cmp	eax, esi	
	hHeap, eax	
jz	short loc_401	160
	· · · · · · · · · · · · · · · · · · ·	
🚺 🚄 📓		
push	esi	; lpModuleName
call	ds:GetModuleH	
mov	dword_40547C,	
call	ds:GetCommand	LineW
call	sub_402473	1.0
	hHeap	; hHeap
mov	esi, eax	
call	ds:HeapDestro	у
<b>II</b> 2	í 🖼	·
loc 4	401160:	; uExitCode
	esi	
push		
push call	ds:ExitProc	ess
call	ds:ExitProc t endp	ess
call		ess
call		ess

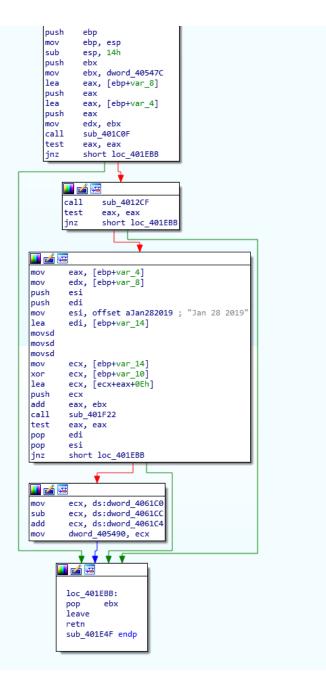


Taking a look at the first function (the main bit), we can see that the malware is opening it's own process and storing the handle in a DWORD, which is set to **-1** if the malware failed to open the process. This then returns back to the main function, where the returned result is compared against the value in **ESI**.



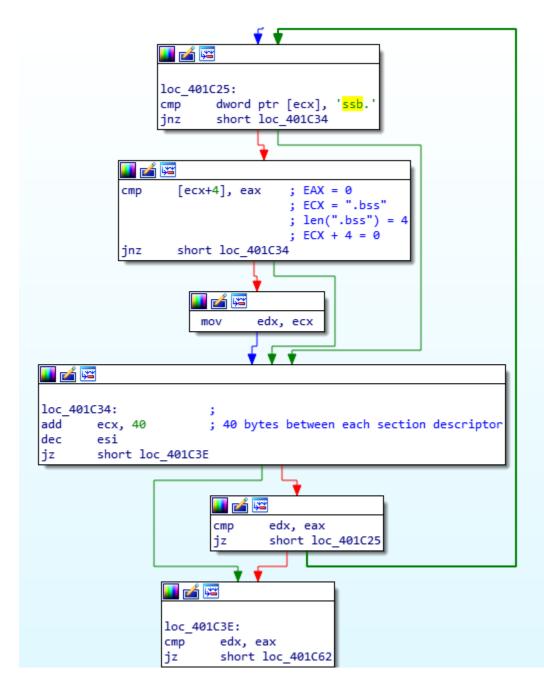


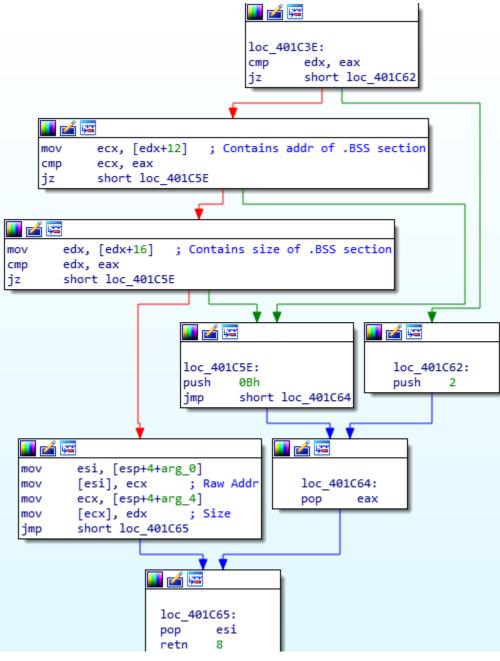
The second function is where things start to get interesting. In the image below, you'll notice the compile date being moved into **ESI**, which, as I mentioned before, is used for string decryption. You might already know this, but for those that don't, ISFB contains a **.BSS** section, which contains multiple strings that are all encrypted using a ROR-XOR algorithm. The XOR key is calculated based on a given date, and in order to decrypt the strings, ISFB needs to perform the calculations again to get the correct key, allowing us to easily reverse it. But first, let's take a look at the two functions called beforehand.



Looking at the first function, you'll notice **ssb.** (endian issues, as it is stored as hex – actually is **.bss**) being compared to the value at **[ECX]**, inside a loop. In order to decrypt the strings in the .BSS section, ISFB must first locate the .BSS section, and in order to do so, it simply reads it's own PE header and gets the size and address of the required section. If the value at **[ECX]** doesn't match **.bss**, **40** is added the the memory address in **ECX**, due to the fact that the spacing between the section descriptors/structures (**.text**, **.data**, etc.) is 40 bytes. The loop will the continue. If there is a match, the length of the string is checked, making sure that it is not longer than 4 bytes. The malware does this by checking the byte after the string, and comparing it to zero. If the string is the correct length, the memory address pointing to "**.bss**" is moved into **EDX**. If everything is successful, ISFB will get the address and size of the .BSS section and store it in memory. In this case, the address is **0x6000** (add this to the image base and you will be able to locate it), and the size is

**0x1000**. If this is still difficult to understand, there is an image of what the section table looks like in x32dbg – this should help you to understand how it is able to get the address and size.





Function: sub\_401CoF #2

📴 CPU	👰 Graph 🛛 🗋 Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source	P Refer	ences	😒 Threads	🖅 Snowman	🔒 Handles	f <sup>7</sup> Trace
dword ptr	000000000000000000000000000000000000	<ul> <li>75 07</li> <li>39 41 04</li> <li>75 02</li> <li>88 D1</li> <li>83 C1 28</li> </ul>	14 16 18 62 73 73 4 3 11	mov ecx, dword p add ecx, edx movzx edx, word push esi movzx esi, word novzx esi, word tea ecx, dword p tr dived, edx cmp dword ptr d ime fixed.dwmp, cmp dword ptr d ime fixed.dwmp, mov edx, ecx add ecx, 28	ptr ds:[ecx+4 ptr ds:[ecx+4 tr ds:[edx+ec s:[ecx],73736 401C34 s:[ecx+4],ea	14] 5] 5x+18] 522E	ecx:".b: ecx:".b: ecx:".b: ecx:".b: ecx:".b:	:5" :5" :5"		-	EBX ECX EDX EBP ESP ESI EDI Default (: 1: [es 2: [es 3: [es	00000000 00400000 00400270 00018FEF0 0018FEC8 00000003 00000000 00404C20 etdcall) p+4] 00401 p+6] 0018F	EE8	0040153D	5 🔄 🕅 Unlocked
.text:004	01C25 fixed_dump.		25 Dump 4 🛛 💷 Dur	mp 5 💮 Watch 1	[x=] Locals	3 Struct			0018FEC8	00000000	5: [es	p+10] 7EFD p+14] 0018	0E000 8FF80 Ldump.00401E66	from fixed	dump, 004010
004001A0 004001E0 004001E0 004001E0 004001E0 00400210 00400210 00400220 00400220 00400220 00400220 00400240 00400250 00400250 00400280 00400280 00400280 00400280 00400280 00400280 00400280	Hex         Hex           00         0	0         0         0         0         0           0         0         0         0         0         0           0	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	00 00 00					<ul> <li>0018FE04</li> <li>0018FE05</li> <li>0018FE05</li> <li>0018FE05</li> <li>0018FE05</li> <li>0018FE04</li> <li>0018FE04</li> <li>0018FE04</li> <li>0018FE04</li> <li>0018FF04</li> <li>0018FF04</li> <li>0018FF04</li> <li>0018FF14</li> <li>0018FF14&lt;</li></ul>	0000000 0000000 7EFDE00 7FFDEFF 0007F00 0000000 0000000 0000000 0000000	return		dump.00401C80		
Command:															Default 🔻

With that function analysed, let's move onto the next one. The main purpose of this function is to add a vectored exception handler using **AddVectoredExceptionHandler**. The second argument is a pointer to the handler function that will be executed when the program runs into an exception, so lets take a look at the function

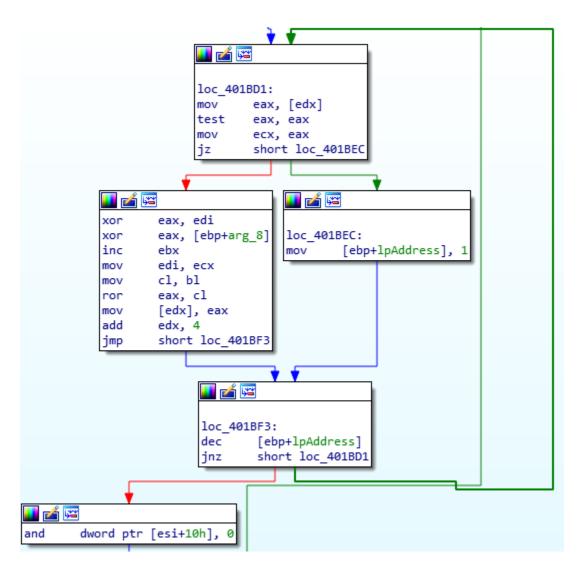
## Exception\_Handle\_Function.q

```
signed int sub_4012CF()
{
 struct _RTL_CRITICAL_SECTION *v0; // eax
 struct _RTL_CRITICAL_SECTION *Critical_Section; // esi
 DWORD v2; // eax
 int v3; // eax
 DWORD v4; // edi
 v0 = (struct _RTL_CRITICAL_SECTION *)Allocate_Heap(0x28u);
 Critical Section = v0;
 if ( !v0)
   return 8;
 v0[1].LockCount = (LONG)&v0[1];
 v0[1].DebugInfo = (PRTL_CRITICAL_SECTION_DEBUG)&v0[1];
 InitializeCriticalSection(v0);
 v2 = TlsAlloc();
 Critical_Section[1].OwningThread = (HANDLE)v2;
 if ( v2 != -1
   & (v3 = AddVectoredExceptionHandler(1, Exception_Handle_Function), (Critical_Section[1].RecursionCount = v3) != 0) )
 {
   lpCriticalSection = Critical_Section;
   v4 = 0;
 }
 else
 {
   v4 = GetLastError();
   if ( v4 )
     Remove_Exception_And_VirtualProtect(Critical_Section);
 return v4;
}
```

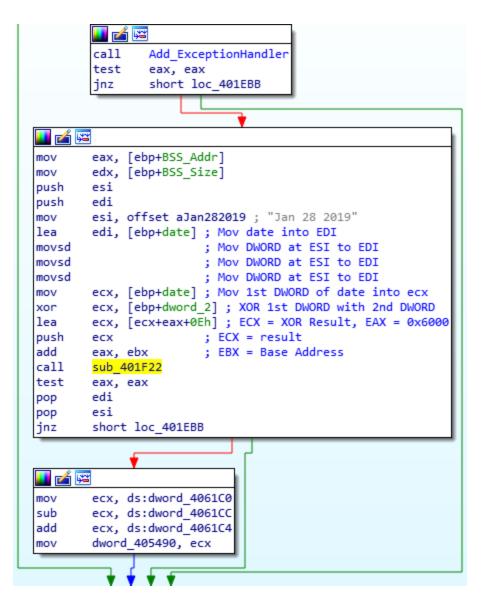
This function checks the value of the exception – whether it is an **EXCEPTION\_ACCESS\_VIOLATION** or **EXCEPTION\_SINGLE\_STEP**, however they both end up executing the same function, so lets move into that function.

```
signed int stdcall Exception Handle Function(int **a1)
{
  int v1; // ebp
 LPCRITICAL SECTION v2; // ebx
 int *v3; // ecx
 int v4; // edx
  void *v5; // edi
 DWORD *v6; // esi
 void *v7; // eax
 v1 = 0;
 v2 = lpCriticalSection;
 if ( lpCriticalSection )
  ł
    v3 = *a1:
   v4 = **a1:
    if ( v4 == 0xC0000005 )
    {
      v5 = (void *)v3[6];
      if ( !sub_401B36(lpCriticalSection, (LPVOID)v3[6], 1) )
      {
        TlsSetValue((DWORD)v2[1].OwningThread, v5);
        a1[1][48] |= 0x100u;
        return -1;
      }
    }
    else if ( v4 == 0x80000004 )
    ł
      v6 = (DWORD *)&lpCriticalSection[1].OwningThread;
      v7 = TlsGetValue((DWORD)lpCriticalSection[1].OwningThread);
      if ( v7 )
      ł
        if ( !sub 401B36(v2, v7, 0) )
        ł
          TlsSetValue(*v6, 0);
          return -1;
        }
      }
   }
  }
 return v1;
}
```

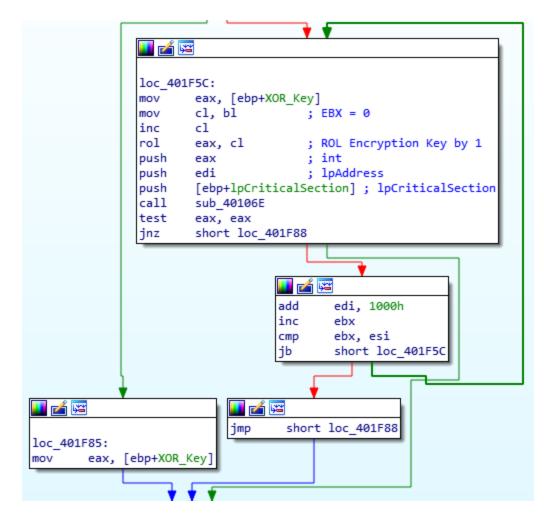
The important part of this function happens soon after it is executed. In the image below, you'll be able to make out a loop, as well as two XOR instructions and a ROR instruction. This is the BSS section decryption – the XOR key that is created later on is actually **arg\_8**, so take note of that. So, now we know that there will be an exception that is caused at some point, which will in turn execute the BSS string decryption function. Now we have this information, we can move onto reversing how the XOR key is created from the compilation/campaign date.



As the XOR key is based off of the date, thats where we need to look. First, the address of the BSS section (**ox6000**) is moved into EAX, and the size of it (**ox1000**) is moved into EDX. Then, the memory address of the compile date is moved into ESI, and a memory address (pointing to an empty section of memory) is moved into EDI. The compile date is then moved into that empty region of memory, using **MOVSD**, which moves a DWORD from the memory address in ESI to the memory address in EDI. Next, the first DWORD of the compile date is moved into ECX, and this is XOR'ed against the second DWORD of the compile date. Then, the result of this is added to the address of the BSS section (**ox6000** here) and the value **oxE**. This is then pushed as the first argument for the next function that will be called.

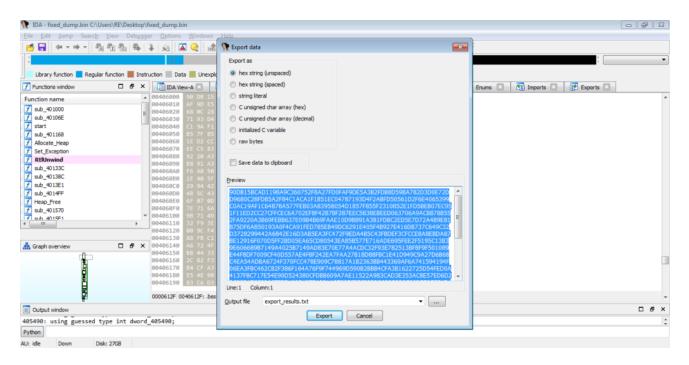


Taking a look at the function, **arg\_o** (the XOR key) is used in a Rotate Left instruction. Here, we see BL being moved into CL, which is incremented by 1 and used to rotate the XOR key left by 1. This results in the final XOR key that is used to decrypt the BSS strings.



Once we have performed these calculations, we get this as the key: **0x249d730c**. Whilst it is possible to get the strings from a debugger and copy it over to an IDA instance, I prefer to replicate custom routines using Python. I'm currently polishing up the decryption script I have been using, and when it is complete, you can get it **here** (my GitHub). Simply put, the algorithm decrypts the data in DWORDs, using a mixture of XOR's and rotate right (ROR) instructions. First, we're going to want to copy out the data in the BSS section. View it in the hex editor mode, select the entire section (where there is data), and then go **Edit-** >**Export Data**, and you can copy the unspaced data. Next, we need to parse this blob of data so that we can decrypt it.

00406000	90 DB 15 BC AD 11 98	A9 C3 60	75 2F 8A 27 FD 0F	.Û.‰.~©Ã`u/Š'ý.
00406010	AF 9D E5 A3 B2 FD B8		82 D3 D0 E7 2D D9	⁻.売ý,Õ~§,ÓĐç-Ù
00406020	68 0C 28 FD B5 A2 F8		1F 1B 51 EC 04 78	h.(ýµ¢øL.ÊQì.x
00406030	71 93 D4 F2 AB FD 50		6E 40 65 39 9C 0A	q"Ôò«ýPV./n@e9œ.
00406040	C1 9A F1 C6 48 78 A5		3A 83 95 80 54 D1	ÁšñÆHx¥wþ°:f•€TÑ
00406050	B5 7F 85 5F 23 10 85		BE BØ 7E C9 51 F1	µ#Õ¾°~ÉQñ
00406060	1E D2 CC 27 CF FC EC		F8 F4 2B 7B F2 87	.ÒÌ'Ïüìjp.øô+{ò‡
00406070	EE C5 83 8E BE ED 06		AC BB 78 B5 52 FA	îÅf޼í.7.©¬»xµRú
00406080	92 20 A3 B6 9F EB B6		4B 69 FA AE 10 D9	'∙£¶Ÿë¶7à>Kiú®.Ù
00406090	B8 91 A3 B1 FD BC 2E		2A 4B 9E 81 87 5D	,'£±ý¾.Õç×*Kž.‡]
004060A0	F6 A8 50 19 3A 0F 4C		78 5E B4 9D C6 29	ö"P.:.L@.íx^´.Æ)
004060B0	1E 40 5F 4B 92 7E 41		C6 49 C3 2D 37 28	.@_K'~Am•7ÆIÃ-7(
004060C0	29 94 42 A6 84 2E 16		A3 FC 47 2F 9E DA	)"B¦"Ó«^£üG/žÚ
004060D0	48 5C 43 FB DE F3 CF		EB DA 87 BE 12 91	K\CûÞóÏÌè¨ëÚ‡¾.'
004060E0	6F 07 0D 5F F2 BD 05		80 54 3E A8 5B 57	oò%.êeÍ€T>"[W
004060F0			C1 3B 39 E6 06 68	~qjÞi_î/Q•Á;9æ.h
00406100	9B 71 49 A4 02 5B 71		E7 0E 77 A4 AC DC	>qI¤.[qIfç.w¤⊣Ü
00406110	32 F9 3E 78 25 13 BF		10 89 E4 4F 8D F7	2ù>x%.¿.ŸP.‱äO.÷
00406120	00 9C F4 0D 55 7A E4		EA 7F AA 27 B1 8D	.œô.Uzä¢òBê.ª'±.
00406130	88 FB C1 E4 1D 94 9C		B6 8C 4E A5 4A DB	^ûÁä."œš'Ö¶ŒN¥JÛ
00406140	A6 72 4F 37 0F CC 47		78 81 7A 1B 23 63	r07.ÌGŽ.œx.z.#c
00406150	BB 44 33 69 AF 6A 74		4F 06 EA 3F BC 46	»D3i <sup>-</sup> jt.".O.ê?%F
00406160	2C 82 F3 86 F1 64 A7		49 69 D5 90 B2 BB	,,ó†ñd§oŸtIiÕ.²»
00406170	B4 CF A3 B1 62 27 25		ØA 41 37 FB C7 17	´Ï£±b'%ÕOí.A7ûÇ.
00406180	E5 4E 90 D5 24 38 0C		9A 7A E1 15 22 A9	åN.Õ\$8.ý»`šzá."©
00406190	83 CA D3 E3 53 AC 8E	57 ED 6D	23 C9 82 4E CC 82	fÊÓãS¬ŽWím#É,NÌ,



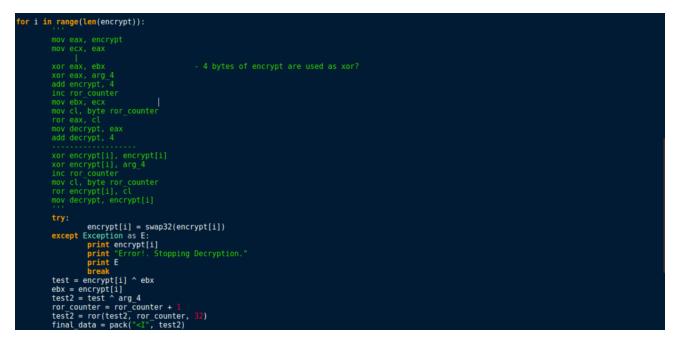
To parse it, I will be using python. All we need to do is split the hex bytes by 4, and store that in a list. This means each value will be a DWORD. If you follow the commands below, you should be able to get the output seen in the image.

r00t@ubuntu:~/Documents/Reverse Engineering/Tools/Malware/Hancitor+Ursnif/Ursnif/Ursnif_1\$ python Python 2.7.14+ (default, Dec 5 2017, 15:17:02)
GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> data = """90DB15BCAD1198A9C360752F8A27FD0FAF9DE5A3B2FDB8D598A782D3D0E72DD9680C28FDB5A2F84C1ACA1F1B51EC04787193D4F2ABFD50561D2F6E4065399C0AC1
9AF1C64878A577FEB03A83958054D1B57F855F2310852E1FD5BEB07EC951F11ED2CC27CFFCEC6A702EF8F42B7BF287EEC5838EBEED063706A9ACBB78B552FA9220A3B69FEBB637E0
9B4B69FAAE10D9B891A3B1FDBC2ED5E7D72A4B9E81875DF6A850193A0F4CA91FED785EB49DC6291E405F4B927E416DB737C649C32D3728299442A6842E16D3AB5EA3FC472F9EDA4B
SC43FBDEF3CFCCE8A8EBDA87BE12916F070D5FF2BD05EA65CD80543EA85B577E716ADE695FEE2F5195C13B39E606689B7149A4025B7149AD83E70E77A4ACDC32F93E782513BF8F9F
501089E44F8DF7009CF40D557AE4F8F242EA7FAA27B18D88FBC1E41D949C9A27D6B68C4EA54ADBA6724F370FCC478E909C78817A1B2363BB443369AF6A741594194F06EA3FBC462C
82F386F164A76F9F744969D590B28B84CFA3B1622725D54FED0A4137FBC717E54E90D524380CFDB8609A7AE11522A983CAD3E353AC8E57ED6D23C9824ECC820CF7B74095497FA981
A8E98FCBFC23BD2922B2D3F98D772E5F474AB30434B2E5ECA1C30B5927FA9266CD4C1F2C69225DEEBE11DD7B9143A56FFE629353B587923FB6EAFCF325974D667D995A3D28D31D95
956A94C0A9BEADDF601FB1B62EA7E0F681F81E6F07F1F6613CF6699E390C0E9E007F47D2AA3E7F5EFF236F52EFBE395E4223FB538DBF475F5E2163345E493115A172DDC047F7A2D7
AEDDBD89B806EA6101531CC8CF43E9E46EAFCE5CEEB6A878F40CE5A4495F90099956A941D184A5CB306F13C2AE1E2BC7AFC9D6D968A776BC335375E190009F7043372F54719144CA
849E16DAF2EEDE0F9079B0"""
>>> len(data)
1304
>>> n = 8
>>> [data[i:i+n] for i in range(0, len(data), n)]
['90DB15BC', 'AD1198A9', 'C360752F', '8A27FD0F', 'AF9DE5A3', 'B2FDB8D5', '98A782D3', 'D0E72DD9', '680C28FD', 'B5A2F84C', '1ACA1F1B', '51EC0478',
'7193D4F2', 'ABFD5056', '1D2F6E40', '65399C0A', 'C19AF1C6', '4878A577', 'FEB03A83', '958054D1', 'B57F855F', '2310852E', '1FD5BEB0', '7EC951F1',
'1ED2CC27', 'CFFCEC6A', '702EF8F4', '2B7BF287', 'EEC5838E', 'BEED0637', '06A9ACBB', '78B552FA', '9220A3B6', '9FEBB637', 'E09B4B69', 'FAAE10D9',
'B891A3B1', 'FDBC2ED5', 'E7D72A4B', '9E81875D', 'F6A85019', '3A0F4CA9', 'IFED785E', 'B49DC629', 'IE405F4B', '927E416D', 'B737C649', 'C32D3728',
'299442A6', '842E16D3', 'ABSEA3FC', '472F9EDA', '4BSC43FB', 'DEF3CFCC', 'EBA8EBDA', '87BE1291', '6F070DSF', '22D05EA', '6SCD8054', '3EA85B57',
'7716ADE', '695FEEZF', '5195C138', '39E60668', '98714944', '02587149', 'AD83E70E', '77A4ACDC', '32F93E78', '2513BF8F', '9F501689', 'E44F8DF7',
'009CF400', '557AE4F8', 'F242EA7F', 'AA27B180', '88FBC1E4', '10949C9A', '27D6668C', '4EA54ADB', 'A6724F37', '0FCC478E', '909C7881', '7A1B2363',
'BB443369', 'AF6A7415', '94194F06', 'EA3FBC46', '2C82F386', 'F164A76F', '9F744969', 'D590B2BB', 'B4CFA3B1', '622725D5', '4FED0A41', '37FBC717', 'E54E90D5', '24380CFD', 'BB609A7A', 'E11522A9', '83CAD3E3', '53AC8E57', 'ED6D23C9', '824ECC82', '0CF7B740', '95497FA9', '81A8E98F', 'CBFC23BD',
C34E5003, 24300CTU, BD003ATA, ELISZAY, B3CAUSES, SACAEST, EUGUZSEY, BZ4EC62, OLTBIAO, 3949TRA, BIABE30F, EUGUZSBU, 2922BD3: 1980D772E, SF47AB3: 043402E5, 1ECALIGAB, S927FA92, 166CHC41F, 2C692250, 1EEBEIDD), 78914AS, 16FFE6293, 53B58792',
29228203, F900/72C, 544/4863, 043492C5, ECALC306, 392/7892, 00CU4CLF, 2C092230, EEBETHUD, 78914383, 0FFC0293, 53650192, 3780647C', F325974D', 6670995A', 30280310', 959566494', CGA9BEAD', DF601FB1', B62EA7E0', F681FB1C', 6677F66', 61376F69', 9E390C0E',
3F80CAFC, F3239740, 0070935A, 30200310, 33930A94, COA90CA0, 07001701, 802CA760, 7081781C, 07071760, 013C7009, 9539000C, 9600777', 102AA3C7F', 52FF8267', 52FF8267', 52FF8236', 5380BF47', 55F2163', 335454931', 15A1720D', 10477A2', 1074EDDBD', 898806EA',
50001747, UZAASETT, SETTESOT, SETTESS, SE4223FD, SSBUP47, STSE2105, 345E4551, ISAT7200, COMPTAZ, UTAEUDDU, 6506000EA,
BC335375', 'E190095F', '704372E', '54719144', 'CA849E16', 'DAF2EEDE', '0F907980']
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Next, we're going to use CyberChef to do a bit more fixing, although we will be replacing different characters, so that there is an **ox** before every DWORD, and so Python won't treat the list as a list of strings.

Download CyberChef 🛓	Last build: 16 d	ays ago - New in v8: Automated	encoding detection and simplified operation building	Options 🏩 About / Support 🕐	
Operations	Recipe	2 🖿 🕯	Input	length: 2116 → î	
rep	Find / Replace         II         200475200, 0X F1044707, 0X 597144903, 0X 05300200, 0X 040074301, 0X 4FE00A41, 0X 37FBC717, 0X E54E9005, 0X 24380CFD, 0X 8B609A7 E11522A9, 0X 83CAD3E3, 0X 53AC8E57, 0X ED0023C9, 0X 824ECC82,			24380CFD, 0x BB609A7A, 0x	
Find / <u>Rep</u> lace	Find , OX	REGEX -	0x 95497FA9, 0x 81A8E98F, 0x CBFC23BD, 0x 2922B2D3, 0x F98D772E, 0		
BSON deserialise	Replace		5F474AB3, 0x 0434B2E5, 0x ECA1C30B, 0x 5927FA92, 0x 66CD4C1F, 0x 2C69225D 0x EEBE11DD, 0x 7B9143A5, 0x 6FFE6293, 0x 53B58792, 0x 3FB6EAFC, 0x 25E524D, 0x 652D024D, 0x 052D32D, 0x 000054D, 0x 000054D, 0x 000054D		
BSON serialise	, 0x	F325974D, 0x 667D995A, 0x 3028031D, 0x 95956A94, 0x C0A9BEAD, 0x DF601FB1 0x B62EA7E0, 0x F681F81E, 0x 6F07F1F6, 0x 613CF669, 0x 9E390C0E, 0x 0x007E74, 0x D5407E76, 0x 657E7576, 0x 657E7570, 0x 65907E1			
Bit shift right	Global match	Case insensitive	9E007F47, 0x D2AA3E7F, 0x 5EFF236F, 0x 52EFBE39, 0x 5E4223FB, 0x 5380BF4 0x 5F5E2163, 0x 345E4931, 0x 15A172DD, 0x C047F7A2, 0x D7AEDDBD, 0x 89B806EA, 0x 6101531C, 0x C8CF43E9, 0x E46EAFCE, 0x 5CEEB6A8, 0x 78F40CE		
Decode NetBIOS Name	✓ Multiline matching         □ Dot matches all           ✓ Multiline matching         □ Dot matches all		CB306F13, 0x C2AE1E2B, 0x		
Encode NetBIOS Name					
Entropy			Output start: 0 end: 1954 length: 1954	time: 1MS length: 1954 lines: 1	
From BCD			[90DB15BC, 0xAD1198A9, 0xC360752F, 0x8A27FC 0x98A782D3, 0xD0E72DD9, 0x680C28FD, 0xB5A2F		
From Hex Content			0x7193D4F2, 0xABFD5056, 0x1D2F6E40, 0x65399 0xFEB03A83, 0x958054D1, 0xB57F855F, 0x23108		
From MessagePack			0x1ED2CC27, 0xCFFCEC6A, 0x702EF8F4, 0x2B7B7 0x06A9ACBB, 0x78B552FA, 0x9220A3B6, 0x9FEB		
From Punycode			0xB891A3B1, 0xFDBC2ED5, 0xE7D72A4B, 0x9E818 0x1FED785E, 0xB49DC629, 0x1E405F4B, 0x927E4		
From UNIX Timestamp			0x299442A6, 0x842E16D3, 0xAB5EA3FC, 0x472F5 0xE8A8EBDA, 0x87BE1291, 0x6F070D5F, 0xF2BD0		
Parse ASN.1 hex string	STEP	BAKE!	0x7E716ADE, 0x695FEE2F, 0x5195C13B, 0x39E60 0xAD83E70E, 0x77A4ACDC, 0x32F93E78, 0x2513E	BF8F, 0x9F501089, 0xE44F8DF7,	
Remove Diacritics	SIEP 👗	Auto Daka	0x009CF40D, 0x557AE4F8, 0xF242EA7F, 0xAA27E 0x27D6R68C 0x4F4544DR 0x46724E37 0x6672		

So now you should have a suitable list of hex DWORDs which we can then decrypt. In order to decrypt it, you simply have to copy it into the script and make sure the key is correct, and then it will decrypt and spit out the raw data, as well as a list of integers, and you will see why just now. In the image below, you can see the decryption part of my script, which isn't too complicated, so if it hasn't gone up on my GitHub yet, it shouldn't be too difficult to replicate.



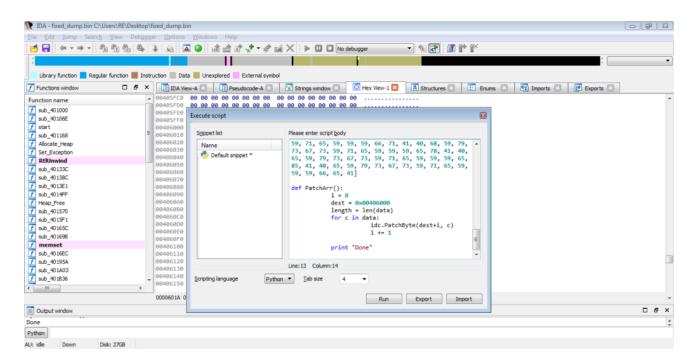
Once the script has decrypted the data, the output looks like this:



So now we can copy the decrypted strings over to IDA. There is an extremely basic IDA Python script in the image below, which overwrites bytes at **dest** with bytes of data in **data**. The list **data** will contain the integer values that our script printed to the terminal, so you can simply copy it from there and paste it into the document.



From there, we need to import it as a script/module in IDA, so you can do that either by importing the file, or by copying the text and pasting it into the box as shown below. All you need to do then is click **Run**, and it should be imported – although we physically have to call the script, **Run** won't execute it – at least not in this case.



Upon typing **PatchArr()** into the console and hitting enter, the script should overwrite all bytes in the BSS section with our decrypted strings, so you should see something similar to the image below.

	View Debugger Options Windows Help		
🚔 🔚 🔅 🖛 🖛 🗸 🖉	n 🐁 🗣 🗼 😥 🖬 🖉 🕘 📾 📾 💕 📌 🛛	🖬 🗙 🕨 🔲 🗖 No debugger 💿 🤨 💽 🗊 🕆 🎬	
			:
Library function 📃 Regul	function 📕 Instruction 📃 Data 📕 Unexplored 📒 External sy	lool	
Functions window	🗆 🗗 🗙 📑 IDA View-A 🙁 📑 Pseudocode-A	Strings window 🗵 🖸 Hex View-1 🗵 🖪 Structures 🗵 👯 Enums 🗵 🛐 Imports 🗵 😿	Exports
nction name	A 00405FD0 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00	
sub 401000	00405FE0 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00	
	00405FF0 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00	
sub_40106E	00406000 4E 54 44 4C 4C 2E 44 4C	4C 00 4E 54 44 53 41 50 NTDLL.DLL.NTDSAP	
start	E 00406010 49 2E 44 4C 4C 00 4B 45	52 4E 45 4C 33 32 2E 44 I.DLL.KERNEL32.D	
sub_401168	00406020 4C 4C 00 5A 77 53 65 74		
Allocate_Heap	00406030 68 72 65 61 64 00 5A 77	47 65 74 43 6F 6E 74 65 hread.ZwGetConte	
Set_Exception	00406040 78 74 54 68 72 65 61 64	00 5A 77 57 6F 77 36 34 xtThread.ZwWow64	
RtlUnwind	00406050 52 65 61 64 56 69 72 74		
sub_40133C	00406060 79 36 34 00 5A 77 57 6F	77 36 34 51 75 65 72 79 y64.ZwWow64Query	
sub_40138C	00406070 49 6E 66 6F 72 6D 61 74 00406080 73 73 36 34 00 2E 62 69		
sub_4013E1	00406090 62 72 61 72 79 41 00 25		
sub_4014FF	004060090 62 72 61 72 79 41 00 23 004060A0 2D 25 30 32 75 20 25 30		
Heap_Free	00406080 25 30 32 75 0D 0A 00 36		
	00406000 74 55 73 65 72 54 68 72		
sub_401570	004060D0 2A 00 00 4C 64 72 47		
sub_4015F1	004060E0 75 72 65 41 64 64 72 65		
sub_40165C	004060F0 65 52 65 6D 6F 74 65 54	68 72 65 61 64 00 5A 77 eRemoteThread.Zw	
sub_401698	00406100 57 72 69 74 65 56 69 72		
memset	00406110 72 79 00 4C 64 72 4C 6F		
sub_4016EC		69 72 74 75 61 6C 4D 65 ProtectVirtualMe	
sub_40195A	_ 00406130 6D 6F 72 79 00 6B 65 72		
		74 65 72 44 6C 6C 4E 6F LdrRegisterDllNo	
m			
	0000602A 0040602A: .bss:0040602A (Synchro	nized with IDA View-A)	
Output window			- <i></i>
:hon>			
:hon>PatchArr()			
e			
thon			

We can then reanalyse the payload, by going **Options->General->Analysis-**

>**Reanalyze Program**, and it should recognize most of the strings, although there will be the occasional error.

1			
Library function Requi	lar function 📕 Instruction	Data Unexplored External symbol	
Functions window		IDA View-A 🛛 🕐 Hex View-1 🔍 📘 Pseudocode-A 🗶 🔝 Strings window 🗶 🖪 Structures 🗶 🗮 Enums 🗶 🛐 Imports 🗶 😿 Exports 🗶	
Function name	*	bss:00406000 aWtdllDll_0 db 'NTDLL.DLL',0 ; DATA XREF: sub_40195A+2E10	
f sub 401000		.bss:00406000 ; sub_401A03+AC1o	
f sub_40106E		bs::0040600A aNtdsapiDll db 'NTDSAPI.DLL',0	
f start	-	.bss:00406016 ; const CHAR ModuleName .bss:00406016 ModuleName db 'KERNEL32.DLL',0 ; DATA XREF: sub 40138C+18to	
f sub_401168	-	.055:00406016 MOULENAME OD KENNELSZ.ULL,0 ; DATA AKET: SUD-00150.41010 .bss:00406016 ; sub-40251641C10 ; sub-40251641C10	
f Allocate_Heap		.bs:00406023; CHAR aZwsetcontextth[2]	
		<ul> <li>bss:00406023 aZwsetcontextth db 'ZwSetContextThread',0</li> </ul>	
f Set_Exception		.bss:00406023 ; DATA XREF: sub 401570+1Eto	
f RtlUnwind		.bss:00406036 ; CHAR String2[2]	
f sub_40133C		bss:00406036 String2 db 'ZwGetContextThread',0	
f sub_40138C		.bss:00406036 ; DATA XREF: sub_401570+Eto	
f sub_4013E1		.bss:00406049 ; CHAR aZwwow64readvir[3]	
f sub_4014FF		bss:00406049 aZwwow64readvir db 'ZwWow64ReadVirtualMemory64',0	
f Heap_Free		.bss:00406049 ; DATA XREF: sub_4029C2+17to	
f sub_401570		.bss:00406064 ; CHAR aZwwow64queryin[11]	
f sub_4015F1		<ul> <li>.bss:00406064 aZwwow64queryin db 'ZwWow64QueryInformationProcess64',0</li> </ul>	
f sub_40165C		.bss:00406064 ; DATA XREF: sub_402EA1+11to	
f sub_401698		.bss:00406085 aBin db '.bin',0 .bss:0040608A aLoadlibrarya db 'LoadLibraryA',0	
7 memset		.DSS:0040008A aLOadllDrarya dD LOadllDraryA, 0	
f sub_4016EC		.0510040007 4020200200200200200200 db '64'.0	
f sub_40195A		.bs:0040000 a04 atlexituserthr db 'ttlExitUserThread'.0	
	*	. bs:0040600C text "UF-16LE", '* * 0	
( III )			
		00006049 00406049: .bss:aZwwow64readvir (Synchronized with Hex View-1)	
Output window			08
ommand "MakeStrlit"	failed		
Command "JumpOpXref" f			
command "JumpOpXref" f			

With the newly decrypted BSS section, we should be able to analyse the rest of the payload without issues.

Once the strings have been decrypted, ISFB then calls a function that utilizes one of the decrypted strings – **IsWow64Process**. As you probably have guessed, this checks to see if the architecture of the system is 64 bit or not. The result (stored in EAX – **1** if x64, **0** if not) is then tested. If the system is not 64 bit, the variable **var\_4** in the graph below is used in

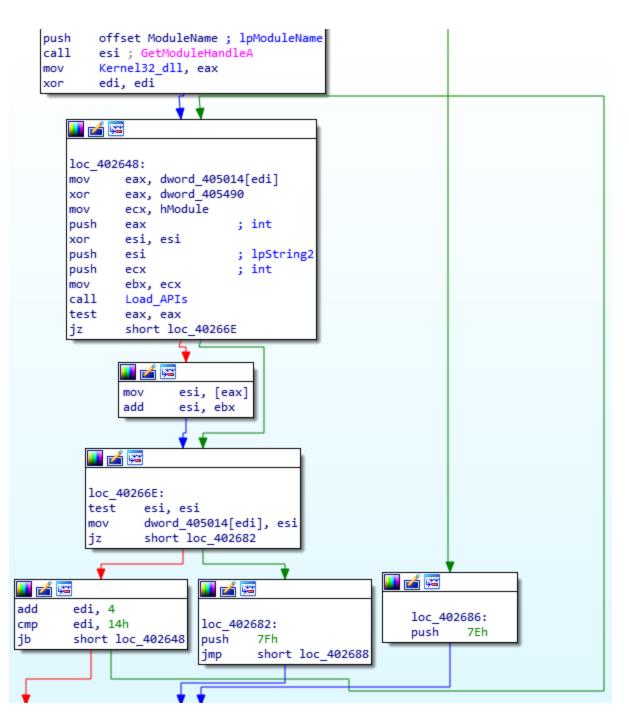
an AND operation with EAX, which would be equal to **o**, meaning the value in **var\_4** would be **o**. If the system is 64 bit, this is skipped. Then, regardless of the system architecture, the value in **var\_4** is moved into EAX, which is the return value.

If we back out of the function, we can see EAX is tested again, and if the result is not zero, the value **1** will be moved into the DWORD **dword\_405478**. As the system I am running on is 64 bit, **dword\_405478** will contain the value **1**. If we search for cross references to this DWORD, we can find test instructions that use it. Therefore we can determine this is an indicator of the architecture.



	mov dwor	d_405478, 1
😅 xrefs to dword_405478		
Directio Tyj Address	Text	
📴 Up r sub_401000+	41 test byte ptr dw	vord_405478, 1
📴 Up r sub_401F96+3	2B test byte ptr dw	vord_405478, 1
🖼 Up r 🛛 sub_401F96+.	213 test byte ptr dw	ord_405478, 1
w sub_402473+	35 mov dword_405	5478, 1
🛛 🖼 D r sub_402473:ld	oc_4025BE test byte ptr dw	ord_405478, 1
	OK Car	ncel Search Help
Line 3 of 5		

The next function that is called is quite complex. In a nutshell, this function is responsible for getting the address to a select few API calls. To do this, it uses a list of predefined values, a "key", and a hashing routine. First, let's take a look at the key and find out what it is. Looking at the image below, just before the value in **hModule** is moved into ECX, EAX is XOR'd with a DWORD inside the binary, however upon viewing this DWORD, it is empty – this means it is resolved dynamically.



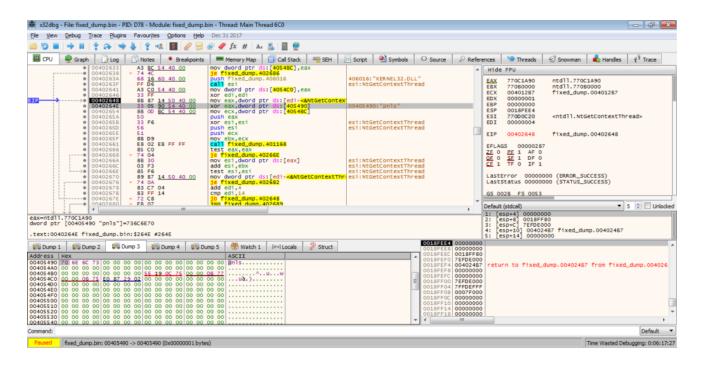
To find out what this DWORD will contain, we need to find cross references to it – specifically, we are looking for a reference to it being used as the destination in a MOV instruction. Luckily enough for us, it only appears once as the destination, inside the BSS Decrypt function. From the image below, we can see that a DWORD from a string is moved into ECX, which is then used in a subtraction and addition instruction. In this case, it may be much easier to understand what is going on by looking at it in a decompiler.

```
🔺 🖼
           ecx, dword ptr ds:aVersionUSoftUU ; "version=%u&soft=%u&user=%08x%08x%08x%08"...
  mov
           ecx, dword ptr ds:aVersionUSoftUU+0Ch ; "oft=%u&user=%08x%08x%08x%08x%erver=%u&".
  sub
           ecx, dword ptr ds:aVersionUSoftUU+4 ; "ion=%u&soft=%u&user=%08x%08x%08x%08x&se"...
  add
  mov
           dword 405490, ecx
 端 xrefs to dword_405490
                                                                                                      Directio Tyj Address
                                      Text
  🖼 Up r sub_4013E1+5
                                      mov
                                           ecx, dword_405490
                                            dword_405490, ecx
  44
             Decrypt_BSS_Strings+6
  🖼 D... r
             sub 401F96+118
                                            edi, dword 405490
                                      mov
  🖼 D... r
             sub_402473+5C
                                            eax, dword_405490
                                      mov
  🖼 D... r
             API_Hashing_Lookup+30
                                           eax, dword_405490
                                      xor
                                      OK
                                                  Cancel
                                                               Search
                                                                             Help
  Line 2 of 5
int __thiscall Decrypt_BSS_Strings(void *this)
ł
 HMODULE v1; // ebx
  int result; // eax
 int dword_1; // [esp+4h] [ebp-14h]
 int dword_2; // [esp+8h] [ebp-10h]
 int BSS_Size; // [esp+10h] [ebp-8h]
int BSS_Addr; // [esp+14h] [ebp-4h]
 v1 = dword_40547C;
  result = Get_Address_And_Size_Of_BSS((int)this, (int)dword_40547C, &BSS_Addr, &BSS_Size);
  if ( !result )
 {
   result = Set_Exception();
   if ( !result )
   {
     strcpy((char *)&dword_1, "Jan 28 2019");
     result = sub_401F22((int)v1 + BSS_Addr, BSS_Size, (dword_2 ^ dword_1) + BSS_Addr + 14);
     if ( !result )
       dword_405490 = *(_DWORD *)&aVersionUSoftUU[4] + *(_DWORD *)aVersionUSoftUU - *(_DWORD *)&aVersionUSoftUU[12];
   }
  }
 return result;
                                                                I
}
```

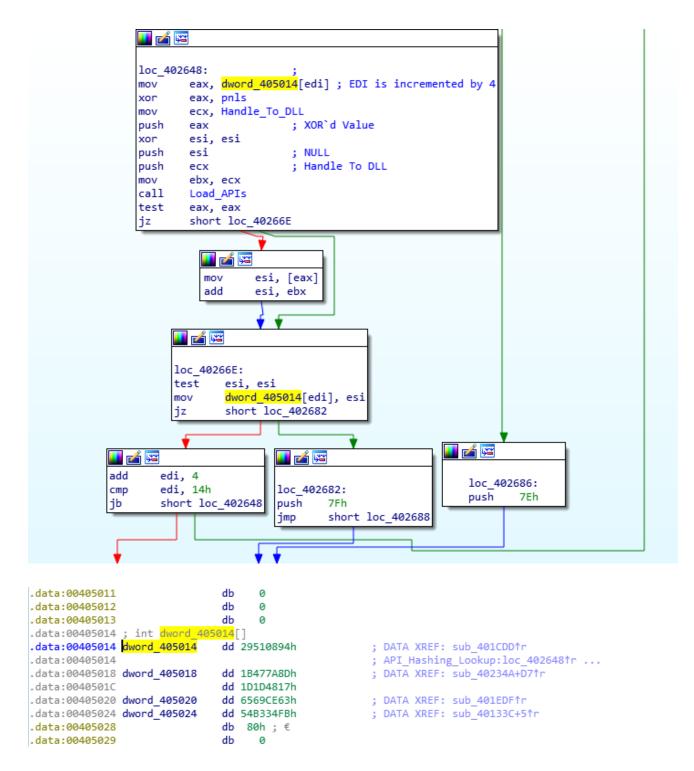
So, from the looks of it, we can get the key used by performing a simple operation, as seen below:

```
key = (dword string[4] + dword string[0]) - dword string[12]
key = ("ion=" + "vers") - "oft="
We need to convert these values to hex to perform addition and subtraction:
key = (0x696f6e3d + 0x76657273) - 0x6f66743d
key = 0x706E6C73
Convert the key from hex and we get this: "pnls"
```

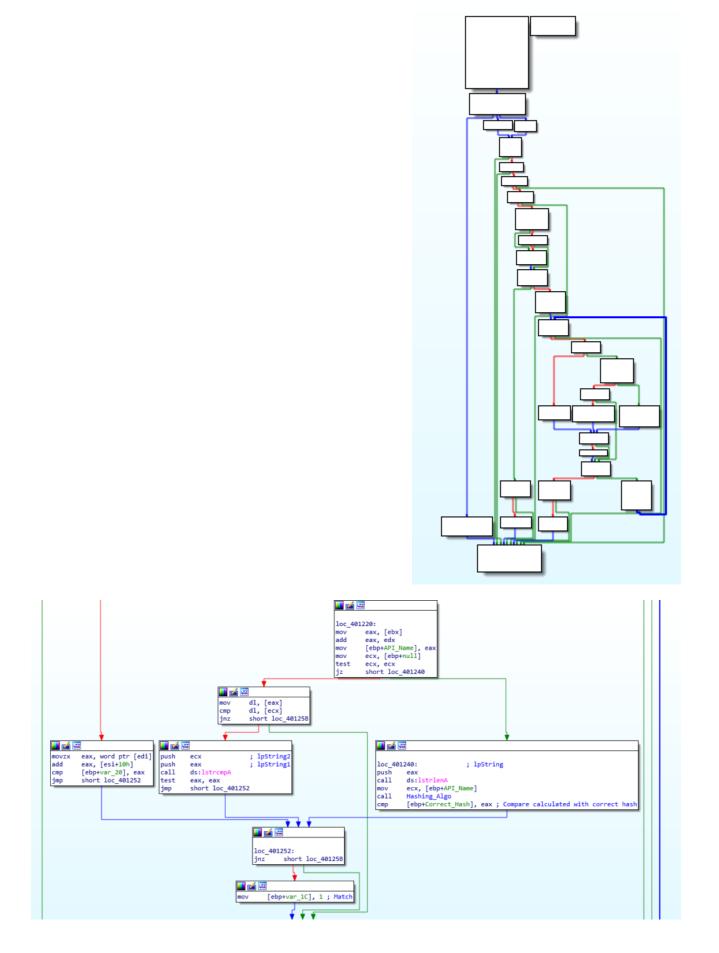
We can double check this by looking at the binary in a debugger, as shown in the image below.



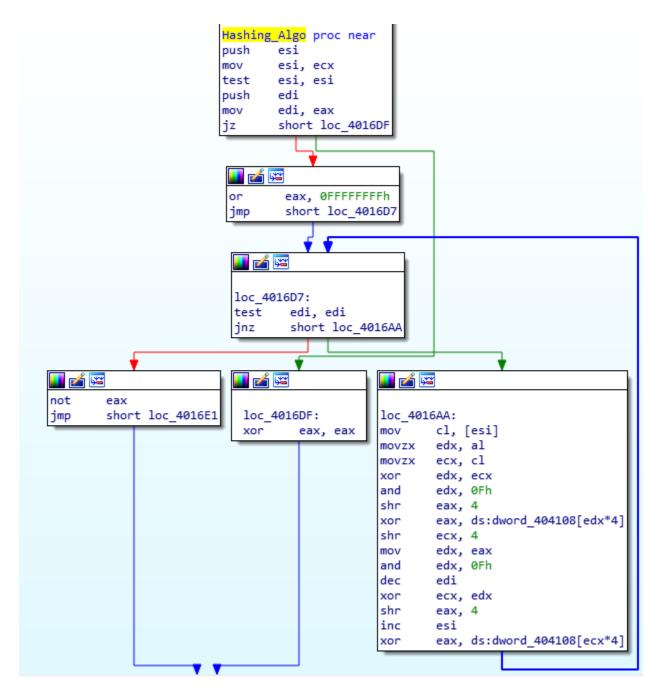
Now we have the key, let's take a look at the values used for determining which API to import. This is quite simple to find, as it is already inside the executable. We can also see that EDI is being used as a counter, as each loop it is incremented by 4, until the value reaches 20, where it returns. This means there are 5 API's in total that are looked up using this method. The values in the embedded list are all XOR'd by the key, which results in the hash lookup value used for comparison. So, from this, we can start examining the hashing algorithm to see how it functions and to locate what API's are lookup up and stored.



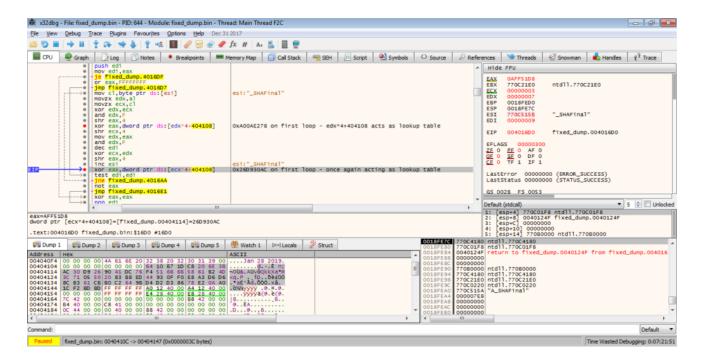
As the function is quite large, I will focus on the section that calls the hashing function. The three arguments passed to this function are; the address of the DLL, the value **o**, and the correct hash for comparison. The function uses the base address to perform some calculations in order to get to the export table inside the DLL. From there, it loops through each of the exports and hashes the name of the export, which is then compared to the predefined hash. If the hashes match, the function retrieves the address to the exported API, and overwrites the predefined hash with the address for later use. If they do not match, the function simply continues onto the next export, until a match is finally discovered.



Taking a look at the actual hashing function, it is quite difficult to come to the conclusion (especially for a beginner) that the function is responsible for hashing, until it is run in a debugger. There is definitely something happening in this function from a static analysis perspective, as there are multiple logic instructions, but we can't know for sure until we analyse it further.



Taking a look in a debugger, it is clear that [edx\*4+404108] and [ecx\*4+404108] are values from a lookup table, as the XOR values used constantly change, however the values are repeated, and therefore we can determine that they are not randomized. Base64 encoding/decoding use lookup tables as well, so if you have looked at that before (or at least the psuedocode), you might be able to recognize the lookup table aspect here. When we take a look at the memory region where the lookup table is located, it is easy to see where it begins and ends. So, now we know that there is a lookup table, how do we find out the hashing mechanism in use?



Typically, custom encryption and hashing implementations are quite difficult to determine, unless you know what you are looking for. As long as the algorithm is publicly available (such as AES or Serpent), and not a custom developed one by the author, there will almost always be specific values or instructions that stick out to those who have looked at crypto before – these are known as Constants. We will revisit these later (in the next post) when looking at the other encryption methods used in this sample, however to sum it up, it basically refers to values that must be used in the algorithm to achieve the correct results. In this case, let's take one value from the lookup table: OXAOOAE278. We can run a quick search for this value online, and as you can see from the image below, this is definitely linked to CRC-32, although it is more of a variant – we just need to find out which variant.



### [FIXED] Working Arduino CRC 32 code - Arduino Forum forum.arduino.cc/index.php?topic=91179.0 -

9 Feb 2012 - 7 posts - 3 authors 0x9b64c2b0, 0x86d3d2d4, **0xa00ae278**, 0xbdbdf21c }; unsigned long crc\_update(unsigned long crc, byte data) { byte tbl\_idx; tbl\_idx = crc ...

#### CRC32 with signed bytes in java - Stack Overflow

# https://stackoverflow.com/questions/28405493/crc32-with-signed-bytes-in-java - 2 answers

26 Apr 2017 - ... 0x6b6b51f4, 0x4db26158, 0x5005713c, 0xedb88320, 0xf00f9344, 0xd6d6a3e8, 0xcb61b38c, 0x9b64c2b0, 0x86d3d2d4, **0xa00ae278**, 0xbdbdf21c }; public ...

generate a Checksum using this algorithm	30 Oct 2018
Converting crc code from C to Java yields unexpected results	12 May 2018
Reading bytes out of a character array using a character pointer	8 Jun 2017
Arduino CRC calculation and Serial reading	24 May 2017
More results from stackoverflow.com	

# 2.1.2.2.1 CRC Lookup Table - Microsoft Docs

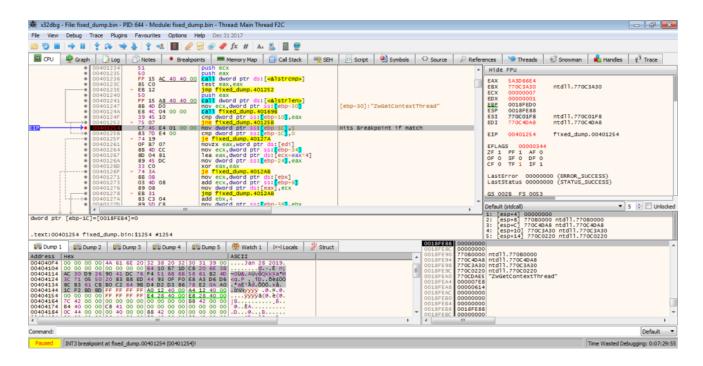
https://docs.microsoft.com/en-us/.../ms.../2500f63e-a1ec-4605-a80d-d41ab63e80e9 ▼ 14 Feb 2019 - ... 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, ...

#### crc32.c

# ftp://ftp.ncdc.noaa.gov/pub/data/software/nexrad2/crc32.c -

... 0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, **0xa00ae278**, 0xd70dd2ee, 0x4e048354, ...

In this specific sample, the program loops through the NTDLL.DLL export functions and hashes them all until it finds a matching hash. What makes hashing worse is that hashes are irreversable, meaning the only way you can find the matching hash is by brute forcing – to do this you could hash each API exported by NTDLL until you found the matching hashes, but this is too time consuming, and it is quite simple to find the matching API anyway – all you have to do is put a breakpoint on the instruction that is hit if a match is found, and execute the program. Once the breakpoint is hit, you'll be able to locate the correct API. From the image below, you can see the first API call that matches is ZwGetContextThread. We can now use this to get the variant of CRC-32 used.

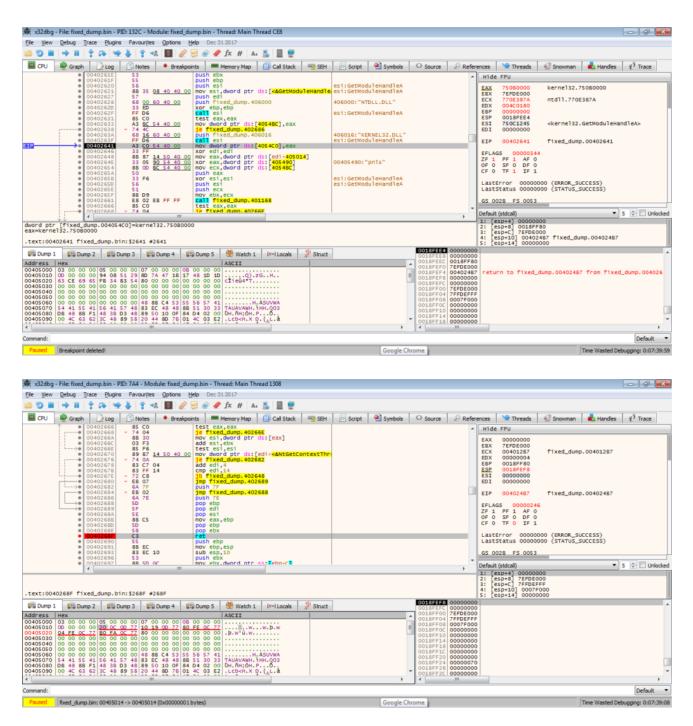


In order to do so, we can use this **<u>site</u>**, which is extremely useful. It hashes the input using several different variants of CRC-8, CRC-16, and CRC-32, allowing us to compare the value from the debugger to the output of the different variants. With the API name, ZwGetContextThread, and the required output, 0x5A3D66E4, we can find the variant used: CRC-32/JAMCRC.

→ C Secure   htt	tps://crccalc.com												
		Or	nline Cl	RC Cal	culator								
		PHP-sources of	on GitHub C#-so	ources on GitHub	Java-sources on	GitHub							
	ZwGetContextThread												
	Input type:      ASCII      Hex												
	Algorithm	Result	Check	Poly	Init	RefIn	Ref0ut	XorOut					
	CRC-32	0xA5C2991B	0xCBF43926	0x04C11DB7	0×FFFFFFFF	true	true	0xFFFFFFFF					
	CRC-32/BZIP2	0xA5DD7B93	0xFC891918	0x04C11DB7	0xFFFFFFFF	false	false	0xFFFFFFFF					
	CRC-32C	0xF4B9ADCE	0xE3069283	0x1EDC6F41	0xFFFFFFFF	true	true	0xFFFFFFFF					
	CRC - 32D	0xA1DFBC37	0x87315576	0xA833982B	0xFFFFFFFF	true	true	0xFFFFFFFF					
	CRC-32/MPEG-2	0x5A22846C	0x0376E6E7	0x04C11DB7	0xFFFFFFFF	false	false	0x00000000					
	CRC-32/POSIX	0xE8D15C8A	0x765E7680	0x04C11DB7	0x00000000	false	false	0xFFFFFFFF					
	CRC-32Q	0x037FE891	0x3010BF7F	0x814141AB	0×00000000	false	false	0×00000000					
	CRC-32/JAMCRC	0x5A3D66E4	0x340BC6D9	0x04C11DB7	0xFFFFFFFF	true	true	0×00000000					
	CRC-32/XFER	0x3524318D	0xBD0BE338	0x000000AF	0x00000000	false	false	0x00000000					

As we can find out the matching APIs, there is no need to write a brute force script, although for families **such as GootKit**, this is sometimes necessary. So, the 5 APIs imported are: ZwGetContextThread, ZwSetContextThread, ZwReadVirtualMemory,

ZwWriteVirtualMemory, and ZwAllocateVirtualMemory. With that, we can move out of this function, and onto the next.



The next function simply retrieves the file name to store in memory, potentially for use later on in the sample. The next function gets much more interesting, so lets move onto that.

```
int Get FileName()
{
 WCHAR *v0; // ebx
 DWORD v1; // esi
 WCHAR *v2; // eax
  int v4; // [esp+Ch] [ebp-8h]
  LPCWSTR lpszShortPath; // [esp+10h] [ebp-4h]
 v4 = GetModule FileName(base addr, (int)&lpszShortPath, 1);
  if ( v4 )
  {
   lpString2 = 0;
  }
  else
  {
    v0 = (WCHAR *)lpszShortPath;
    v1 = GetLongPathNameW(lpszShortPath, 0, 0);
    if ( v1 && (v2 = (WCHAR *)Allocate Heap(2 * v1 + 2), (lpString2 = v2) != 0) )
    {
      GetLongPathNameW(v0, v2, v1);
      Heap_Free(v0);
    }
    else
    {
      lpString2 = v0;
    }
  }
  return v4;
}
```

If you've looked at ISFB or read anything about it before, you might be aware of an embedded FJ, F1, or JJ structure that is located just after the section table. This structure contains pointers to appended (also known as joined) data, such as configuration information, an RSA public key, or even another executable. There are differences between FJ, F1, and JJ, although they are quite small changes. In this case, the sample is using an embedded JJ structure, identifiable from the magic value 0x4A4A ("JJ").

📓 fixed_afterdump.bin				
Offset(h) 00 01	02 03 04 0	5 06 07 08 09 0A 0E	OC OD OE OF Decoded text	
000001E0 00 00	00 00 00 0	0 <b>00 00 00 00 00</b> 00	00 00 00 00	
000001F0 00 00	00 00 00 0	0 00 00 2E 74 65 78	74 00 00 00tex	t
00000200 6F 26	00 00 00 1	0 00 00 00 30 00 00	00 10 00 00 060	
00000210 00 00	00 00 00 0	0 00 00 00 00 00 00	20 00 00 60	
00000220 2E 72	64 61 74 6	1 00 00 9E 06 00 00	00 40 00 00 .rdataž	
00000230 00 10	00 00 00 4	0 00 00 00 00 00 00	00 00 00 00@	
00000240 00 00	00 00 40 0	0 00 40 2E 64 61 74	61 00 00 00@@.dat	a
00000250 14 05	00 00 00 5	0 00 00 00 10 00 00	00 50 00 00P	.P
00000260 00 00	00 00 00 0	0 00 00 00 00 00 00	40 00 00 C0	@À
00000270 2E 62	73 73 00 0	0 00 00 8D 02 00 00	00 60 00 00 .bss	· ` · ·
00000280 00 10	00 00 00 6	50 <b>00 00 00 00 00</b> 00	00 00 00 00`	
00000290 00 00	00 00 40 0	0 00 C0 2E 72 73 72	63 00 00 00@À.rsı	c
000002A0 10 00	00 00 00 7	0 00 00 00 10 00 00	00 70 00 00p	.p
000002B0 00 00	00 00 00 0	0 00 00 00 00 00 00	40 00 00 40	00
000002C0 2E 72	65 6C 6F 6	3 <b>00</b> 00 <b>00</b> 70 <b>00</b> 00		
	00 00 00 8	0 00 00 00 00 00 00		
000002E0 00 00	00 00 40 0	0 00 40 00 00 00 00	00 00 00 00@@	
000002F0 00 00	00 00 00 0	00 00 00 00 00 00 00	00 00 00 00	
00000300 00 00	00 00 00 0	00 00 00 00 00 00 00		
00000310 4A 4A	00 01 8F C	6 B1 D4 OC 4A 15 9E	00 84 00 00 JJƱÔ.J.ž	
00000320 00 B6	00 00 00 0	00 00 00 00 00 00 00		
00000330 00 00	00 00 00 0	00 00 00 00 00 00 00	00 00 00 00	
00000340 00 00	00 00 00 0	00 00 00 00 00 00 00	00 00 00 00	
00000350 00 00	00 00 00 0	00 00 00 00 00 00 00	00 00 00 00	
00000360 00 00	00 00 00 0	00 00 00 00 00 00 00	00 00 00 00	
00000370 00 00		00 00 00 00 00 00 00		
00000380 00 00		00 00 00 00 00 00 00	00 00 00 00	
00000390 00 00		00 00 00 00 00 00 00		
000003A0 00 00		0 00 00 00 00 00 00		
000003B0 00 00		0 00 00 00 00 00 00		
000003C0 00 00		0 00 00 00 00 00 00		
000003D0 00 00	00 00 00 0	0 00 00 00 00 00 00	00 00 00 00	
Offset(h): 0				0

Overwrite

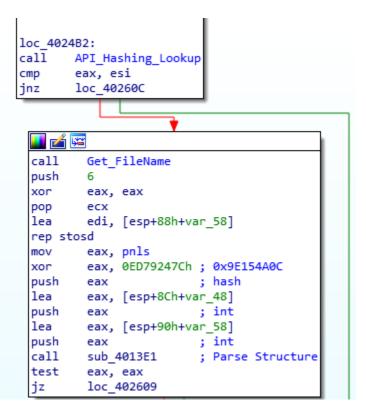
The format of this structure is shown below:

```
Joined Resource Structure {
	WORD Magic Value
	WORD Flags
	DWORD XOR Key
	DWORD CRC-32 Hash
	DWORD Address
	DWORD Size
}
```

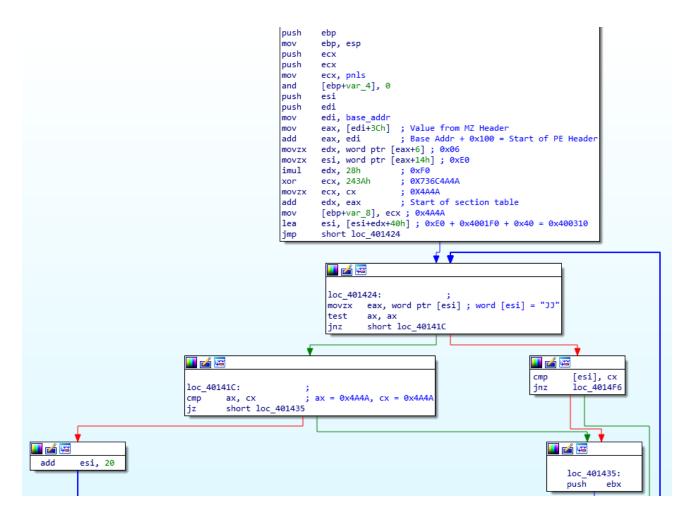
The parsed structure in this binary can be seen below:

```
Joined Resource Structure {
    WORD 0x4A4A
    WORD 0x0001
    DWORD 0xD4B1C68F
    DWORD 0x9E154A0C
    DWORD 0x00008400
    DWORD 0x0000B600
}
```

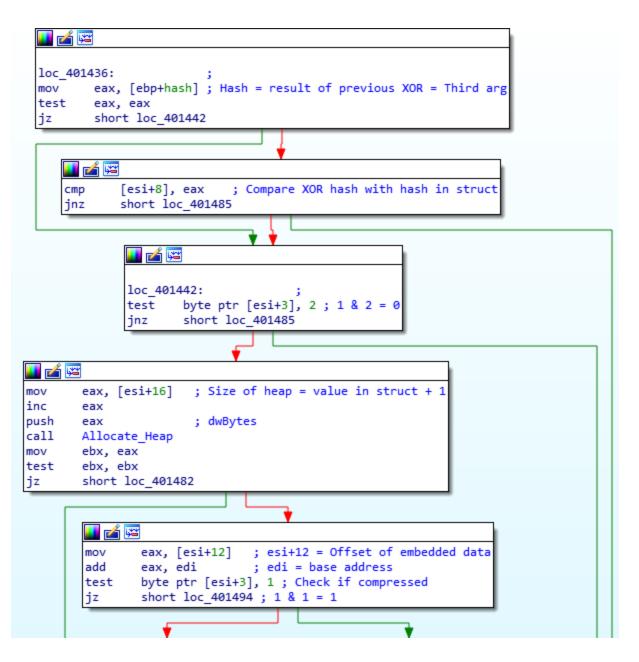
The function we are looking at is responsible for parsing this structure and locating the needed data. The main argument for this function is the third one, which is the result of an XOR between the "pnls" string and an embedded hex value. The first and second arguments are simply the start and end of an empty region of memory, so they are not that important right now.



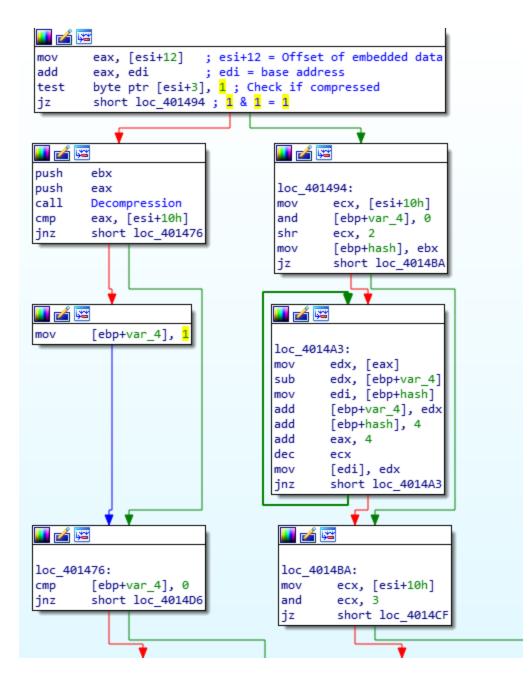
The first half of the parsing function traverses through the MZ and PE header until it has reached the resource structure located just after the section table. It then checks that the magic value is in fact "JJ" (0x4A4A), otherwise it will add 0x14 to the current address and try again.



Once the structure has been located successfully, the function compares the third argument (0x9E154AoC) with the embedded CRC-32 hash, which is also 0x9E154AoC. If these do not match, the function will return or loop around again. If they do match, the function performs a bitwise AND on the structure flag and 0x2, which must return 0, otherwise the function will loop or return. If 0 is returned, a heap is allocated based on the size value stored in the structure. From there, it will get the full memory address of the joined data by adding the address in the structure to the base address of the executable, and then a bitwise AND is performed on the structure flag and 0x1. If the result is 0 the joined data is not compressed and is encoded, and if the result is not 0 (which in this case it is 1), the joined data is compressed. As the joined data is compressed in this binary, we will focus on the compression method.



The two arguments pushed to the function are the location of the compressed data, and the address of the newly allocated heap.

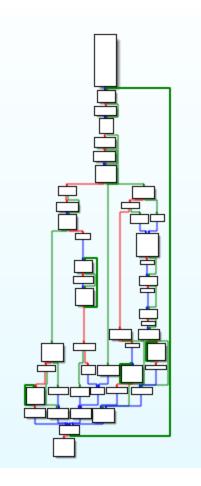


Looking at an overview of the function, you can see it is quite intricate and difficult to follow. Remember what I was saying about crypto constants earlier? They are also present in some, if not all, (de)compression routines, although it is much more difficult to identify the algorithm compared to encryption algorithms.

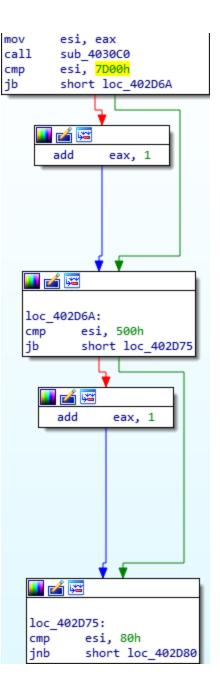
Searching through the graph, there are calls to the same functions several times over, but we are looking for a hardcoded value that is not stored in memory. Sure enough, after searching through the right branch, we can see a CMP instruction being used, comparing the value in esi to the value ox7Doo. Using this, we can search "ox7doo compression" on Google and after scrolling down a bit, come across this **<u>site</u>**. Comparing this python code:

```
def lengthdelta(offset):
    if offset < 0x80 or 0x7D00 <= offset:
        return 2
    elif 0x500 <= offset:
        return 1
    return 0</pre>
```

To a section of the assembly:



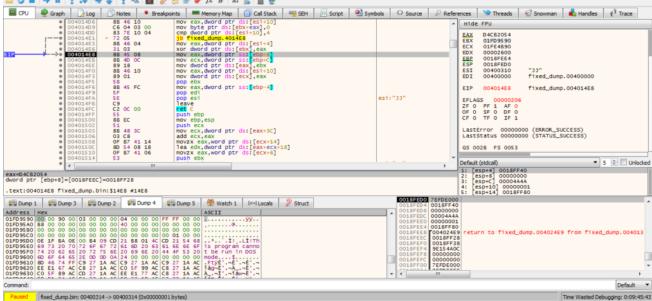
We can see some clear similarities. From this, we can deduce that the decompression algorithm used here is APLib, and with this knowledge we can write a simple extraction script to extract any joined data from the binary and decompress it using the great Python library **Mlib**, created by Mak. You might be wondering how I was able to determine if something was decompressed before even looking at the function. I have read up on ISFB and have also analysed it quite frequently, however it is quite easy to figure out that the joined data is compressed by simply looking at it. Opening up the binary in a hex editor and going to the location of the joined data, you might be able to recognize the string "This Program Cannot Be Run In DOS Mode", although part of it is obfuscated. Recognizing compression also depends on the level of compression, however for the ISFB malware strain, APLib is the defacto compression method for now, so it is relatively simple.



📓 fixed_afterd	ump	.bin															
_																	
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	OF	Decoded text
000083F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00008400	8F	0E	C6	21	D4	03	19	02	04	9C	09	FF	60	70	B8	Α4	Æ!Ôœ.ÿ`p,¤
00008410	01	70	40	85	79	01	87	06	0E	1F	BA	3E	00	Β4	09	CD	.p@y.‡°>.´.Í
00008420	21	B8	FA	4C	C0	AO	54	68	69	73	20	0E	70	72	6F	67	!,úLÀ.This .prog
00008430	67	61	6D	87	63	47	6E	1F	4F	74	E7	62	65	AF	CF	75	gam‡cGn.Otçbe Ïu
00008440	5F	98	69	06	44	4F	7E	53	03	6D	6F	64	65	2E	0D	89	_~i.DO~S.mode‰
00008450	AO	24	4C	43	8D	01	46	74	FF	C9	27	1A	AC	58	04	EE	.\$LCFtÿÉ'.¬X.î
00008460	3E	E1	67	30	C8	11	C0	5F	99	8E	7C	89	54	CD	18	77	>ág0È.À_™Ž ‰TÍ.w
00008470	86	CF	10	74	8A	CA	2C	9F	1B	18	57	11	AO	28	7C	47	†Ï.tŠÊ,Ÿ₩( G
00008480	43	CA	3E	45	21	C8	95	15	10	67	50	6B	C5	EC	4A	80	CÊ≻E!È∙gPkÅìJ.
00008490	60	38	65	62	41	80	52	69	63	68	44	68	68	C7	4C	38	`8ebA.RichDhhÇL8
000084A0	01	05	05	10	1B	4F	5C	63	10	ΕO	80	02	21	0B	33	01	0\c.à€.!.3.
000084B0	08	12	8A	29	1B	26	14	0C	Α5	5A	0B	10	CA	09	A0	OF	Š).&¥ZÊ
000084C0	BD	54	0C	02	92	38	86	34	F1	4A	ЗD	47	0D	29	02	2B	⅓T′8†4ñJ=G.).+
000084D0	Β4	58	39	80	0D	47	12	82	33	11	Α6	02	50	4F	64	44	'X9G.,3.¦.POdD
000084E0	DO	0D	C4	BA	1A	01	Α4	49	<b>A</b> 8	40	23	AO	Β4	Α2	29	80	Ð.ݤI″@# ´¢).
000084F0	C0	1F	58	2E	74	39	65	78	E2	11	OF	88	BA	91	12	EC	À.X.t9exâ^º`.ì
00008500	36	68	40	20	60	07	2E	72	64	61	74	28	0A	D3	0C	FC	6h@ `rdat(.Ó.ü
00008510	E6	0E	09	56	8E	28	8C	40	07	2E	Α4	27	C8	D8	02	95	æVŽ(Œ@¤'ÈØ.∙
00008520	B0	$\mathbf{FD}$	CA	9C	28	F8	C0	2E	38	62	73	DO	0C	DD	70	0A	°ýÊœ(øÀ.8bsÐ.Ýp.
00008530	AC	99	81	58	9E	4A	28	C0	72	65	6C	6F	63	AE	9C	53	⊣™.XžJ(Àreloc®œS
00008540	DO	28	21	AA	5B	9C	8E	OF	AЗ	4A	8F	00	11	D9	CF	B1	Ð(!ª[œŽ.£JÙϱ
00008550	07	D4	64	5E	28	E1	04	D8	0A	84	14	41	41	9C	C8	01	.Ôd^(á.Ø.".AAœÈ.
00008560	E1	DD	D1	8F	C1	DA	01	B0	01	C2	BD	01	FE	01	56	57	áÝÑ.ÁÚ.°.Â⅓.þ.VW
00008570	6A	28	E8	55	4B	80	12	8B	FO	85	F6	74	4F	B0	8F	1D	j(èUK€.∢ð…ötO°
00008580	18	56	89	1C	1C	EE	0C	0E	15	A0	3C	19	10	0C	88	11	.V‱î <^.
00008590	83	F8	FB	03	28	24	74	<b>1</b> E	68	5A	AE	1A	32	6A	01	2A	føû.(\$t.hZ⊗.2j.*
000085A0	8C	23	85	C0	0E	28	20	74	0A	A3	35	54	B2	06	1A	33	Œ#…À.( t.£5T°3
000085B0	FF	EB	17	2E	44	3C	8B	3E	F8	85	01	74	0B	56	E8	9F	ÿëD<<>øt.VèŸ
000085C0	59	80	Α6	EΒ	03	6A	80	5F	8B	73	C7	1D	5E	C3	55	8E	Y€¦ë.j <sç.^ãuž< td=""></sç.^ãuž<>
000085D0	EC	83	46	14	53	DB	7D	27	7D	8D	61	FO	5E	E8	ED	4C	ìfF.SÛ}'}.að^èíL
000085E0	BB	F9	18	DC	4A	F3	D8	85	ЗF	DB	74	00	FF	75	0C	8D	»ù.ÜJóØ?Ût.ÿu

Backing out from the decompression function, the program makes sure that the size of the decompressed executable is the same size as the value stored in the JJ structure. If the sizes do not match, the decompressed executable is cleared from memory, using HeapFree(). Then, the malware XORs the first DWORD of the decompressed data with the XOR key stored in the structure. Taking a look at the decompressed data, we can see that the first DWORD of the executable is not valid – it should look like 00009000, but instead looks like 54205BD4. After the XOR, the valid value can be seen. The function cleans up after itself, and then returns back to the calling function. If you were to dump the executable at this stage and add "MZ" to the start, you would be unable to open it in PE Bear – this is due to the fact that not only is "MZ" missing, "PE" is also missing from the header, so make sure you add that as well.

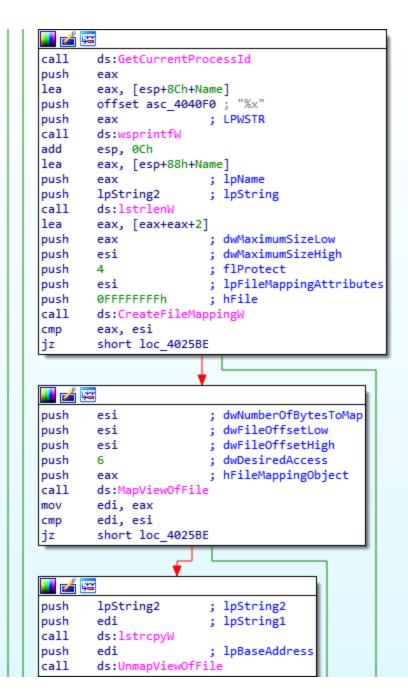
bug Trace Bugins Fevourites Options ■ ■ 2 cm + 2	Image: Second	SEH 💿 Script 💽 Symbols	⇔ Source	ces 🖤 Threads 🔣 Snowman 💼 Handles 🦸 Trace
004014D6 88 46 10     004014D9 66 04 03 00     004014D9 83 7E 10 04     004014DD 83 7E 10 04     004014E1 72 05     004014E1 88 46 04	mov     eax.dword     cal Stack     eax       mov     eax.dword     ptr     ds:[esi+10]       mov     byte     ptr     ds:[esi+10],       mov     dword     ptr     ds:[esi+10],4       jb     fixed_dump.4034E8	EH 🝺 Script 🕙 Symbols	Source & Referen	ces 🖤 Threads 🐨 Snowman 💼 Handles 🕫 Trace
004014D6 88 46 10     004014D9 66 04 03 00     004014D9 83 7E 10 04     004014DD 83 7E 10 04     004014E1 72 05     004014E1 88 46 04	<pre>mov eax,dword ptr ds:[esi+10] mov byte ptr ds:[ebx+eax],0 cmp dword ptr ds:[esi+10],4 jb fixed_dump.4014E8</pre>	SEH 💿 Script 🐏 Symbols	♦ Source	xes 😒 Threads 💰 Snowman 💼 Handles 🧳 Trace
004014D9     C6 04 03 00     004014D0     83 7E 10 04     004014E1     72 05     004014E1     88 46 04	<pre>mov byte ptr ds:[ebx+eax].0 cmp dword ptr ds:[esi+10].4 jb fixed_dump.4014E8</pre>			I HOLE
004014DD     83 7E 10 04     004014E1     72 05     88 46 04	<pre>cmp dword ptr ds:[esi+10],4 jb fixed_dump.4014E8</pre>			Hide FPU
004014E3     8B 46 04     8				EAX D4CB2054
	mov eax,dword ptr ds:[esi+4]			EBX 01FD 95 90
>0 004014E8 88 45 08	xor dword ptr ds:[ebx].eax			ECX 01FE4B90 EDX 00002600
004014EB     8B 4D 0C	mov eax, dword ptr ss: ebp+8 mov ecx, dword ptr ss: ebp+C			EBP 0018FEE4
	mov dword ptr ds:[eax],ebx mov eax.dword ptr ds:[esi+10]			ESP 0018FED0 ESI 00400310 "JJ"
004014F3 89 01	mov dword ptr ds:[ecx],eax			EDI 00400000 fixed_dump.00400000
004014F6 88 45 FC	mov eax, dword ptr ss:[ebp-4]			EIP 004014E6 fixed_dump.004014E6
004014F9 5F     004014FA 5F	pop edi	est:	"11"	
004014FB C9	leave			EFLAGS 00000216 ZE 0 PE 1 AF 1
004014FF     55	push ebp			0F 0 SF 0 DF 0 CF 0 TF 0 IF 1
00401500     88 EC     00401502     51				
00401503 88 48 3C	mov ecx, dword ptr ds:[eax+3C]			LastError 00000000 (ERROR_SUCCESS)
00401506     03 C8     00401508     0F B7 41 14	movzx eax, word ptr ds:[ecx+14]			LastStatus 00000000 (STATUS_SUCCESS)
0040150C 8D 54 08 18     00401510 05 87 41 06	<pre>lea edx,dword ptr ds:[eax+ecx+18]</pre>			GS 0028 FS 0053
e 00401514 53	push ebx			Default (stdcall) 🔹 🗐 Unk
•				Default (stdcal) • 5 🕏 🖾 Unk 1: [esp+4] 0018FF40
				2: [esp+8] 00000000
			1	4: [esp+10] 00000001
26 11Xed_dump.0111.31426 #1426				5: [esp+14] 0018FF80
📲 Dump 3 📲 Dump 2 📲 Dump 4 🚦	🗒 Dump 5 🛛 😨 Watch 1 🛛 💷 Locals 🛛 🖉 St	nuct	0018FED4 0018FF40	
00 00 00 00 00 00 00 40 00 00 00 00	00 00 00		0018FEE0 00000001	
	00 00 00		0018FEE8 004024E9	return to fixed_dump.004024E9 from fixed_dump.00402
1F BA OE 00 B4 09 CD 21 B8 01 4C CD	21 54 68		0018FEEC 0018FF28 0018FEF0 0018FF38	
20 62 65 20 72 75 6E 20 69 6E 20 44	4F 53 20 t be run in DOS		0018FEF4 9E154A0C	
6F 64 65 2E 0D 0D 0A 24 00 00 00 00 46 74 FF C9 27 1A AC C9 27 1A AC C9	00 00 00 mode\$		0018FEFC 00000000	
E1 67 AC C8 27 1A AC CO 5F 99 AC C8	27 1A AC 1ag-E'AE'		ANALAREA . APPREPRI	
SF 89 AC CD 27 1A AC EE EI 77 AC C8	27 1A AC A		•	
				Default
200001732004	<ul> <li>0 04014EP</li> <li>89 18</li> <li>0 04014FP</li> <li>88 46 10</li> <li>0 04014FP</li> <li>89 01</li> <li>9 01</li> <li>9 04014FP</li> <li>88 45 FC</li> <li>0 0401500</li> <li>80 54 08 18</li> <li>0 0401500</li> <li>90 54 09 00 00 00 00 00 00 00</li> <li>18 01 40 00 00 00</li> <li></li></ul>	Borden14EE     By 18     Mov dword ptr ds:[eax],ebx     word ptr ds:[eax+etx]]     word ext,dword ptr ds:[eax+etx]]	• 004014EE 89 18 mov dword ptr ds:[ext].etx         • 004014EF 89 18 mov dword ptr ds:[ext].etx         • 004014F9 89 40 10 mov eax,dword ptr ds:[ext].eax         • 004014F9 89 45 FC mov eax,dword ptr ds:[ext].eax         • 004014F9 55 pop esi         • 004014F9 57 pop esi         • 004014F9 57 72 56 50 20 53 61 61 for 00 70 1 [for, y;         • 004014F9 57 20 50 50 50 50 50 50 50 50 50 50 50 50 50	<pre></pre>



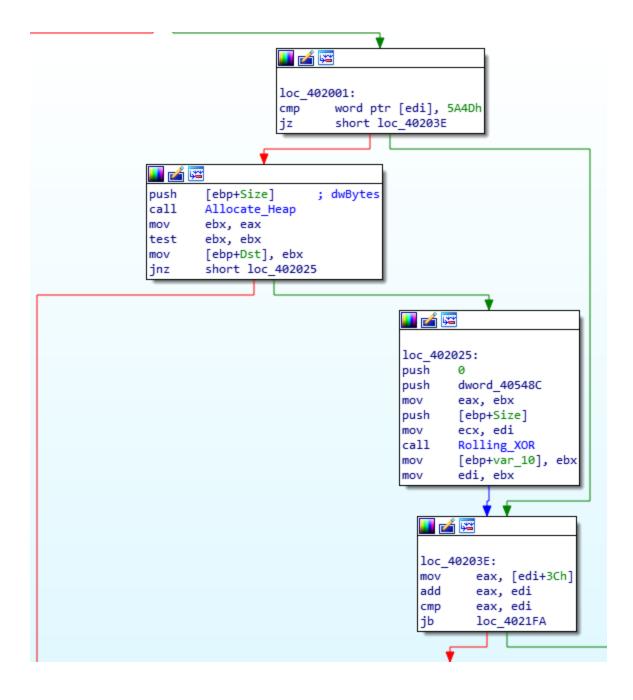
Back in the calling function, the malware then overwrites the decompressed executable using a rolling XOR algorithm. I'm not exactly sure why it does this, as the executable is later decrypted, so if someone could let me know that would be great! Anyway, once the executable has been encrypted, the function carries on.

🕷 x32dbg - File:	le: fixed_dump.b	in - PID: B24 - N	Module: fixed_du	ump.bin - `	Thread: Main Th	hread C94										
File View Deb	ebug Trace P	lugins Favouri	ites Options	Help De	c 31 2017											
👄 🖻 🔳 🔶	÷ II  † ⊅	唑 🎍 🕆	-2 📓 🥜	S 🖉	🥒 fx # 🛛 A	Az 🧾 📃 👮										
🕮 СРИ 🛛 👰 (	💡 Graph 🛛 🛃 L				Memory Map	Call Stack	SEH 25	Script	Symbols	Source	P Refer	ences	😒 Threads	🕄 Snowman	Handles	₹ <sup>9</sup> Trace
312	Ochools     O	04 89 7 06 89 7 07 88 0 08 99 4 12 7 4 75 4 14 75 4 14 88 4 15 88 4	C 24 1C 4 24 10 7 4 24 14 4 24 10 7 24 10 7 24 10 7 04 8 3 0 2 2 04 C 24 14 0 2 2 04 C 2 4 14 0 3 1 1 A A 4 0F	mov do mov ec mov do inc fi inc fi mov ec rol ec add ec xor ec mov do add ec dec do inc fi mov ec and ec sub ec sub ec	word ptr ss: word ptr ss: word ptr ss: word ptr ss: word ptr ss: ax,dword ptr ss; di,4 di,4 di,4 di,4 di,4 di,4 di,4 di,4	[esp+10] ds:[ed] s:[esp+10] [edx],es1 [esp+14] 2514 s:[esp+1C] 546						EAX EBX EDX EDX EDY ESI EDI EIP EFLA ZF 1 OF 0 CF 0 Last Last GS 0	PF 1 AF 0 SF 0 DF 0 TF 1 IF 1 Error 00000	4 000 (ERROR_S 000 (STATUS_	SUCCESS)	5 🕀 🕅 Unlock
ecx=8680 dword ptr [es	esp+1C]=[0018	FF14]=8600										2: [6	esp+8] 7EFDE esp+C] 7FFDE	000		
.text:0040253	534 fixed_dum	p.bin:\$2534	#2534									4: Če	esp+10] 0000 esp+14] 0000	2D 80 0000		
Address         Hex           02289590         000           02289580         DC           02289580         DC           02289500         DC           02289500         DC           02289500         DC           02289500         DC           02289500         CA           022896200         F2           4         CA	x 0 00 20 01 0C E8 2F 01 DC C8 2F 01 DC C8 2F 01 DC C8 2F 01 DC D5 33 3E 8E 00 C2 5E 14 A3 4A BB C4 BC 59 D5 DF 72 32 A9 F0	00 20 01 20 E8 2F 01 DC C8 2F 01 DC C8 2F 01 DC E1 30 E8 83 99 5E 82 22 76 F3 3A 80 FF DA 57 CD ED 5A 19 60	00 20 01 00 C8 2F 01 00 C8 2F 01 00 C8 2F 01 00 S81 3A 2F C 09 EF 32 40 05 F0 38 77 FF DA 57 CC 038 79 22	C C8 2F C C8 2F C C8 2F 1 04 EC C 67 80 9 31 F5 0 FF DA 7 AF 29	ASCII 01 U2/.U2/.U 01 U2/.U2/.U 00 U2/.U2/.U 00 U2/.U2/.U 03 03>.401 53 1.A^^. 79 .£J>Ävó:. 57 .XYÔ&ýÚMJ 88 0720012.2		Struct		,	<ul> <li>0018FF00</li> <li>0018FF04</li> <li>0018FF05</li> <li>0018FF10</li> <li>0018FF14</li> <li>0018FF14</li> <li>0018FF14</li> <li>0018FF16</li> <li>0018FF10</li> <li>0018FF20</li> <li>0018FF20</li> </ul>	00000000 0000000 7EFDE000 7FFDEFFF 00002080 0000000 0000000 00000000 00000000					,
Command:																Default
Paused Dum	ump: 02289590 ->	02289590 (0v00	000001 bytes)												Time Wasted D	ebugging: 0:10:04:

First, it creates a new file mapping, and calls MapViewOfFile, and then copies it's filename to the newly mapped region, before calling UnmapViewOfFile. Again, I'm not too sure why this is being called, as it doesn't seem to do anything, but if someone could drop an answer to this that would be great, and I can incorporate it into the post.



Moving on, the final function in this sample is called. This is quite a large function, so I will attempt to summarize the main points. Firstly, the function begins by moving different pointers around, and then compares the first 2 bytes of the encrypted executable to the value 0x5A4D (MZ) to check if the data is encrypted or not. As it is encrypted, the result will not be 0, and so it will allocate a heap based on the size seen in the embedded JJ structure. Next, it will call another function responsible for performing another rolling XOR algorithm, that will decrypt the data. This is a bit different to the last one, but should return the same executable that was decompressed earlier. Then, it will locate the value in the PE header that indicates the architecture of the executable, which in this case is 0x14C, meaning it is x86 based.





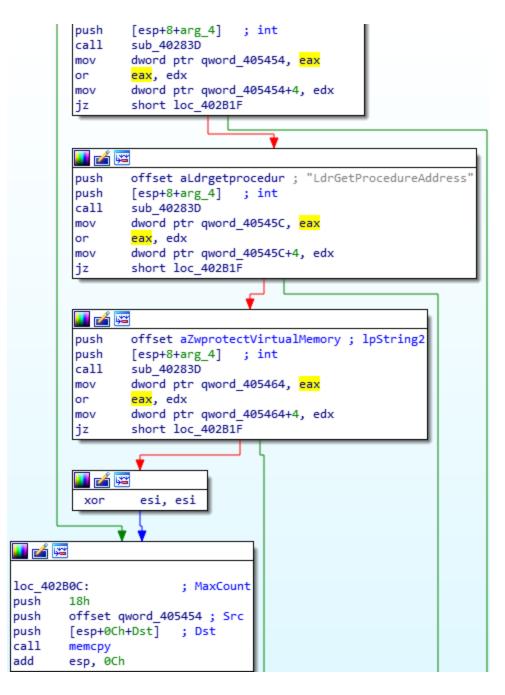
Next, the malware will allocate a brand new section of memory using NtCreateSection(), which will be set to Read-Write-eXecute. Eventually, this will contain the decompressed executable.

```
ULONG stdcall Create Section(int a1, int a2, void *SectionHandle)
{
  DWORD *v3; // ebx
 NTSTATUS v4; // eax
 ULONG v5; // edi
 struct _OBJECT_ATTRIBUTES ObjectAttributes; // [esp+10h] [ebp-2Ch]
 union _LARGE_INTEGER MaximumSize; // [esp+28h] [ebp-14h]
 void *Dst; // [esp+34h] [ebp-8h]
 v3 = SectionHandle;
 MaximumSize.QuadPart = (unsigned int)a1;
 ObjectAttributes.Attributes = 64;
 SectionHandle = 0;
 Dst = 0;
 ObjectAttributes.Length = 24;
 ObjectAttributes.RootDirectory = 0;
 ObjectAttributes.ObjectName = 0;
 ObjectAttributes.SecurityDescriptor = 0;
 ObjectAttributes.SecurityQualityOfService = 0;
 v4 = NtCreateSection(&SectionHandle, 0xF001Fu, &ObjectAttributes, &MaximumSize, 0x40u, 0x800000u, 0);
 if ( v4 < 0 )
  {
   v5 = RtlNtStatusToDosError(v4);
 }
  else
  {
    v5 = Map_View_Of_Section(SectionHandle, (HANDLE)0xFFFFFFFF, &Dst);
   if ( !v5 )
   {
     memset(Dst, 0, MaximumSize.LowPart);
      *( DWORD *)a2 = Dst;
      if ( v3 )
       *v3 = SectionHandle;
   }
  if ( SectionHandle && !v3 )
   ZwClose(SectionHandle);
 return v5;
}
```

From there, an address located inside the newly created section of memory will be formed, pointing to 0x0022EC50, which will be used later. The created section of memory is replicated, to the address 0x00240000. Next, the program begins to copy over the executable to the new addresses, however it skips the entire MZ header and simply copies everything from the PE header.

fixed_dur	np.ł	bin (	2852	2 <b>) (0</b> )	×220	000	- 0x	2360	)00)									
00000000	<b>B</b> o	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	
											_	_					5 5	<u> </u>
																	5 0	
																	· · · · · · · · · · · · · · · · · · ·	
																	5 0	
																	······	
																	5 5	
																	D	
																	D	
					_	_					_	_	_				D	
																	0	
																	D	
																	D	
000000d0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	D	
000000e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	
				_	_	_					_	_					D	
																	0 L 0\	
00000110	00	00	00	00	e0	00	02	21	0b	01	08	00	00	8a	00	00	0!	
00000120	00	26	00	00	00	00	00	00	a5	5a	00	00	00	10	00	00	D.&Z	
00000130	00	a0	00	00	00	00	00	10	00	10	00	00	00	02	00	00	0	
00000140	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	0	
00000150	00	e0	00	00	00	04	00	00	00	00	00	00	02	00	00	00	0	
00000160	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	0	
00000170	00	00	00	00	10	00	00	00	a0	ac	00	00	33	00	00	00	0	
00000180	00	аб	00	00	50	00	00	00	00	00	00	00	00	00	00	00	0P	
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	
000001a0	00	<b>d</b> 0	00	00	c4	04	00	00	00	00	00	00	00	00	00	00	0	
000001b0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	
000001c0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	
																	D@	
																	0	
000001f0	00	00	00	00	00	00	00	00	2e	74	65	78	74	00	00	00	0text	
																	٥	
																	D	
																	0.rdata	
						_					_	_	_				0	
																	0@@.data	
00000250																00	0	-
										-						<u></u>	· "	

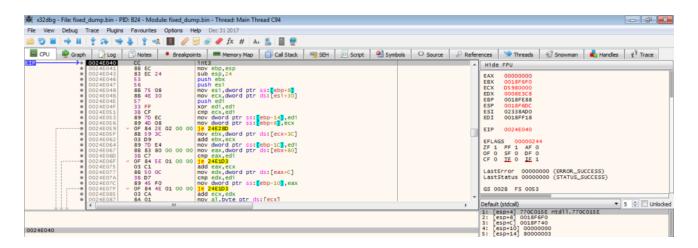
Then, a function is called that imports some more API calls from NTDLL, however this time it is not done using CRC hashes, and the name is simply passed as an argument. The imported API calls are; LdrLoadDll, LdrGetProcedureAddress, and ZwProtectVirtualMemory. The pointers to the addresses are stored in the same region of memory, split by 4 null bytes each. These addresses are then copied over to the executable at 0x00220000 for usage during it's execution. Next, a region of code is copied over to the executable in memory, just 40 bytes after the loaded APIs. This is what will be called before the next stage is completely executed.



Continuing on, the next function is responsible for passing execution over to the executable. There are 2 functions that pass over execution to the executable, however they depend on the architecture. In this case, I will be looking at the x64 version. It is quite simple, as all it does is call the function at the address 0x0022E040, which will prepare the next stage. Therefore, you can think of this stage as another "unpacker", as all it does is unpacks the executable, however there are many functions that carry over to the next stage, making it much easier to analyse.

Ķ x32dbg - File: fixed_dump.bin - PID: B24 - Module: fixed_dump.bin - Thread: Main Thread C94	
File View Debug Trace Plugins Favourites Options Help Dec 31 2017	
🖴 😳 🔳 字 🖩 🙄 🖓 🛬 🌲 🖹 🥐 纪 📕 🥜 😓 🖉 🦧 🎁 🗛 📗	
🔟 CPU 👰 Graph 🗋 Log 🗋 Notes 🔹 Breakpoints 📟 Memory Map 🔲 Call Stack 🧠 SEH 💿 Script 🔮 Syn	mbols 🗘 Source 🔎 References 😒 Threads 😴 Snowman 🜲 Handles 🐔 Trace
Ord0243C     FF 15 50 40 40 00     Ord0243C     FF 15 50 40 40 00     Ord0244C     FF 15 50 40 00     Ord0244D     FF 15 10     Ord0244D     FF 15 10     Ord0244D     Source     Ord0244     Source	Call next stage Call next stag
word ptr [esi+8]=[02338AD8]=0024E040	1: [esp] 0024E000 <&LdrLoadD11> 2: [esp+4] 00000318 3: [esp+8] 0000008
text:0040244A fixed_dump.bin:\$244A #244A	4: [esp+C] 00000000 5: [esp+10] 00000000
Constraint         Constra	ODJELETAL         0024E000           ODJELETAL         00246000           ODJELETAL         0000000           ODJELETAL         000000
ommand:	Default
Paused Dump: 0024E000 -> 0024E000 (0x00000001 bytes)	Time Wasted Debugging: 0:11:

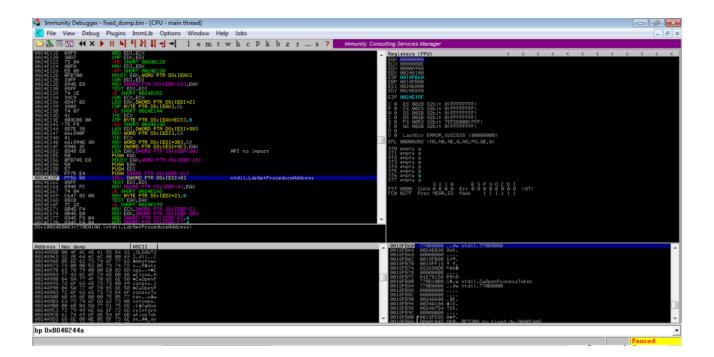
When we step into the function, there is an int3 waiting for us, which acts as a breakpoint that raises an exception, stopping us from stepping over it. To fix this, we can put a breakpoint on the previous call, restart the debugger and then run it until the breakpoint is hit. This should hopefully remove the int3.



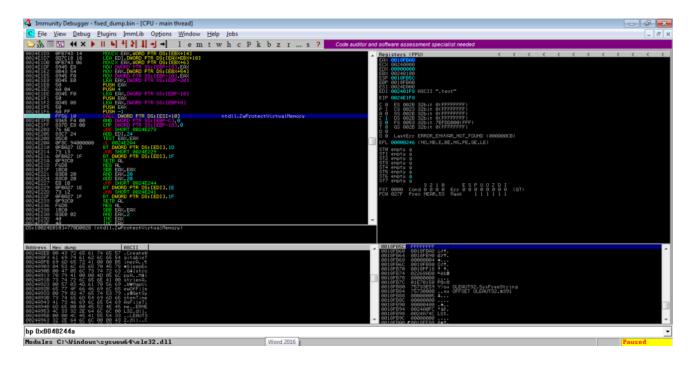
As it didn't remove it in my case, I will use Immunity Debugger to carry on this section. The code is responsible for importing DLLs using the previously imported API's – First, there is a loop which stores a DLL name in memory that will be loaded using LdrLoadDll. Then, a list of hardcoded APIs are looped over, with each being passed to LdrGetProcedureAddress. The DLLs that are loaded are; NTDLL.DLL, KERNEL32.DLL, AND OLEAUT32.DLL.

Image: Set in the left of the left	🗳 Immunity Debugger - fixed_dump.bin - [CPU - main thread]	
Address grad       Address grad	C Eile Yiew Debug Plugins ImmLib Options Window Help Jobs	_ <i>8</i> ×
Contraction       Write DLL have to memory         Contraction       Write DLL have to memory         Contraction       Contraction         Contraction		ilting Services Manager
Add:46322 db 11 20 65 48 00 65 33 : / Pb.03       Add:4532 db 11 20 65 48 00 65 23 : / Pb.03         Add:46322 db 11 20 65 48 00 65 23 : / Pb.03       Add:4532 db 11 20 75 41 48         Add:4632 db 11 20 65 48 00 65 23 : / Pb.03       Add:45 20 75 41 16 75 16 75 41 16 75 16	Docketopy         Contraction         Docketopy         Contraction           Docketopy         Contraction         Docketopy         Contraction           Docketopy         Contraction         Docketopy         Docketopy           Docketopy         Docketopy         Docketopy         Docketopy           Docketopy         Docketopy         Docketopy         Docketopy           Docketopy         Docketopy         Docketopy         Docketopy           Docketopy         Docketopy         Docketopy         Docketopy	Ch: 00197004           CD: 002420100           ED: 00240100           ED: 00240100           ED: 00240100           ED: 00240100           ED: 00240100           ED: 0024000           ED: 0025000           ED: 0025000           ED: 0025000           ED: 0025000           ED: 0025000           ED: 00200000           ED: 00200240
Paused	0024E832 00 11 20 05 04 80 05 53 31 (P\$.05) 0024E832 00 05 05 04 80 05 54 00 05 04 00 05 0024E830 42 05 05 04 04 05 05 05 05 04 00 05 04 00 05 04 01 00 0024E8504 02 00 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 0024E852 00 00 00 00 00 00 00 00 00 000 00 00 00 00 00 00 00 00 00 00 000 00 00 00 00 00 00 00 00 00 00 000 00 00 00 00 00 00 00 00 00 00 00 000 00 00 00 00 00 00 00 00 00 00 00 00	Weiserber devisioned reise deserber devisioned
		Paused

🗳 Immunity Debugger - fixed_dump.bin - [CPU - main thread]	
C Eile View Debug Plugins ImmLib Options Window Help Jobs	_ <i>8</i> ×
🗀 🐝 🗏 🕊 🗙 🕨 II 🖣 📲 🔰 📲 lemtwh c Pkbzr s ? Code audi	tor and software assessment specialist needed
Addet 198         Start         Note DULPOIT           Addet 198         Start         Note DULPOIT           Addet 198         Start         Note DULPOIT           Addet 198         Start         Start           Addet 198         Start         Start	<pre>Registers (FPU)</pre>
Eddress         Hex. dunc.         ROTI         ROTI           000240908         74         74         65         60         65         51         10           000240908         74         74         65         60         65         11         10           000240908         74         74         65         60         64         64         11         10           000240908         60         65         65         45         87         11         10           000240908         60         45         55         458         10         11         10           000240908         60         47         45         65         458         10         11         10           000240908         60         47         45         155         54         38         10         11         10           000240908         67         67         10         60         43         2.51         11         10         10         10         10         10         10         10         10         10         10         10         10         10         10         10         10         10         10	00157201         77600000          Am indii.77600000           00157202         00157202         00157202         00157202           00157202         00157202         00157202         00157202           00157202         00157202         00157202         00157202           00157202         00157202         00157202         00157202           00157202         00157202         00157202         00157202           00157202         01157202         01157202         01157202           00157202         000000000         01157202         000000000           00157202         0000000000         01157202         000000000           00157202         0000000000         01157202         000000000           00157202         0000000000         01157202         000000000           00157202         0000000000         01157202         0000000000           00157202         0000000000         01157202         000000000000000000000000000000000000
bp 0x0040244a	•
	Paused



Once the APIs have been imported, ZwProtectVirtualMemory is called several times in a loop to alter the protections on several regions of memory.



In order to jump to where the execution of the payload happens, we want to put a breakpoint on a call to a register, which in this instance is EBX. Run to that, and then step into it, and you should find yourself in the next stage of ISFB!

🗳 Immunity Debugger -	fixed_dump.bin - [CPU - main thread]	
C File View Debug	<u>Plugins</u> ImmLib Options <u>Window</u> <u>H</u> elp Jobs	
🗁 🚴 🗏 🔣 🕂 🗙 🕨		Immunity: Consulting Services Manager
0024E243 50 0024E244 8B4F E4 0024E247 894D F0	FXX         Discrete FTR SociEEP1-EC3           FXX         Discrete FTR SociEEP1-B3           FXX         ECX, DWORP FTR SociEEP1-B3           FXX         ECX           Expecting Control FTR SociEEP1-B3           FXX         ECX           Expecting Control FTR SociEEP1-B3           FXX         ECX           Expecting Control FTR SociEEP1-B3           EXPECTING	▲ Registers (FPU) < < < < < < < < < < < < < < < < < < <
0024E244 804F E4 0024E247 8940 F0 0024E24A 884F E8 0024E24A 884F E8 0024E24D 804D F8 0024E258 8940 F8 0024E258 8040 F8	HOU ECX, DWORD FTR SSIEEDF+83 HOU DWORD PTR SSIEEDF+83 HOU DWORD PTR SSIEEDF-93,ECX	EDX 0009E3C8 EDX 00245AA5 ESX 017E5A4
00246265 51	5050 550	ESP 0019FE04 ESP 0019FE00 ES1 0024E018 ED1 0024E018
0024E259 8045 F0 0024E258 50 0024E258 50 0024E25C 8045 F8	PUSH ERA LEA ERA,DUORD PTR SS:LEBP-10] PUSH ERA LEA ERA,DUORD PTR SS:LEBP-0]	EIP 0024E286
0024E25F 50 0024E25F 50 0024E260 6A FF	PUSH ERX	C 0 ES 0428 32bi 0 (FFFFFF) P 1 C5 0425 32bi 0 (FFFFFFF) A 0 55 0428 32bi 0 (FFFFFFF) 2 0 55 0428 32bi 0 (FFFFFFFF)
0824E262 FF66 10 0824E265 83C7 28 0824E268 FF45 F4	PUER EN (TIL DUGED PTR DS:(ESI:18) ACD EDI:28 INC DUGED PTR SS:(EEP-C)	2 0 DS 0022 325it 0(FFFFFFF) S 0 FS 0053 325it 7/FFD0000(FFF) T 0 65 0028 325it 0(FFFFFF)
0824E268 8B4D F4 0824E26E 3B4D E8 0824E271 ^72 95	ACC EDT, 28 (2014) TO DOCTORER SQL(EBP-C) TO DOCTORER SQL(EBP-C) COMP ECX, DOCHO PTR SSL(EBP-C) COMP ECX, DOCHO PTR SSL(EBP-C) SQL(EBP-C) EXX, DOCHO PTR SSL(EBP-C) COMP ECX, DOCHO PTR SSL(EBP-C) COMP EXX, DOCHO PTR	0 0 0 0 LastErr ERROR_ENVUAR_NOT_FOUND (000000CB)
al224225F 50 al224256 al242266 AFF6 al242268 AFF6 al242268 BB40 F44 al24227 BB40 F44 al24227 BB40 F44 al24227 BB50 B26 BB40 F44 BB20 E276 BB50 B2 BB50 B3 BB50 B3 B350 B350 B3 B350 B350 B3 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B350 B3	CHP LCC, CAUGO FMT SSILEEF-18] TEST EXC, CAUGA CAU TEST EXC, CAUCAR CAU CONTRACT NEXT AND A SSILEEF-18] TRO EEX. (NOW OF THE SSILEEF+63) RCO EEX. (NOW OF THE SSILEEF+63) FUGH CAUGO FTE SSILEEF+63] FUGH CAUGO FTE SSILEEF+63] FUGH CAUGO FTE SSILEEF-163. FEX.	EFL 00000206 (H0,H8,HE,A,MS,PE,6E,G)
0024E27A 035D 08 0024E27D 8356 18	ADD EBX, DAURO PYR SS: [EBP+8] ADD ESI, 18	STI anoty 0 STI anoty 0 STI anoty 0 STI anoty 0 STE anoty 0 STE anoty 0 STI anoty 0 STI anoty 0 STI anoty 0 STI anoty 0 STI anoty 0
0024E280 56 0024E281 6A 01 0024E283 FF75 08	PUSH ESI PUSH DNORD PTR SSILEBP+8]	ST4 empty g ST5 empty g ST6 empty g
0024E280 56 0024E281 6A 01 0024E283 FF75 08 0024E283 FF75 08 0024E288 8P45 EC 0024E288 EB 17 0024E288 EB 17 0024E280 8045 DC	10 SHOPT 00245204	517 emoto 5 557 emoto 5 551 0000 Cond 8 0 8 0 E 5 9 0 0 0 0 0 0 0 0 FCU 027F Prove FERR 5 1 Mak 1 11 11 11
0024E28D 8D45 DC 0024E290 50 0024E291 8D46 38	LEN ERK DUCKD FTR SS:LEEP-24) PURE ERK DUCKD FTR SS:LEEP-24) ERK ERK DUCKD FTR DS:LESI+38)	FCM 027F Prec NEAR,53 Mask 1 1 1 1 1 1
0024E294 50 0024E295 57 0024E295 57	PUSH EDI	
FF16 8824E299 8508	PUSH EDI CALL DWORD PTR DS: (ESI) TEST FRX.FRX	*
EBX100245HH5		
Address Hex dump	ASCII	0010FD64 00240000
082448E8 08 43 72 65 61 0824A8F3 61 69 74 61 62 0824A8F8 69 60 65 72 41	19511         1           2655         1           2655         1           2655         1           2655         1           270         4577           270         4577           270         4577           270         4577           270         4577           270         4577           270         4577           270         4577           270         5575           271         10001           272         10001           273         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         10001           274         100000000	0018FB6C 0024E018 1∞5. 0018FF70 0018FF18 1 1. 0018FF74 0225AD0 +6.00
00240903 04 53 6C 65 65 00240908 00 47 05 6C 73 00240913 70 79 41 00 00	70 45 78 +SleepEx 74 72 63 .641stro 40 05 65 pug. M&I	0015FB74 002205HD0 ~040 0015FB72 00000000 0015FB7C 01E70150 P0x0
00240918 73 74 72 6C 65 00240923 00 57 03 4D 61 00240929 65 77 45 62 65	6E 41 00 strienA. 70 56 69 .WWHADU	0015FB68 00000040 0 0016FB64 75730000sw OFFSET OLEMUT32.#391 0015FB68 0000005 4
00240920 05 77 4P 06 40 00240933 00 79 02 47 65 00240938 73 74 65 60 54	69 60 65 stenTine	0115F50C 00001000 0115F59C 00001000 0115F594 00000005 0115F595 00240000
00240943 41 73 46 69 60 00240948 60 65 00 00 45 00240953 4C 33 32 25 64	65 54 69 HAFILETL 52 4E 45 me.,ERHE 66 60 00 L32,d11,	0015F595 0024D000 .45. 0015F595 0000000 0015F596 0015F535
0024A958 00 00 4C 45 41 0024A963 32 2E 64 6C 6C	55 54 33LEAUT3 00 00 43 2.dllC	01355BR0 00107583 001 RETURN to fixed_dw.0040244D
bp 0×0040244a		·
[07:17:02] Breakpo	pint at 0024E286	Paused
4 Immunity Debugger -	fixed_dump.bin - [CPU - main thread]	
	<u>P</u> lugins ImmLib Options <u>W</u> indow <u>H</u> elp Jobs	- 8 ×
🗀 🚴 🗏 🔣 🕂 🗙 🕨	• II • • • • • I emtwhcPkbzrs?	
00245AA9 56 00245AA9 56 00245AAA 33F6	PUSH ESI XOR ESI.ESI	▲ Registers (FFU) < < < < < < < < < < < < < < < < < < <
88245AAA 8376 88245AAC 46 88245AAC 83E8 00 88245AB0 74 23 88245AB2 48 88245AB3 75 34 88245AB3 68 04812480	PUG EST PUG EST PUG PUG EST PUG EST	EDX 0608E3C8 EDX 06245A96 EDX 06245A96
00245AB2 48 00245AB3 75 34 00245AB5 49 04012400	DEC ERX URL SHORT 00245RE9 PUSN 240104	ESP 601 (BPR60 EFP 001 0FPR01 EST 0024E019 
00245ABA FF15 10A02400 00245AD0 38C6	CHL DWORD PTR DS:[24A010] kernel32.InterlookedInorement	EIP 00245A95
00245082 48 00245083 75 34 00245083 75 34 00245086 68 D4812400 00245087 FF15 10802400 0024500 3805 0024500 75 25 0024500 FF7424 08 0024500 FF7424 08 0024500 FF7424 08	************************************	C 0 ES 0628 32bit 0(FFFFFFFF) P 1 CS 0623 32bit 0(FFFFFFFF) A 0 ES 0628 32bit 0(FFFFFFFF)
08245ACD 8508 08245ACF 74 18 08245AD1 33F6	TEST ERX,ERX JE SHORT 00245AE9 XOR ESI,ESI	2 0 DS 002B 32bit 0(FFFFFFF) S 0 FS 005B 32bit 7EFD0000(FFF) T 0 GS 002B 32bit 0(FFFFFFF)
002458CD 8508 002458CF 74 18 002458CF 74 18 002458D3 33F6 002458D3 EB 14 002458D5 68 D4812400 002458D5 68 D4812400 002458D5 68 14802400	COLL 00244277 TEST EXX. 00246279 We Shift WorkSame 9 Prove State 20 Prove State 2	0 0 0 0 LastErr ERROR_ENWAR_NOT_FOUND (000000CB)
00245AE0 8500 00245AE2 75 05 00245AE2 75 05 00245AE4 E8 BE0E0000 00245AE9 8806	TEST EAX, EAX UR2 SHORT 08245AE9	EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G) ST0 empty g
00245AE9 88C6 00245AE9 5E 00245AE8 5E 00245AEC C2 0C00	HOU EXX,ESI POP ESI	STO eneru e STI eneru e
08245AEC C2 0C00 08245AEF 55 08245AF0 8BEC	RETN &C PUSH EBP NOU EBP, ESP	ST4 empty g ST5 empty g
00245AF2 51 00245AF3 53 00245AF4 8850 08	PUSH ECX PUSH EBX MOV EBX,DWORD PTR SS:[EBP+0]	
00245RF7 56	PUSH EDI	911 9009 9 3 2 10 E E P U D Z D I PST 0000 Cond 0 0 0 E E P 0 0 0 0 0 0 0 0 (GT) FCW 027F Prec NEAR,53 Hask I I I I I I I
00245AF9 6A 20 00245AF9 6A 20 00245AFB 88F9 00245AFD 33C9	PUSH 20 HOV EDT.ECX XOR ECX.ECX	
00245HFF 5E 00245B00 2B75 0C 00245B03 8940 FC	POP ESI SUB ESI, DWORD PTR SS:(EBP+C) UDU TWORD PTR SS:(EBP-4).FCK	*
Stack SS:[0018FB68]=000 EAX=00000000	R0601	
Address Hex dump	ASCII	0016F660 0024E288 61s, RETURN to 0024E288
0024H8E8 00 43 72 65 61 0024A8F3 61 69 74 61 62 002408F8 69 60 65 72 41	74 65 57 .CreateW 66 65 54 altableT 80 00 85 imerAM	0018FB68 00000001 0 0018FB6C 0024E018 1+5. 00186F95 0019E10 + +
	70 45 78 eSleepEx 74 72 63 .6#istrc	

bp 0×0040244a

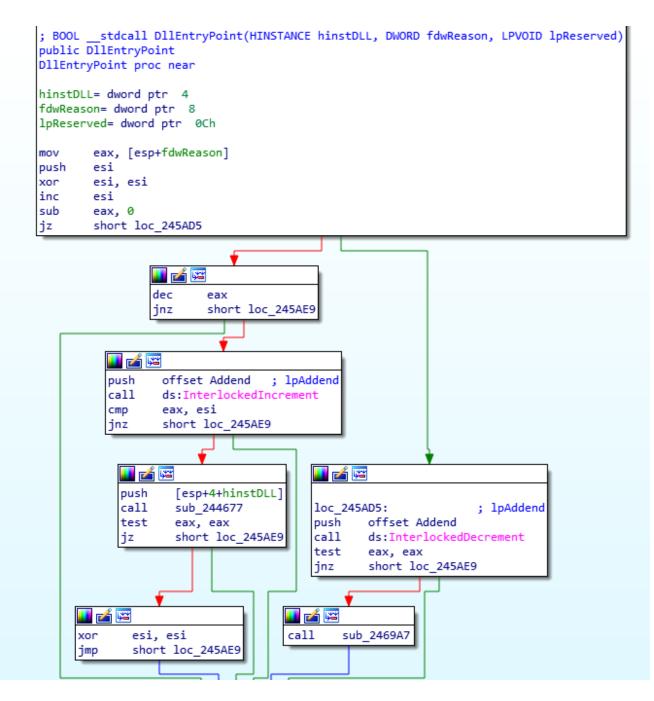
All we need to do now is to dump it out, add "MZ" and "PE" to the header, unmap and rebase it using PE Bear, and then we can start analysing the next stage!

FFSET OLEAUT32.#391

Offset(h)	00	01	02	03	04	05	06	07	80	09	0A	0B	0C	0D	0E	OF	Decoded text
00000000	4D	5A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	MZ
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	•••••
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000100	50	45	00	00	4C	01	05	00	10	1B	4F	5C	00	00	00	00	PEL0\
00000110	00	00	00	00	ΕO	00	02	21	0B	01	08	00	00	8A	00	00	à!Š
00000120	00	26	00	00	00	00	00	00	A5	5A	00	00	00	10	00	00	.&¥Z
00000130	00	A0	00	00	00	00	00	10	00	10	00	00	00	02	00	00	
00000140	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	
00000150	00	ΕO	00	00	00	04	00	00	00	00	00	00	02	00	00	00	.à
00000160	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	
00000170	00	00	00	00	10	00	00	00	A0	AC	00	00	33	00	00	00	
00000180		A6	00	00	50	00	00	00	00	00	00	00	00	00	00	00	P
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001A0	00	DO	00	00	C4	04	00	00	00	00	00	00	00	00	00	00	.ÐÄ
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	•••••
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000001D0	00	00	00	00	00	00	00	00	00	A0	00	00	40	01	00	00	····
000001E0	B4	A2	00	00	C0	00	00	00	00	00	00	00	00	00	00	00	´¢À
000001F0	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	00	00	text

	0 1 2 3 4 5	6789				456789	ABCDEF		
0	4D 5A 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00		M Z				
10	00 00 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00						
20	00 00 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00						
30	00 00 00 00 00 00	00 00 00 0	0 00 00 00 01 00 00						
40	00 00 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00						
50	00 00 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00						
60	00 00 00 00 00 00	00 00 00 0	0 00 00 00 00 00 00						
Disasm	General DOS Hdr	File Hdr	Optional Hdr Section	Hdrs Exports	Imports	BaseReloc.	DelayedImps	]	
		File Hur	Optional Hur Section	Hurs Exports	imports	BaseReloc.	Delayeumps		
÷ +	- 8								
Offset	Name	Func. Coun	t Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk	
A600	ntdll.dll	15	FALSE	A750	0	0	A7A4	A100	
A614	KERNEL32.dll	58	FALSE	A650	0	0	A94E	A000	
A628	OLEAUT32.dll	4	FALSE	A73C	0	0	A95C	A0EC	
Details									
	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint			
Details Call via A100	Name ZwOpenProces		Original Thunk A99A	Thunk A99A	Forwarder	Hint 449			A
Call via			-		Forwarder				
Call via A100 A104	ZwOpenProces		A99A	A99A	Forwarder - -	449			
Call via A100 A104 A108	ZwOpenProces ZwOpenProcess		A99A A98A	A99A A98A	Forwarder - -	449 448			
Call via A100 A104 A108 A10C	ZwOpenProces ZwOpenProcess ZwClose	-	A99A A98A A980	A99A A98A A980	Forwarder - - -	449 448 3E0 552 575			
Call via A100 A104 A108 A10C A110	ZwOpenProces ZwOpenProcess ZwClose strcpy	-	A99A A98A A980 A976	A99A A98A A980 A976	Forwarder - - - -	449 448 3E0 552			
Call via A100	ZwOpenProces ZwOpenProcess ZwClose strcpy wcstombs	-	A99A A98A A980 A976 A980	A99A A98A A980 A976 A980	Forwarder - - - - -	449 448 3E0 552 575			E

Check for updates



MD5 of Dumped DLL: 52b4480de6f4d4f32fba2b535941c284

Congratulations! You have managed to analyse the loader and "unpack" the next stage, which I will be analysing in the next post (because this one has now amassed over 6,000 words which is much longer than I planned). So, feel free to ask any questions you have down below, or over Twitter (@overflow\_) and I will be glad to answer them! I apologize again for the lack of posts recently, I've been working on my course as well, so I've had a lot of stuff to do! Hopefully the next post on ISFB shouldn't take too long to do, so make sure to sign up to my mailing list to stay updated whenever I post! Thanks again

# 2 Comments

Comments are closed.