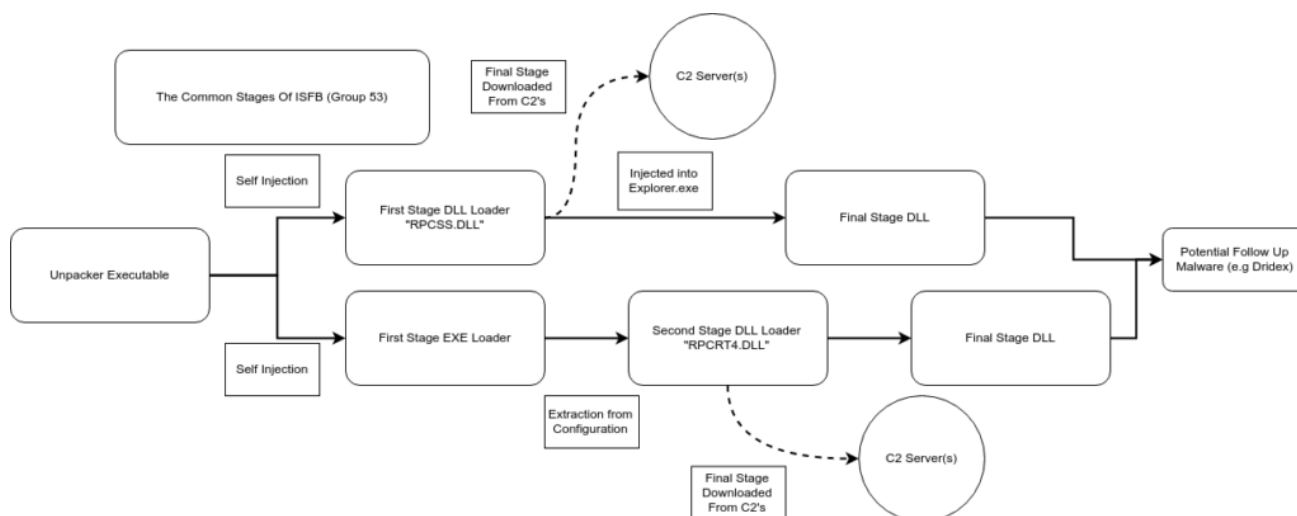


Analysing ISFB – The First Loader

- overflow
- 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 **rpqrt4.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 **VirusBay**. 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 [@VK Intel](#), [@Nazywam](#), and [@Maciekkotowicz](#) (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

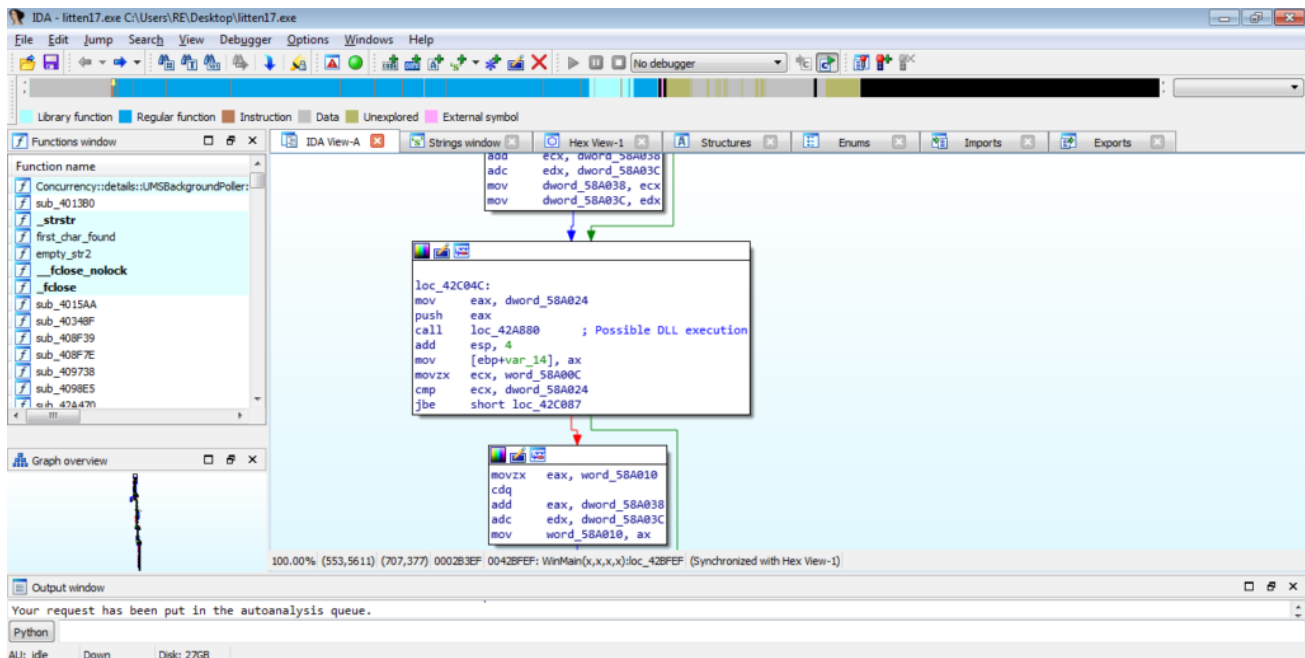
Typically I would open it up in PEStudio or perhaps perform some static analysis, however at this point I have already determined that it is highly likely to be a sample of ISFB, based on the Word Macro, although we still want to unpack the first EXE to be 100% sure that it is in fact ISFB. First, let’s 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.



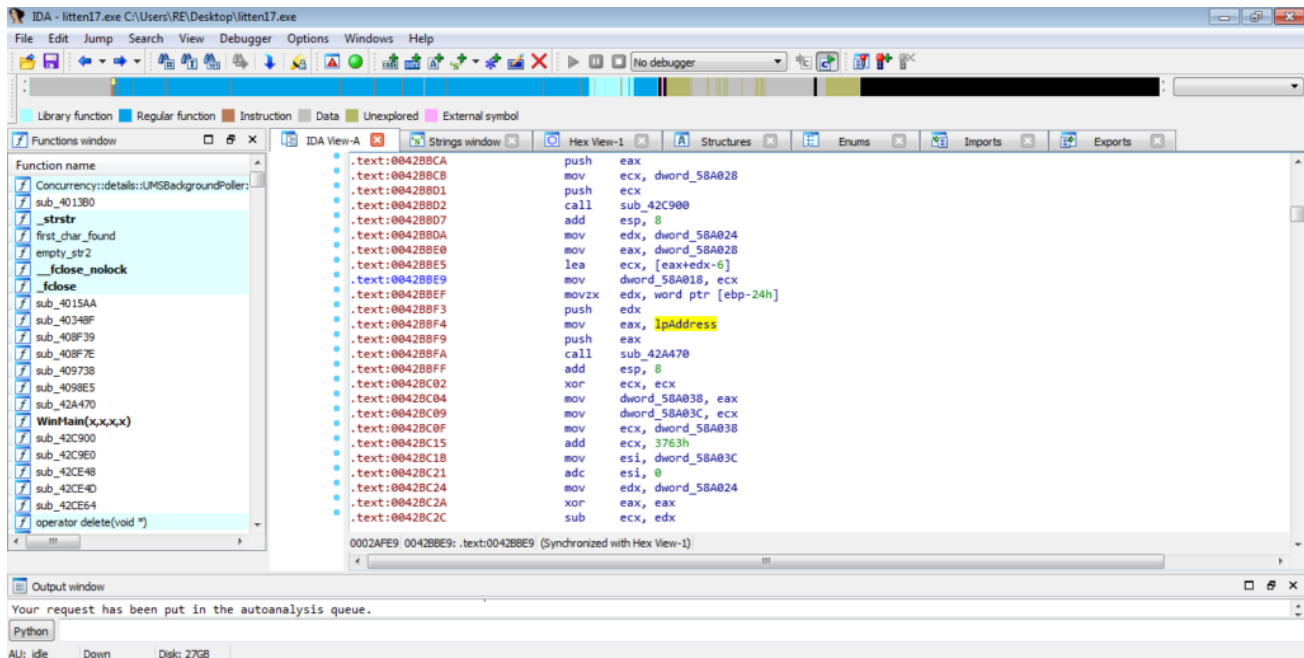
It definitely looks like there is some unpacking going on here (based on the length and intricate flow shown in the image above), and upon looking at the strings in the binary we can see that there aren’t many that are legible, or meaningful. Normally when unpacking a

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 ununpacker 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 ununpacker 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.

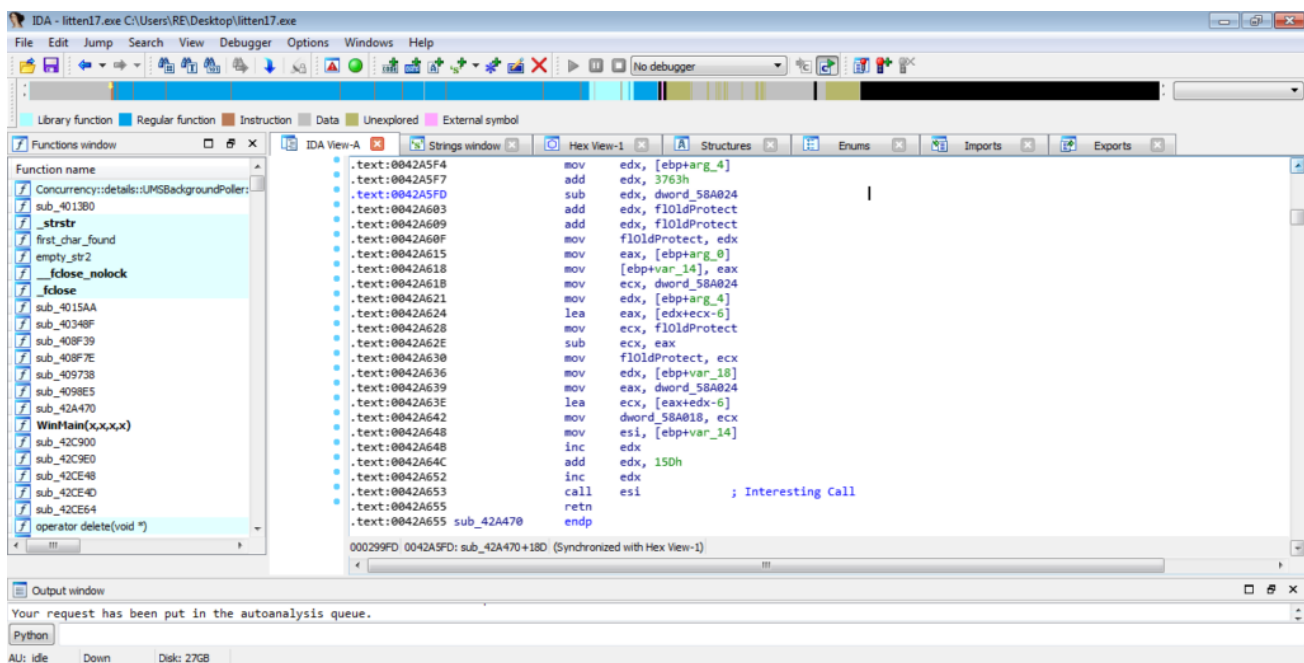
In this specific file, there is no call or jmp to a register or memory region – however, there is a call to **loc_42A880**. It's the last call in WinMain, so as mentioned before, this function is most likely responsible for transferring execution over to the EXE.



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.

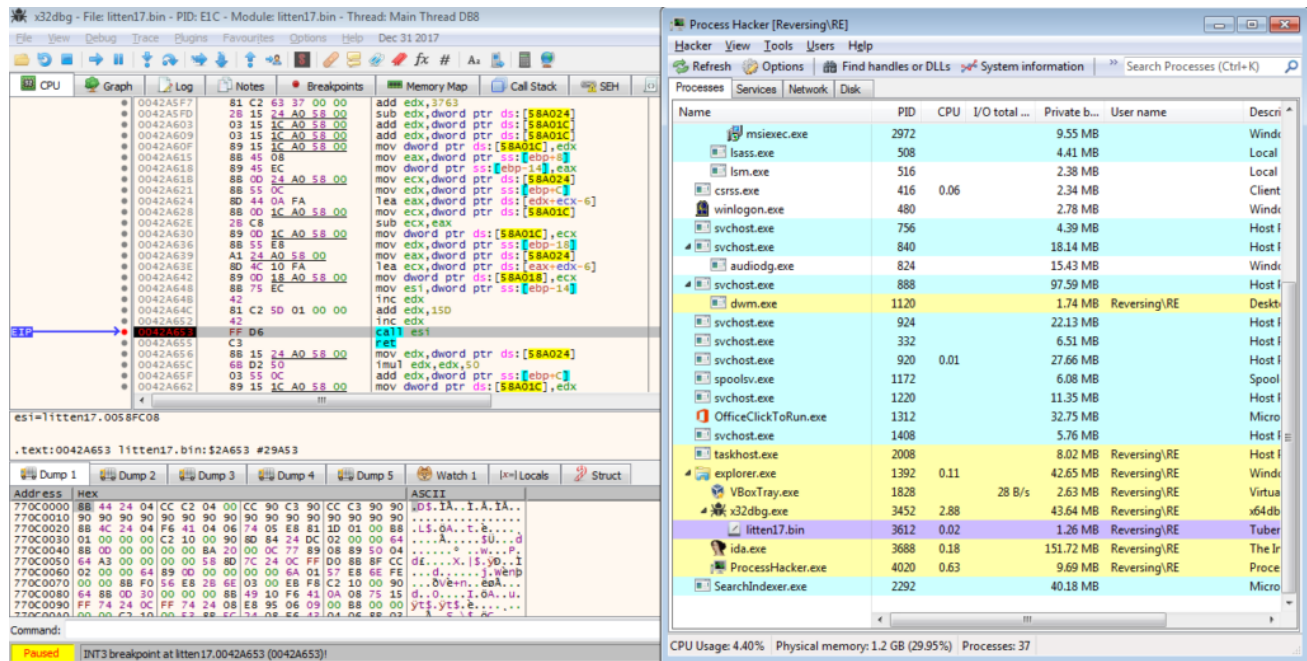


Inside this function, if we jump straight to the end, we can see a call to **ESI**. Hit space to jump from the Graph view back to Text view in order to get the address of this call. This could be the call to our unpacked EXE, and so we need the address of it to view it in x32dbg and put a breakpoint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

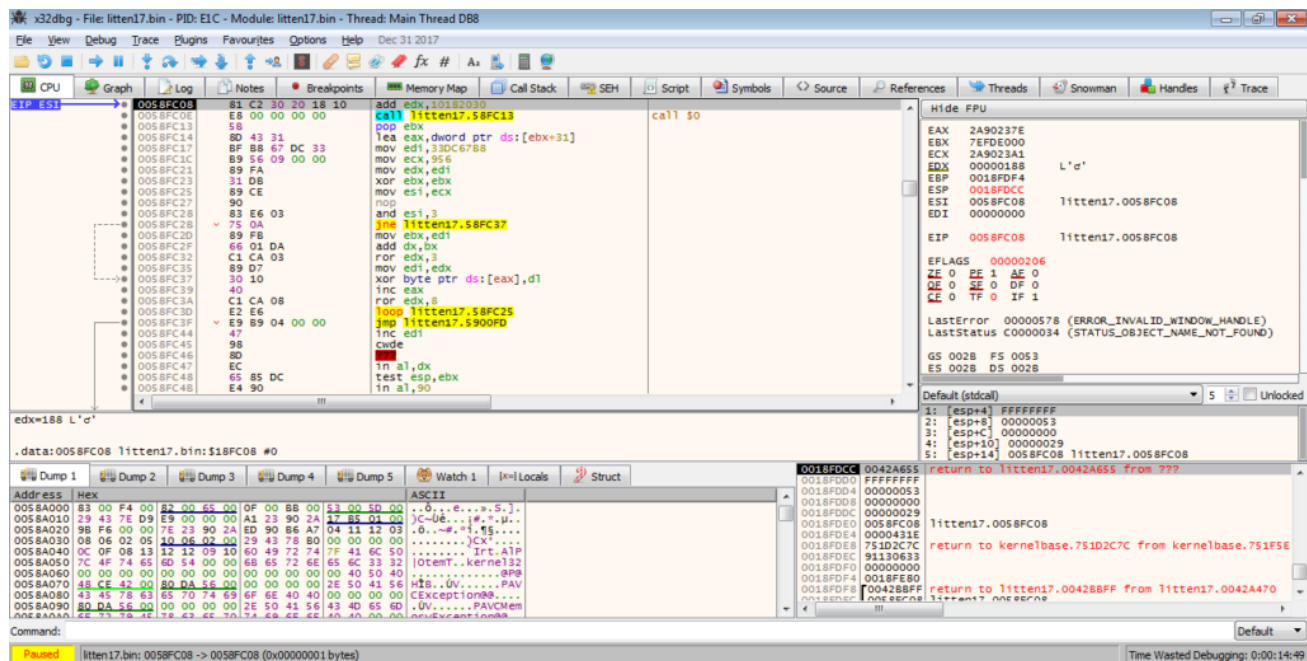


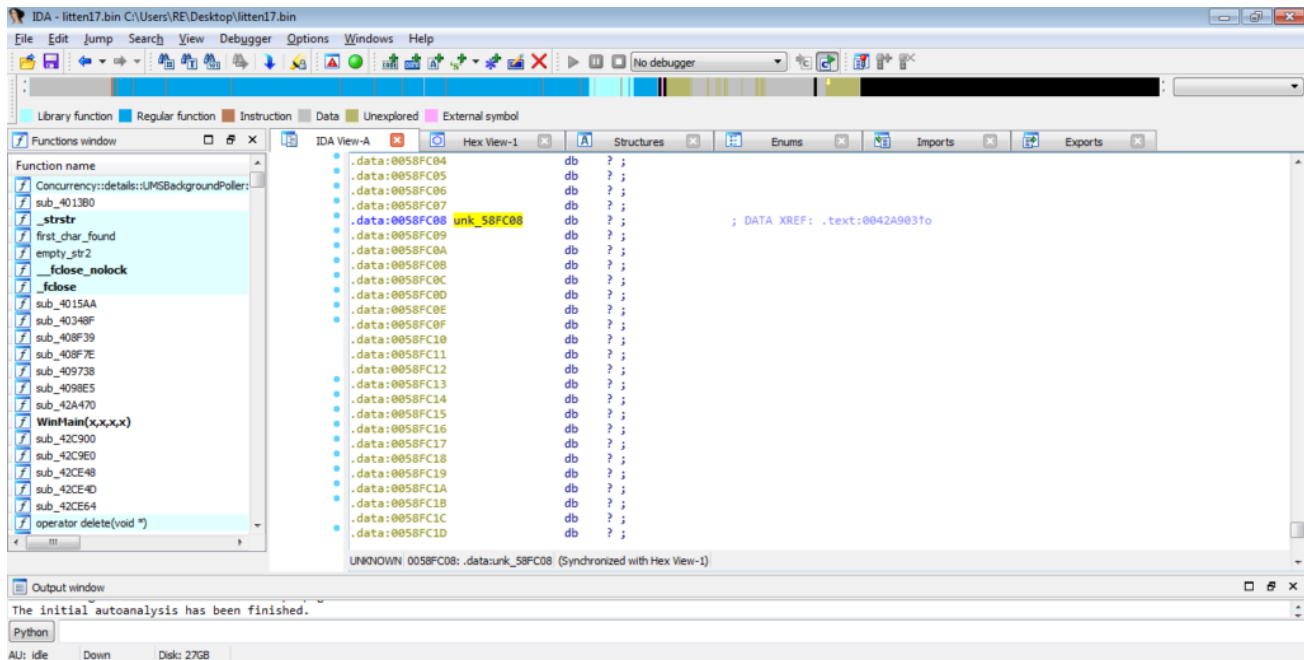
So the memory address to this call is **0x0042A653**, so let's now open this up in x32dbg and jump to this address. Simply push CTRL-G in x32dbg and type in **0042A653**, and then hit enter – this will jump to that address, allowing you to put a breakpoint on it. Whilst

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.



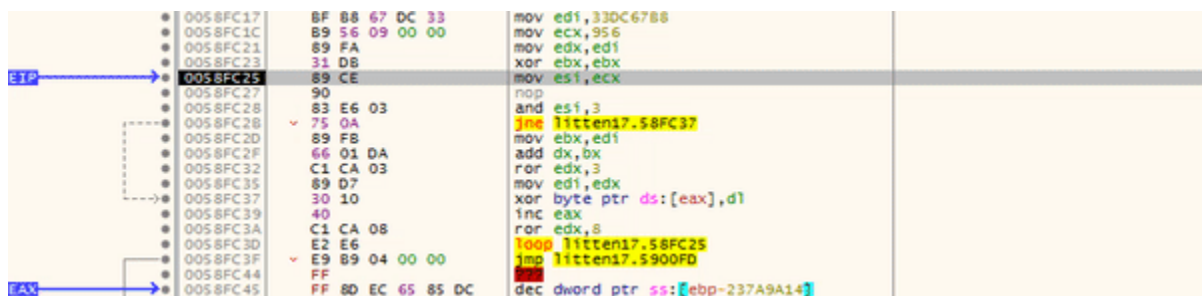
Once the breakpoint has been hit, step into **ESI** and in this sample there is what seems to be a second unpacking “stage” – this code was not here originally, in fact if we try and view it in IDA, you will simply see a lot of question marks and the variable **unk_58FC08**.



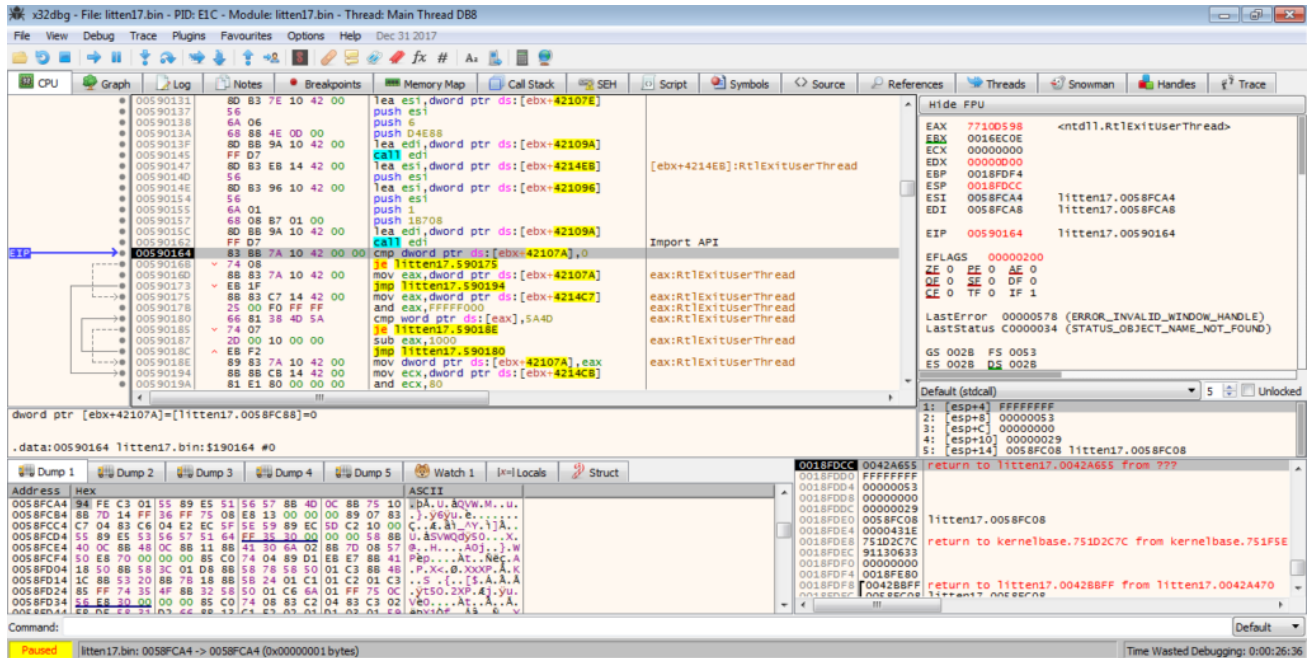


As it is difficult to statically analyse this section without dumping it and opening it up in IDA, I will be jumping over functions, rather than stepping through them manually. This will also help to speed things up, but make sure your network adapter is not attached, as one of these functions may execute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

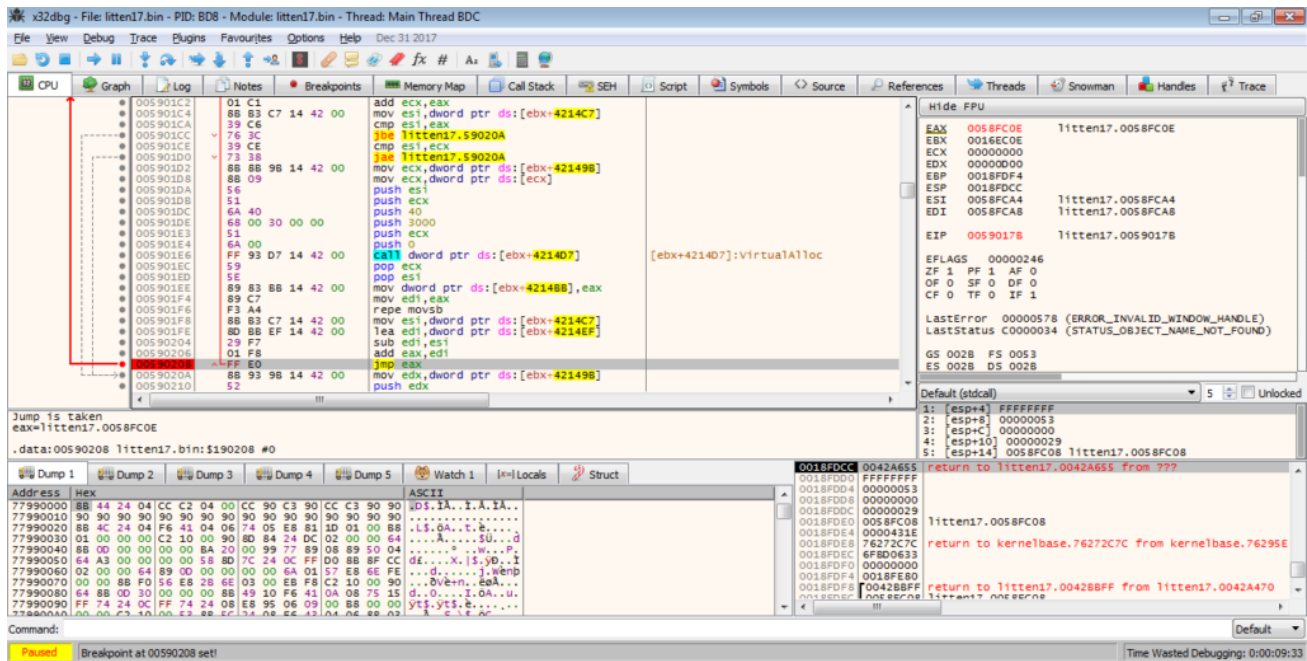
In the GIF below, you can see there is some form of loop going on, with XOR being used to XOR a byte at the memory address stored in EAX, with the value in DL. You might also notice that the assembly is changing as each loop goes on – this is another example of the second unpacking stage. We can simply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



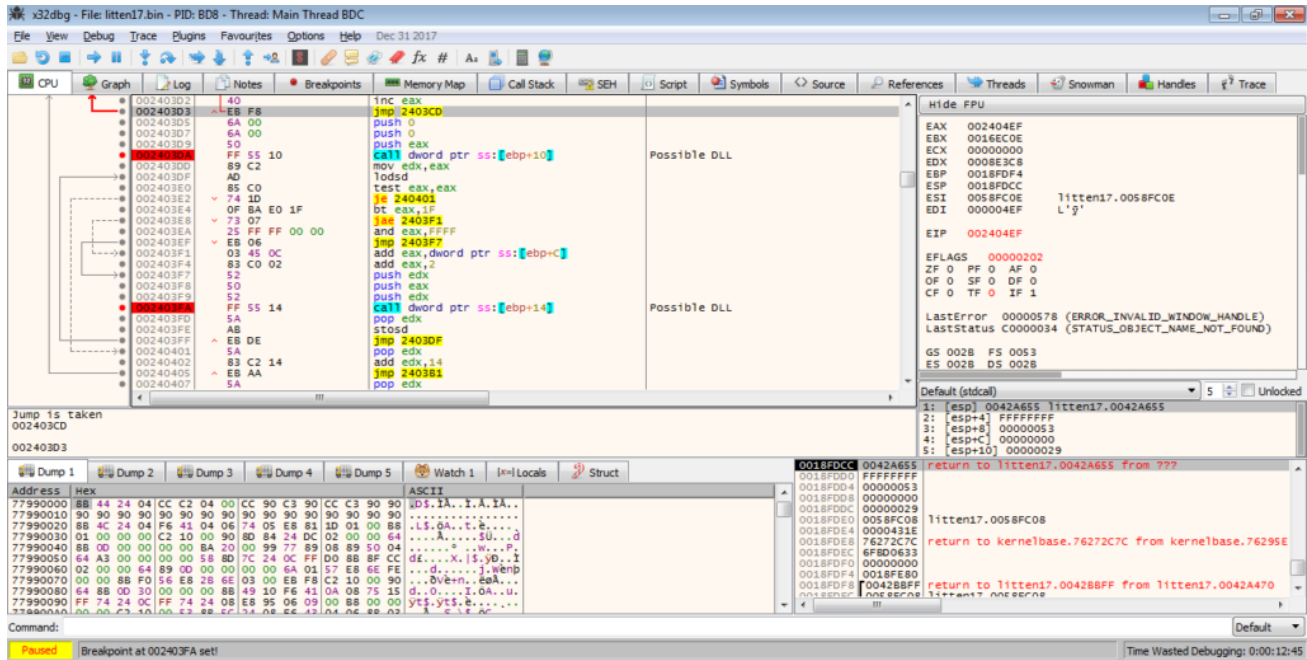
Once we follow the jump, we are met with several calls to **[ebx+xxxxxxx]**. Each of these could jump to the unpacked EXE, however as we progress further on, it is clear that these are simply calls to Windows API functions. Notice the call to **EDI**? EDI is pointing to a function that dynamically imports 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**.



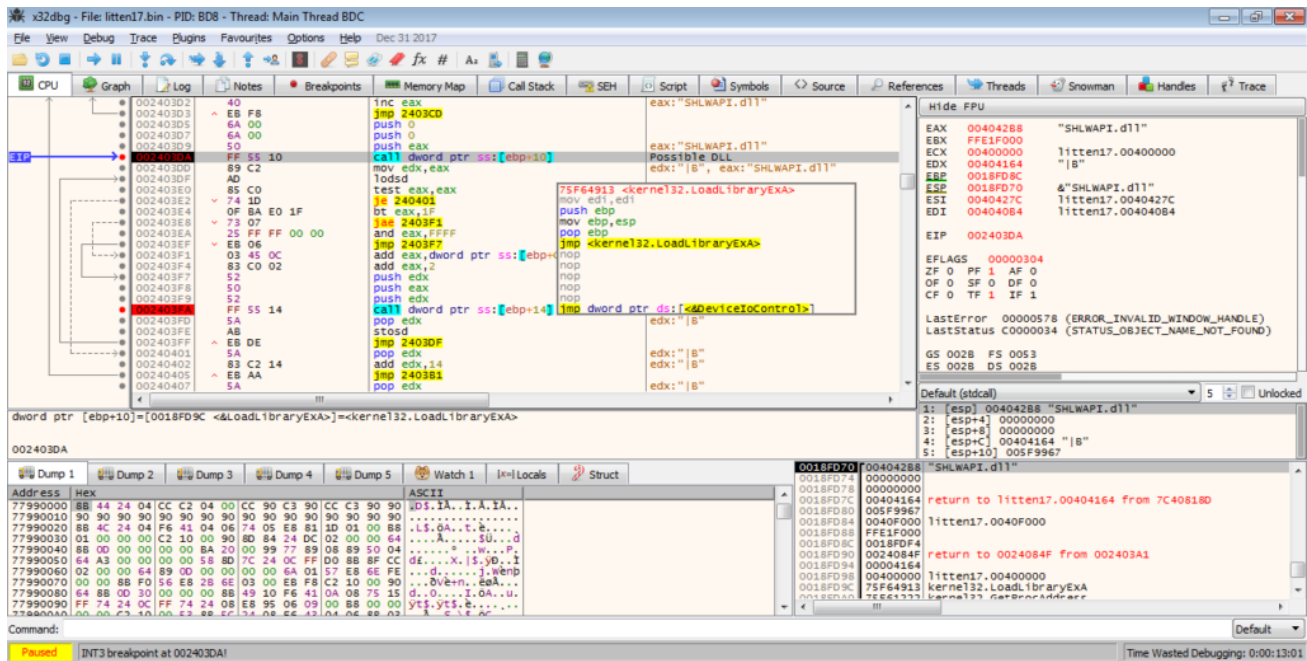
Scrolling down a bit, we can see a **jmp eax**, so let's put a breakpoint on that and run to it, and then follow the jump.

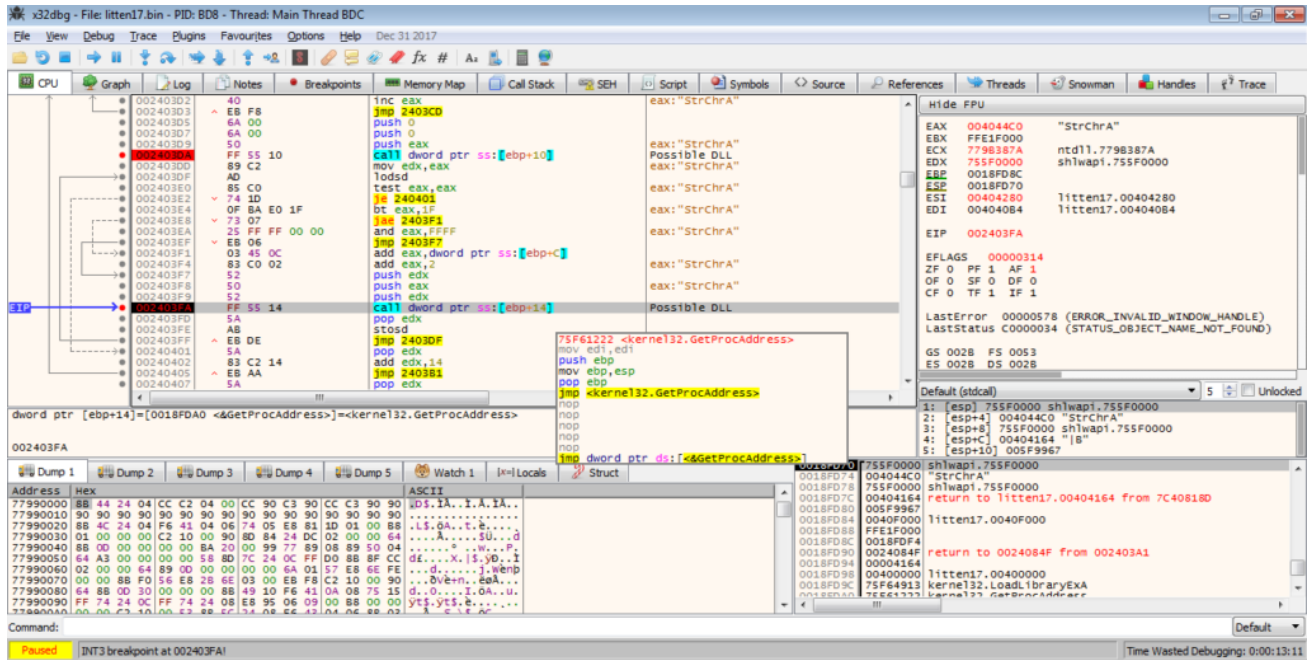


This jump takes us to a newly allocated region of memory, with more code. I didn't want to examine each function, so I scrolled down until the **ret** instruction, and started examining the **(local - not API calls)** functions backwards. The last function did not seem likely to be the executable "executor", 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.

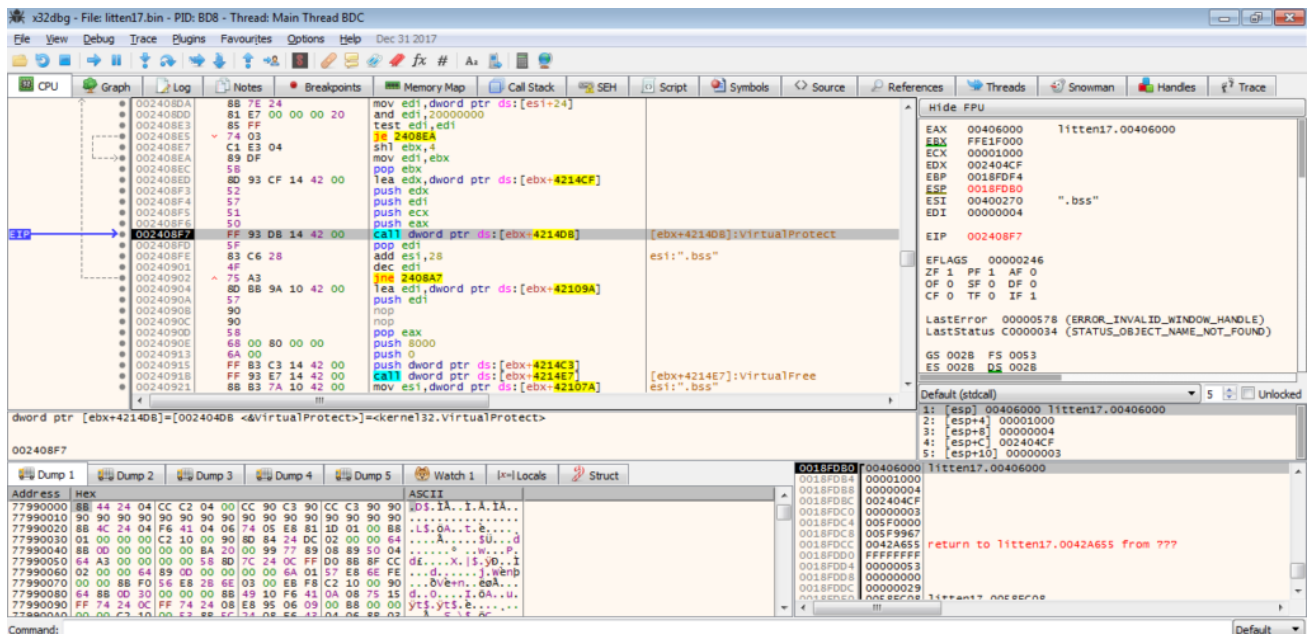


However, upon executing the program and hitting the breakpoint, it is clear that they are simply calling **LoadLibrary** and **GetProcAddress**.

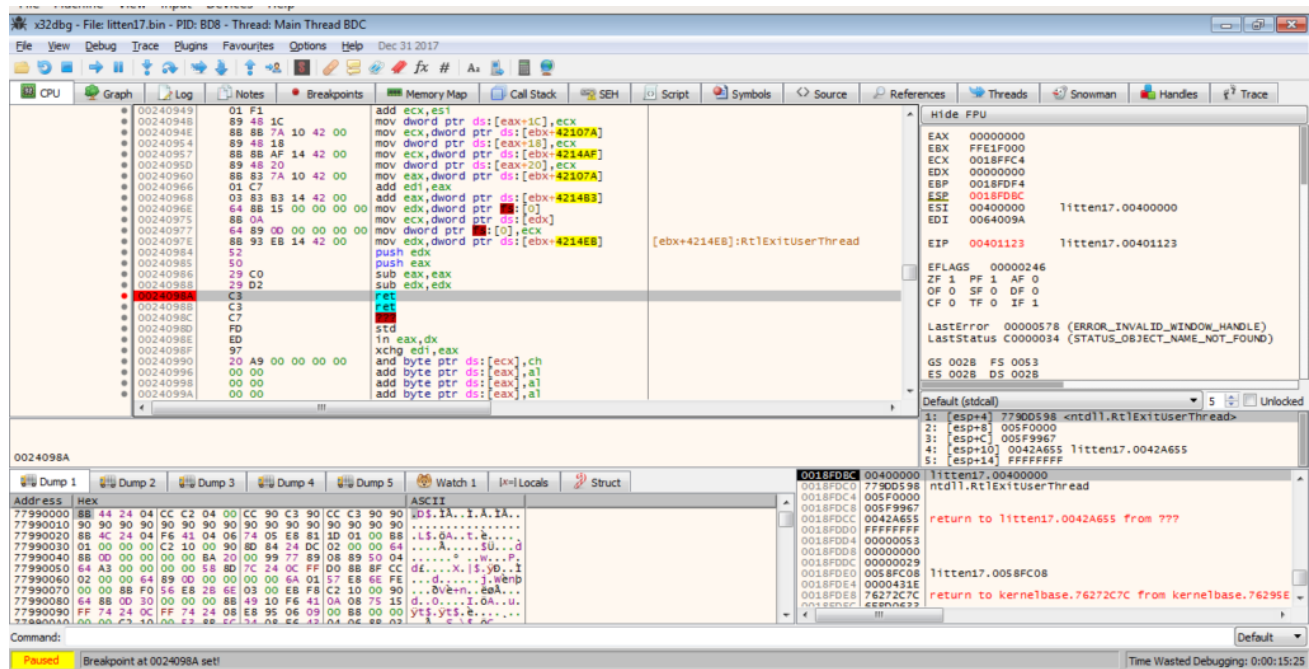




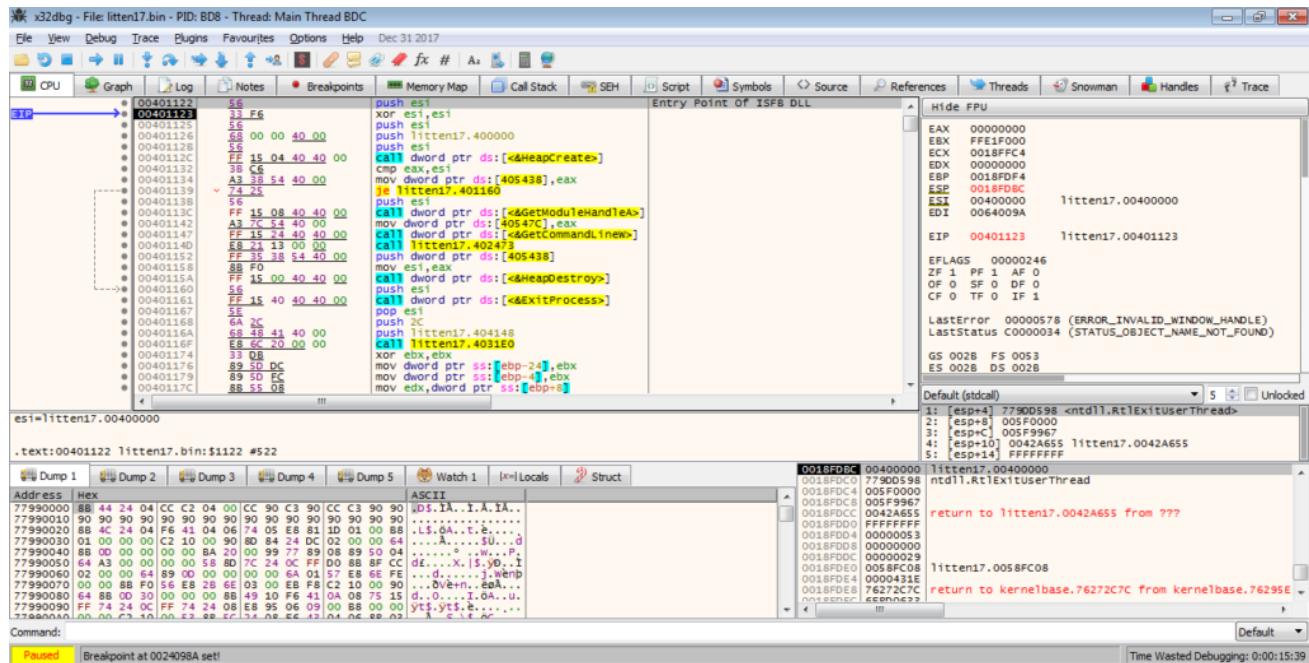
Therefore, we can simply return from this function, as it is just importing API's, which isn't extremely important to us yet. After exiting the function, there are several calls to API's, in particular **VirtualProtect**. **VirtualProtect** is responsible for changing the permissions/protection of different 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.



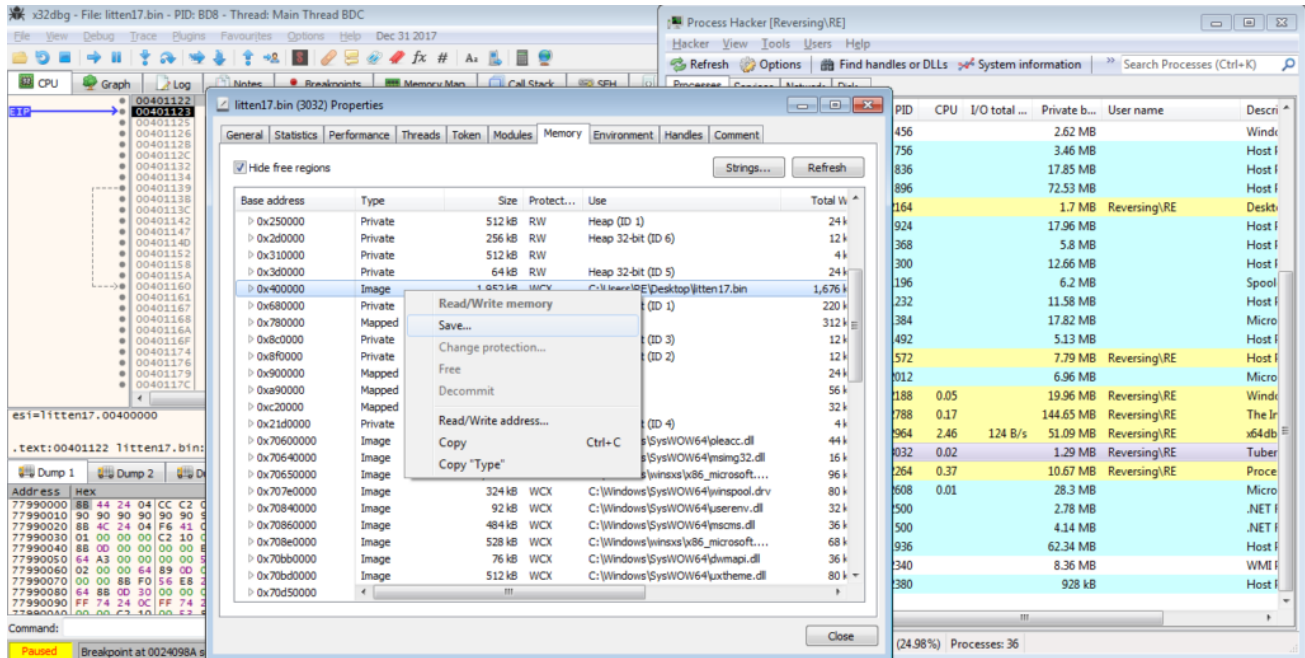
At this point, there isn't much going on in the function, so we can simply put a breakpoint on the **ret** instruction and return from this function.



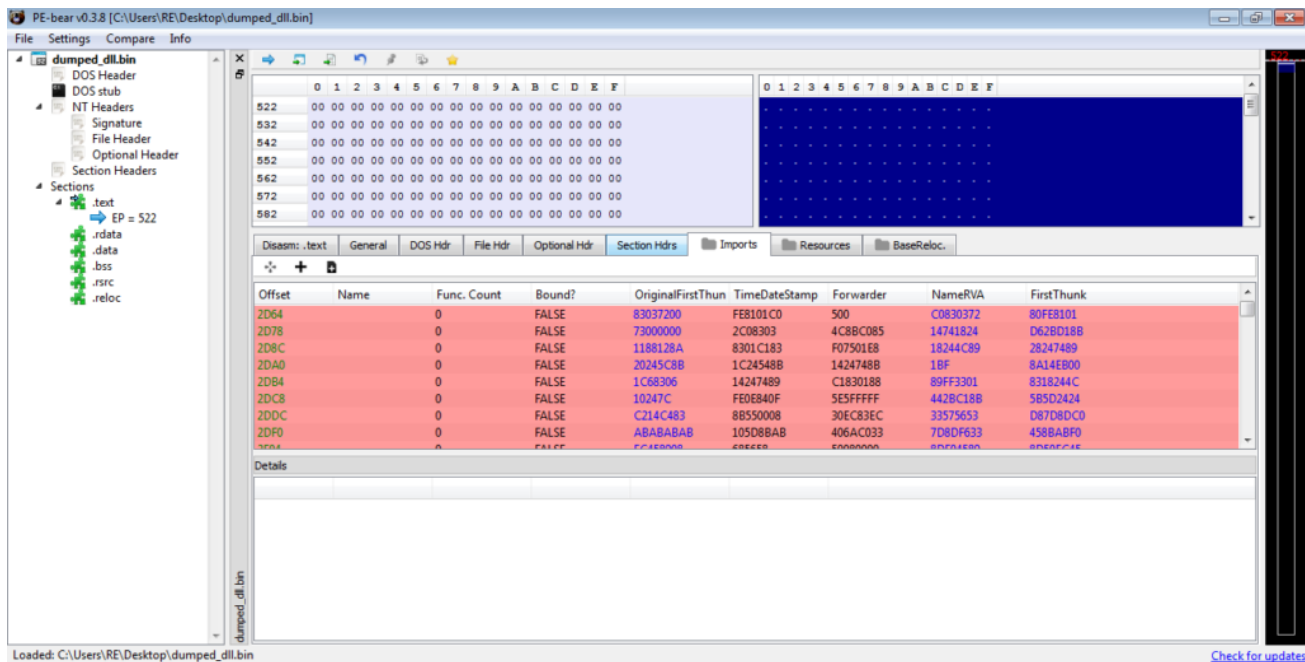
Upon exiting the function, you'll notice the memory address is in the **0x00401000** region. This is the unpacked program! Now we can dump it out, so make sure you have Process Hacker open, although you can dump it from x32dbg.



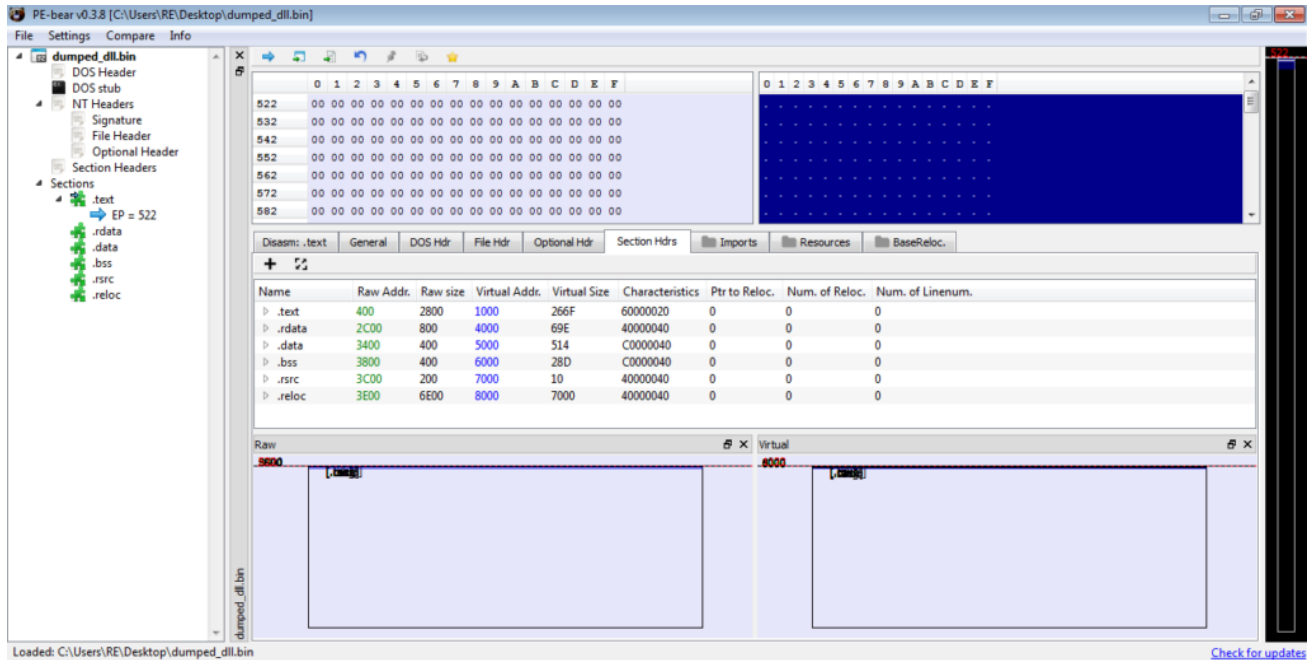
To dump it from memory, simply double click the process in Process Hacker, and right click the **0x400000** region of memory, and select **Save**.



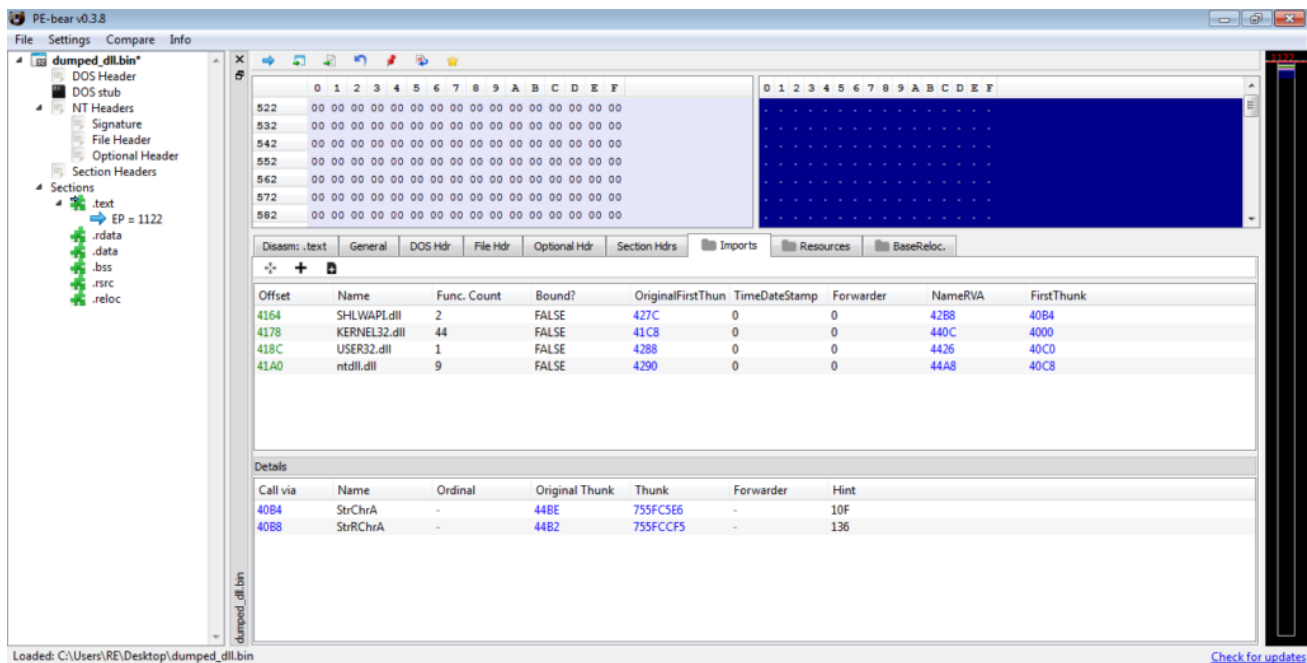
Now we can open it up in PEBear and unmap the dumped file. Upon doing so, you can see that the imports have not been successfully resolved by PEBear. This is why we need to unmap the file. When a program is about to be executed, it needs to be mapped in memory, so that it can be interpreted correctly.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



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!

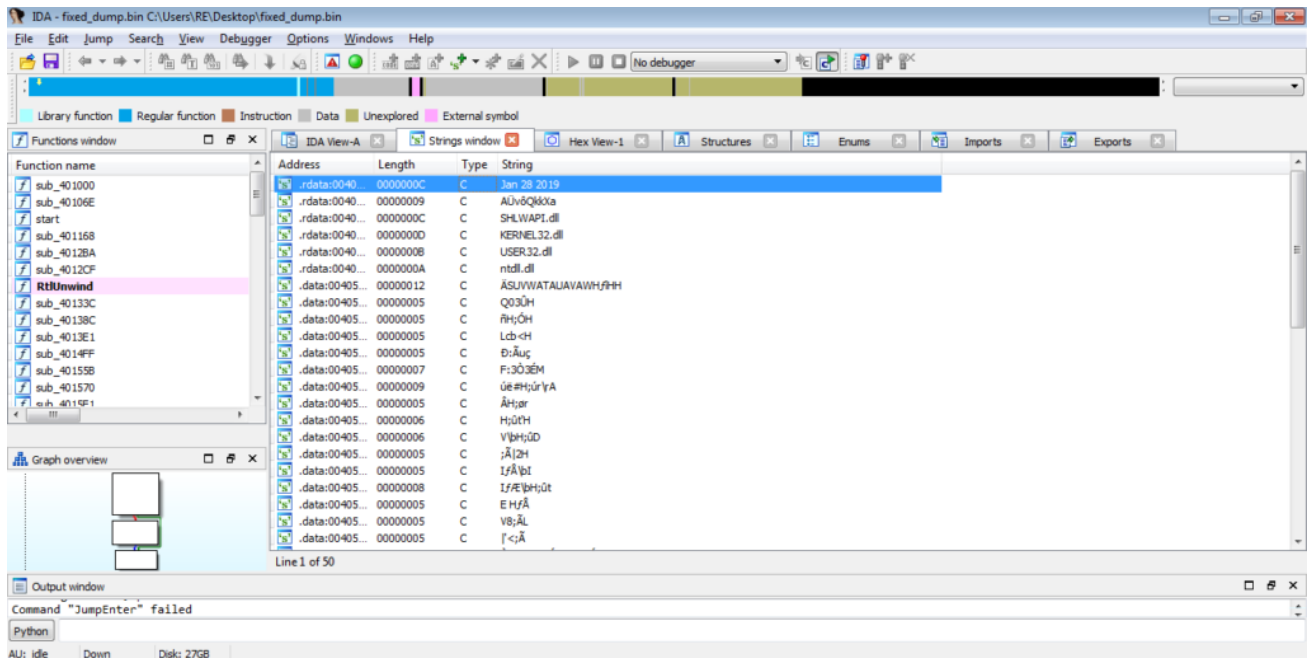
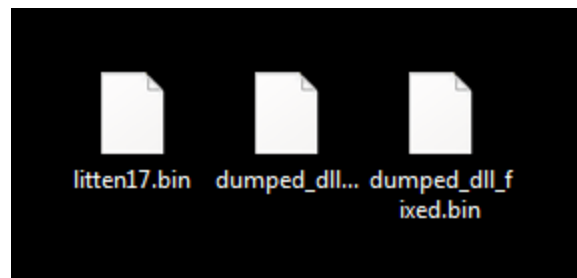


Part 2: Analysing the Dumped Executable

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.



Strings

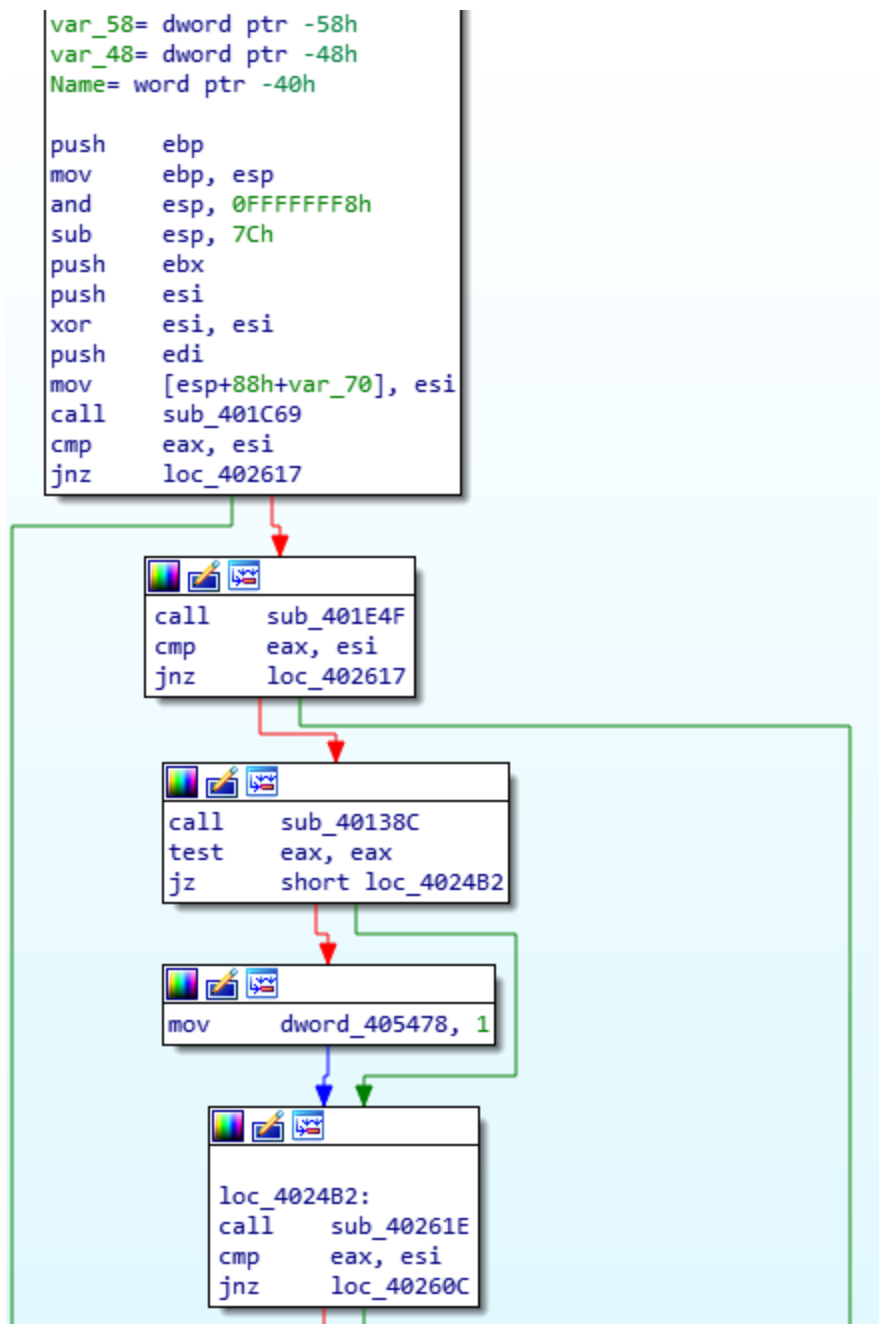
There are several functions called inside this function, so let's take it section by section, so first let's take a look at the first four functions. From the image below, it is clear that the return value of **sub_401C69** (stored in **eax**) needs to match the value in **esi**, otherwise it will jump 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 act in the same way as the first and second function, in the sense that the returned value is compared 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.

```
; Attributes: noreturn

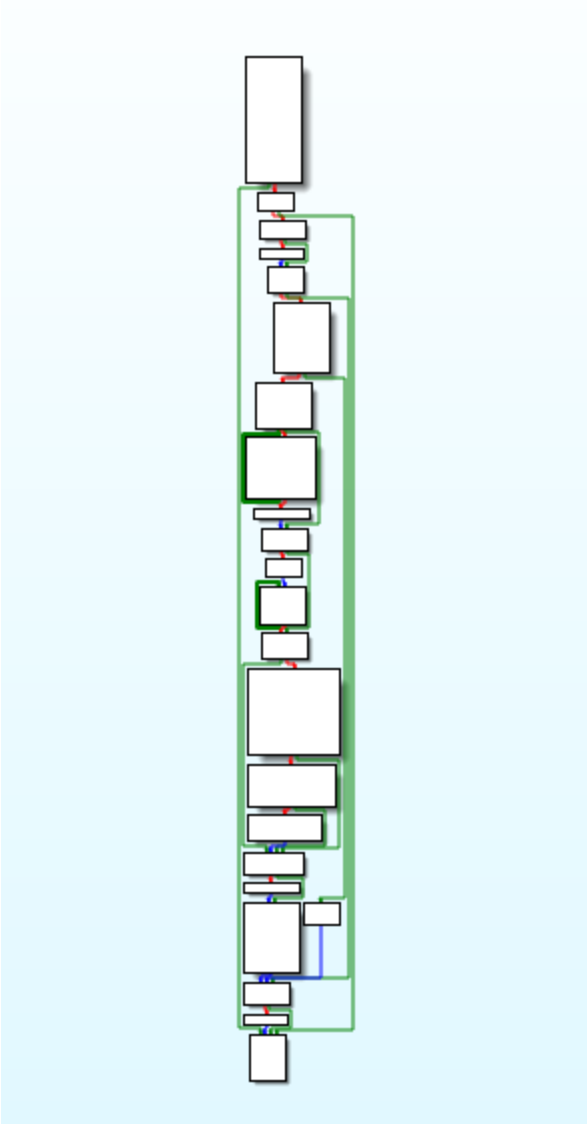
public start
start proc near
push    esi
xor     esi, esi
push    esi           ; dwMaximumSize
push    400000h      ; dwInitialSize
push    esi           ; flOptions
call    ds:HeapCreate
cmp     eax, esi
mov     hHeap, eax
jz     short loc_401160
```

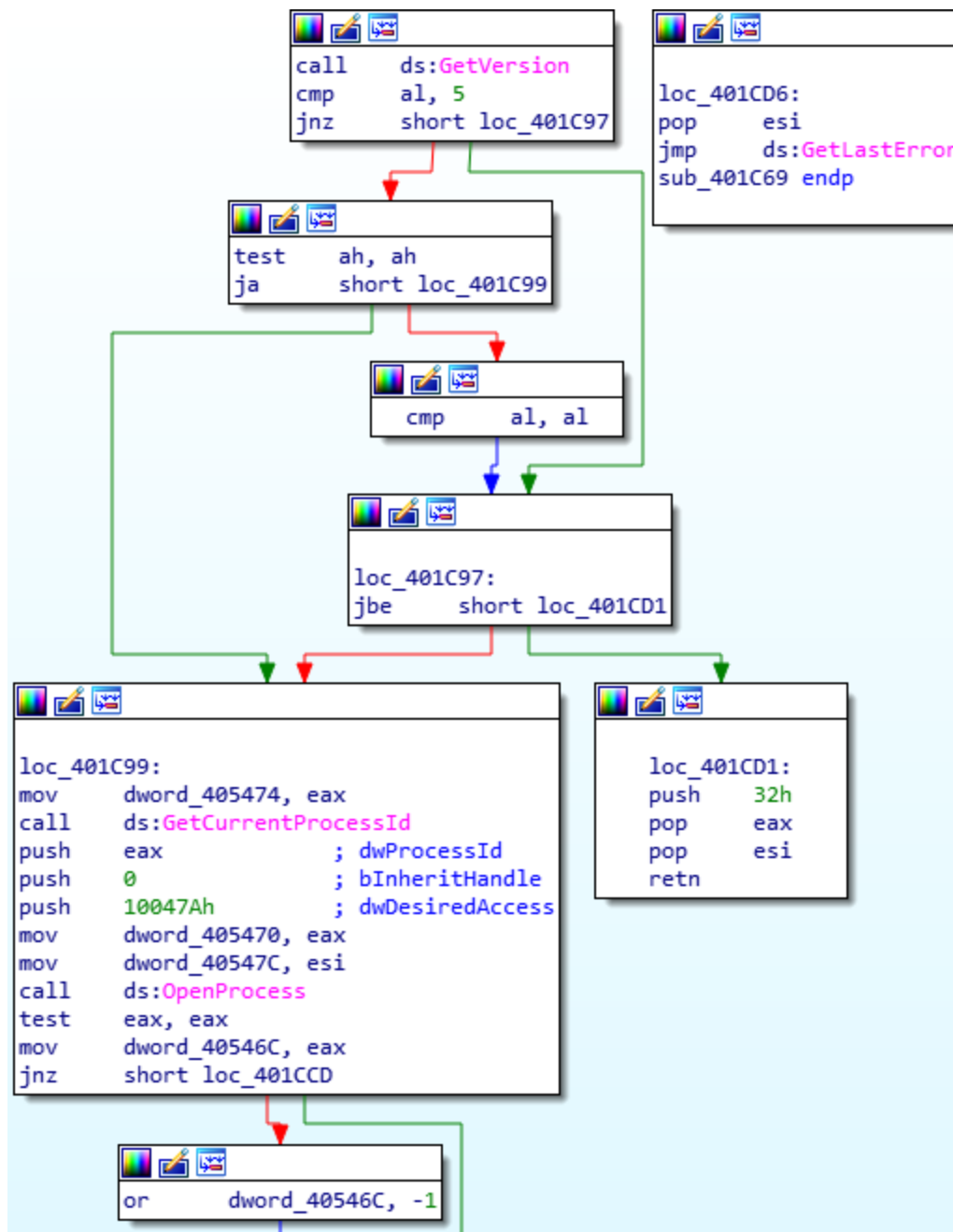
```
push    esi           ; lpModuleName
call    ds:GetModuleHandleA
mov     dword_40547C, eax
call    ds:GetCommandLineW
call    sub_402473
push    hHeap         ; hHeap
mov     esi, eax
call    ds:HeapDestroy
```

```
loc_401160:           ; uExitCode
push    esi
call    ds:ExitProcess
start endp
```

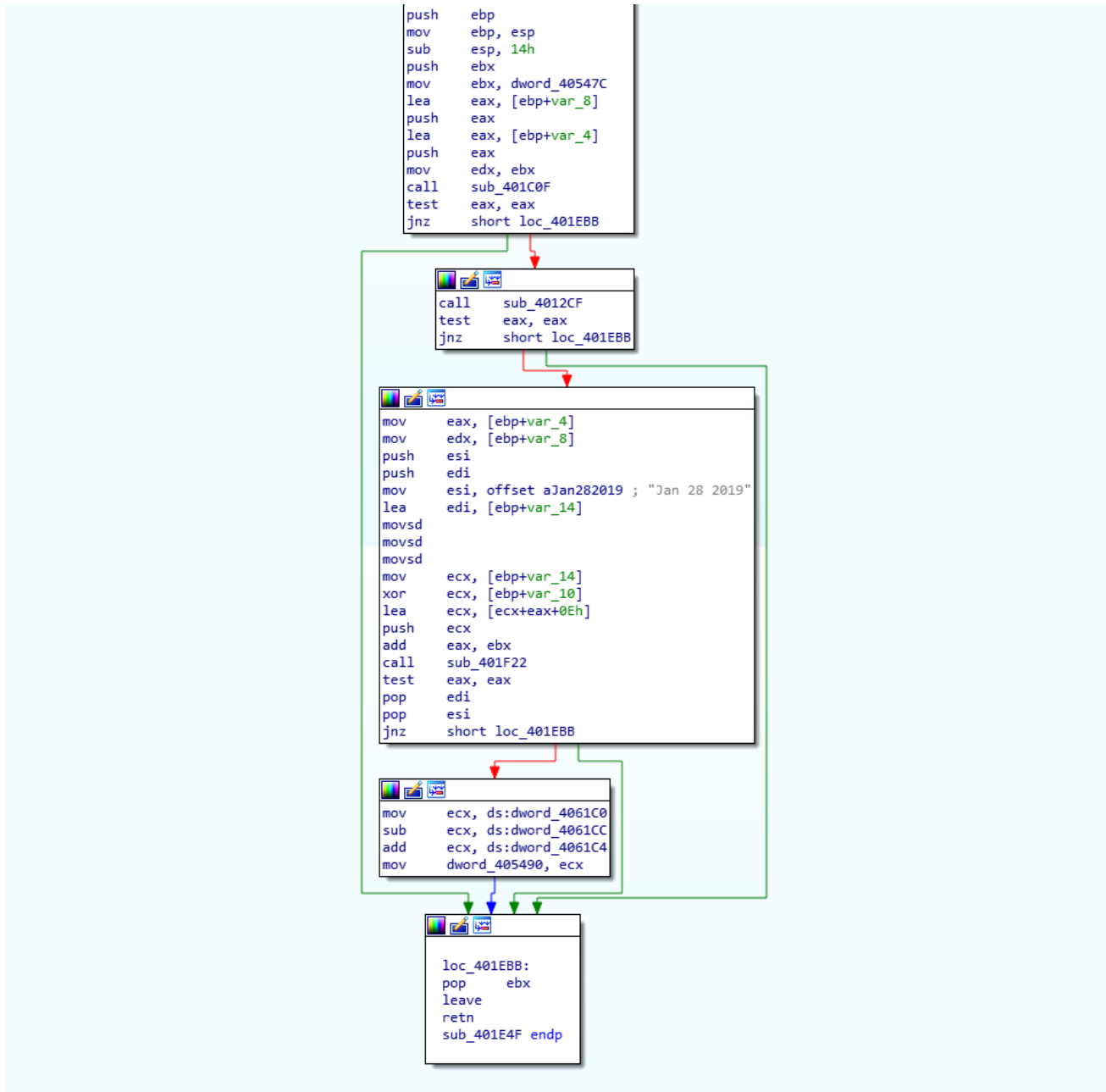


Taking a look at the first function (the main bit), we can see that the malware is opening its own process and storing the handle in a DWORD, which is set to **-1** if the malware failed to open the process. This then returns back to the main function, where the returned result is compared against the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



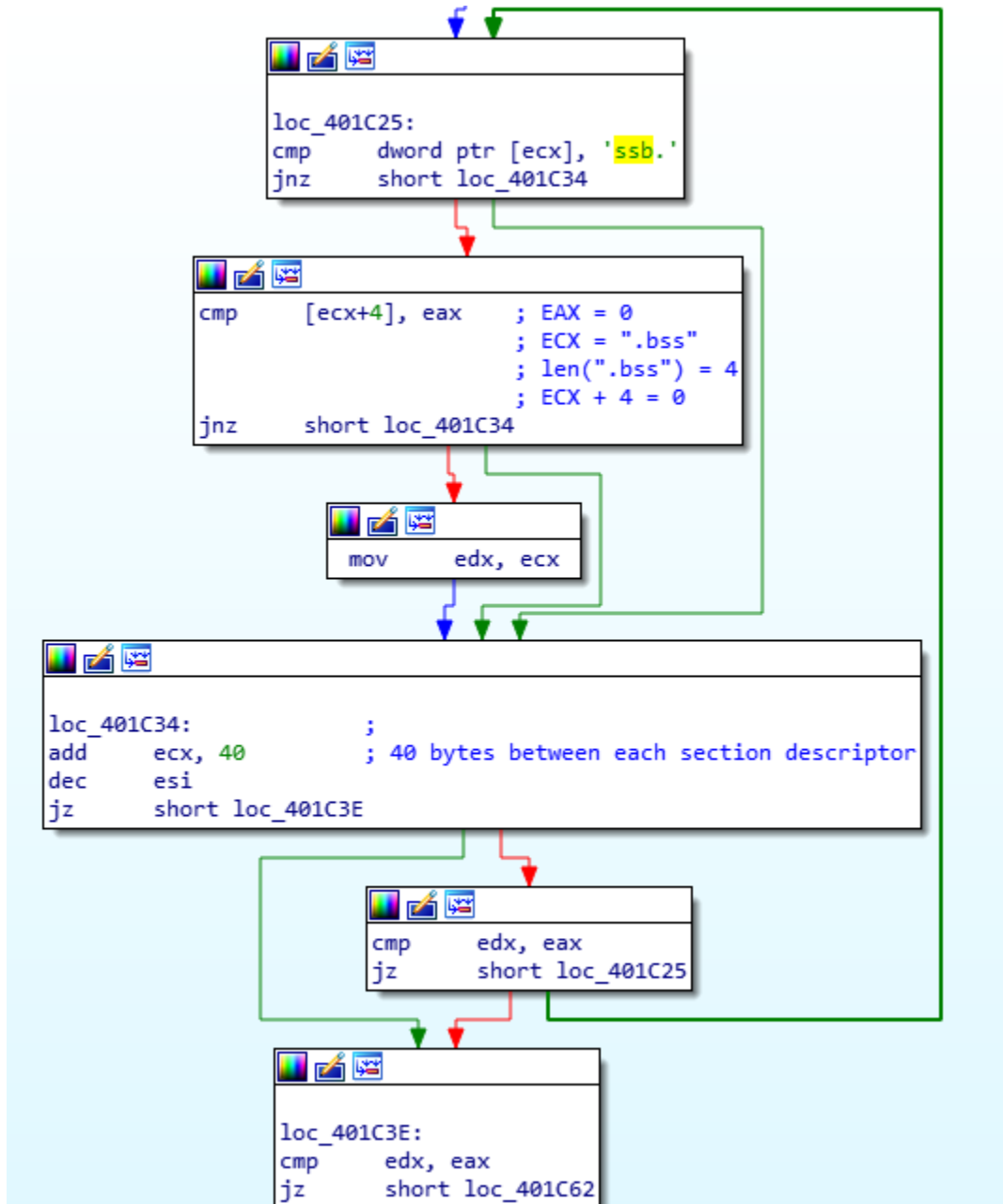


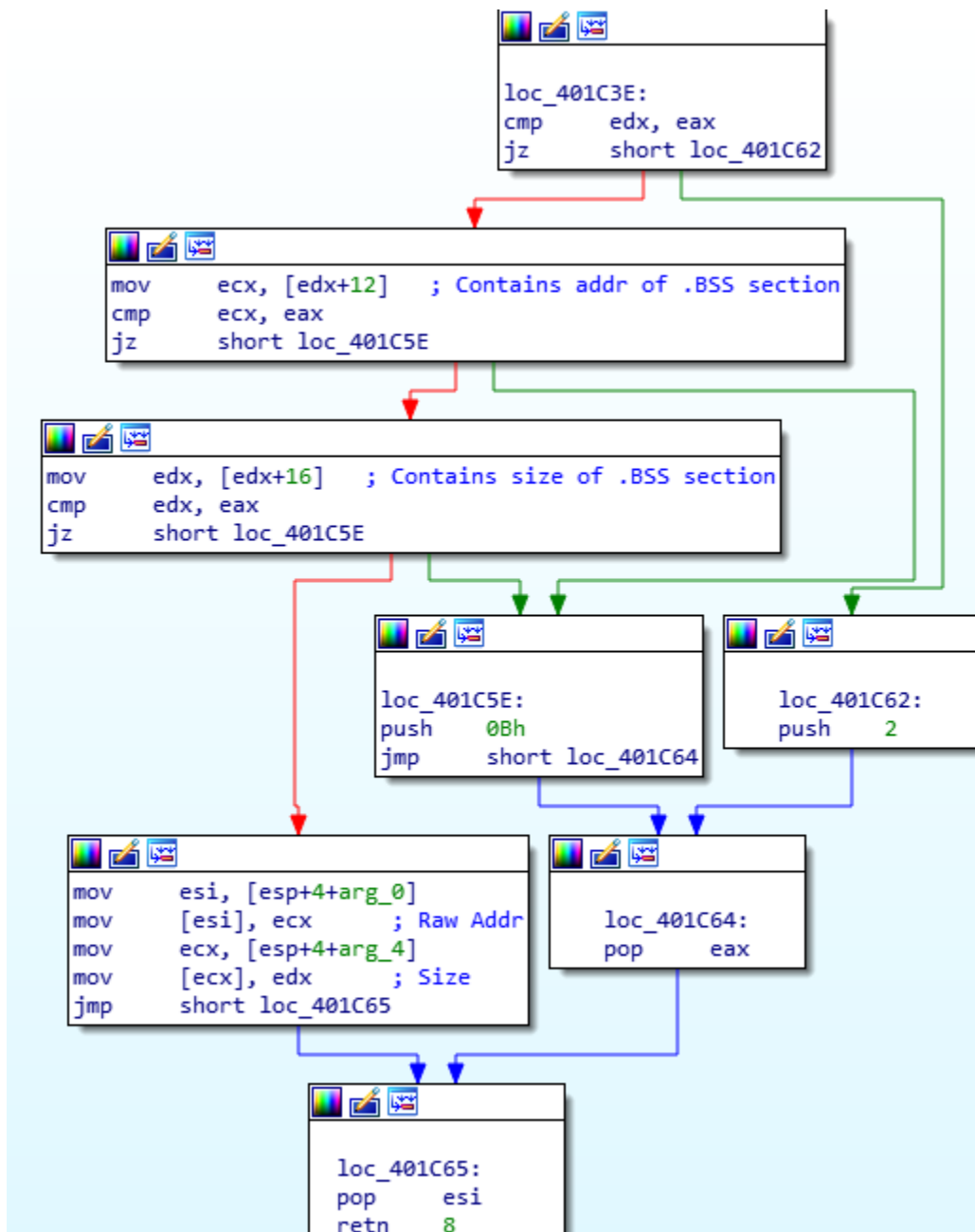
The second function is where things start to get interesting. In the image below, you'll notice the compile date being moved into **ESI**, which, as I mentioned before, is used for string decryption. You might already know this, but for those that don't, ISFB contains a **.BSS** section, which contains 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.



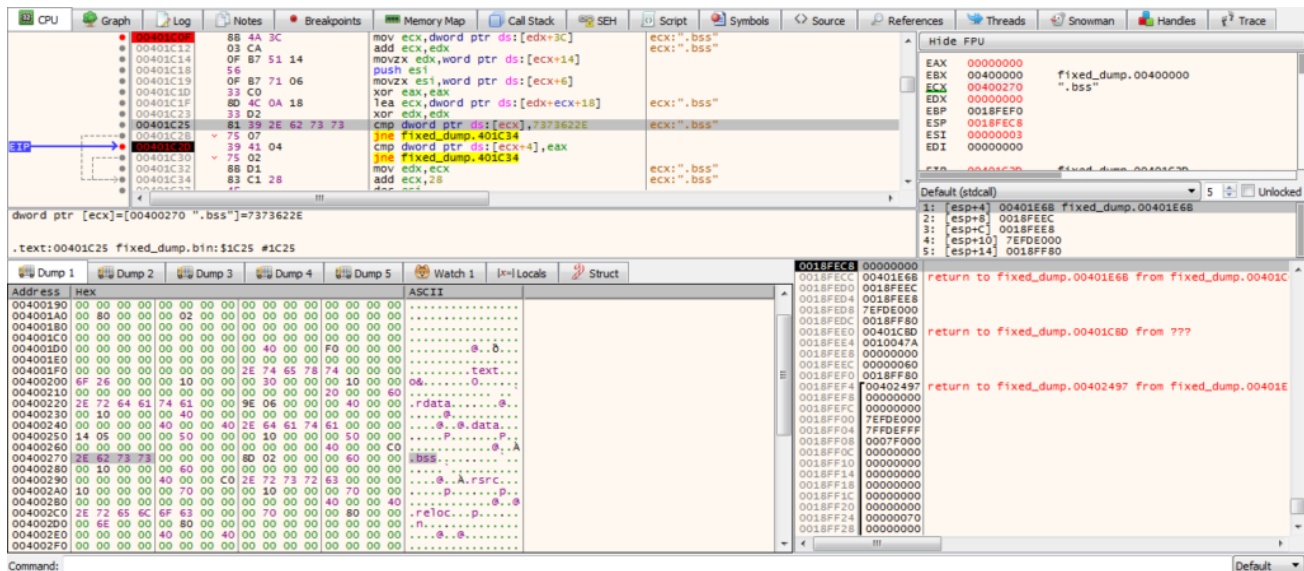
Looking at the first function, you'll notice **ssb**. (endian issues, as it is stored as hex – actually is **.bss**) being compared to the value at **[ECX]**, inside a loop. In order to decrypt the strings in the **.BSS** section, ISFB must first locate the **.BSS** section, and in order to do so, it 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 to 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 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



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 let's 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 let's move into that function.

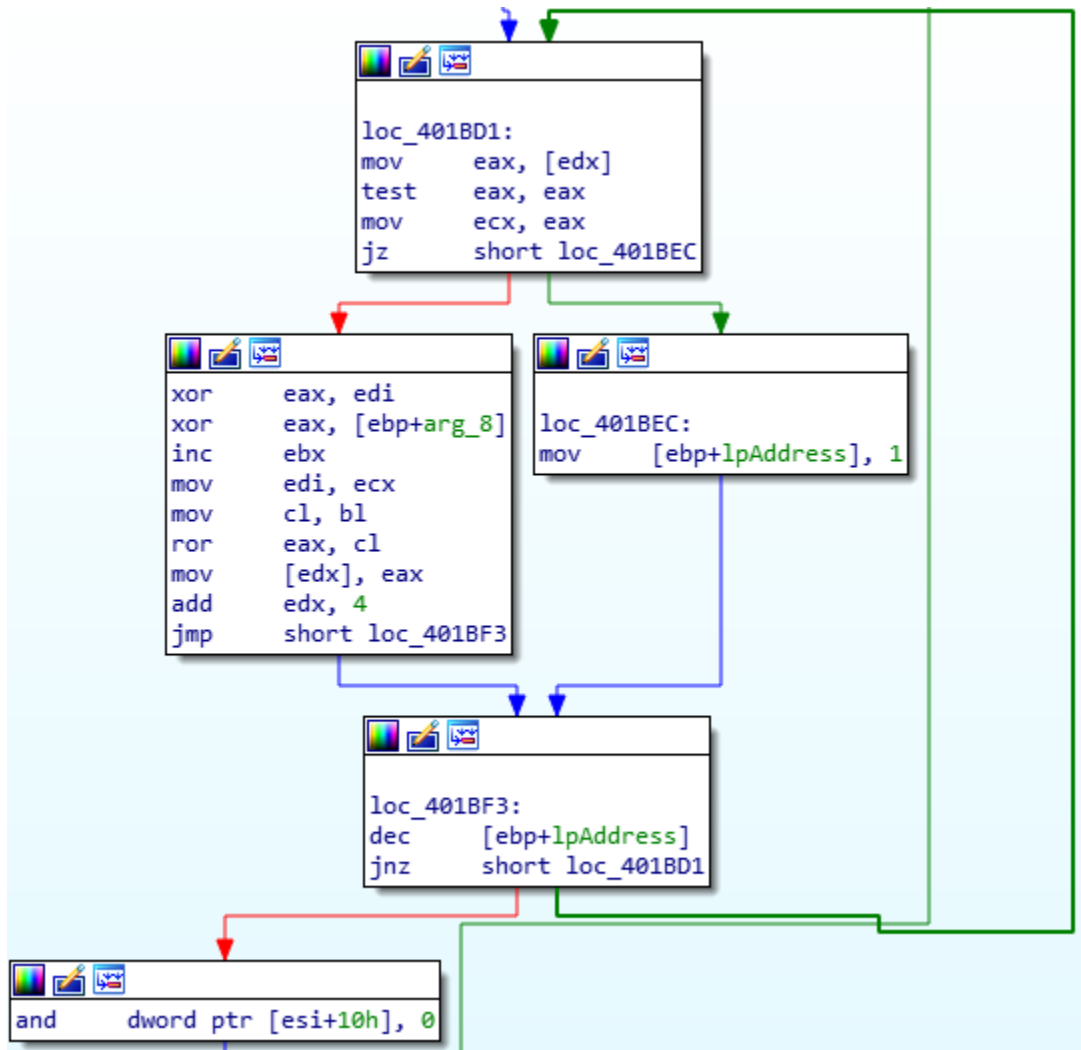
```

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

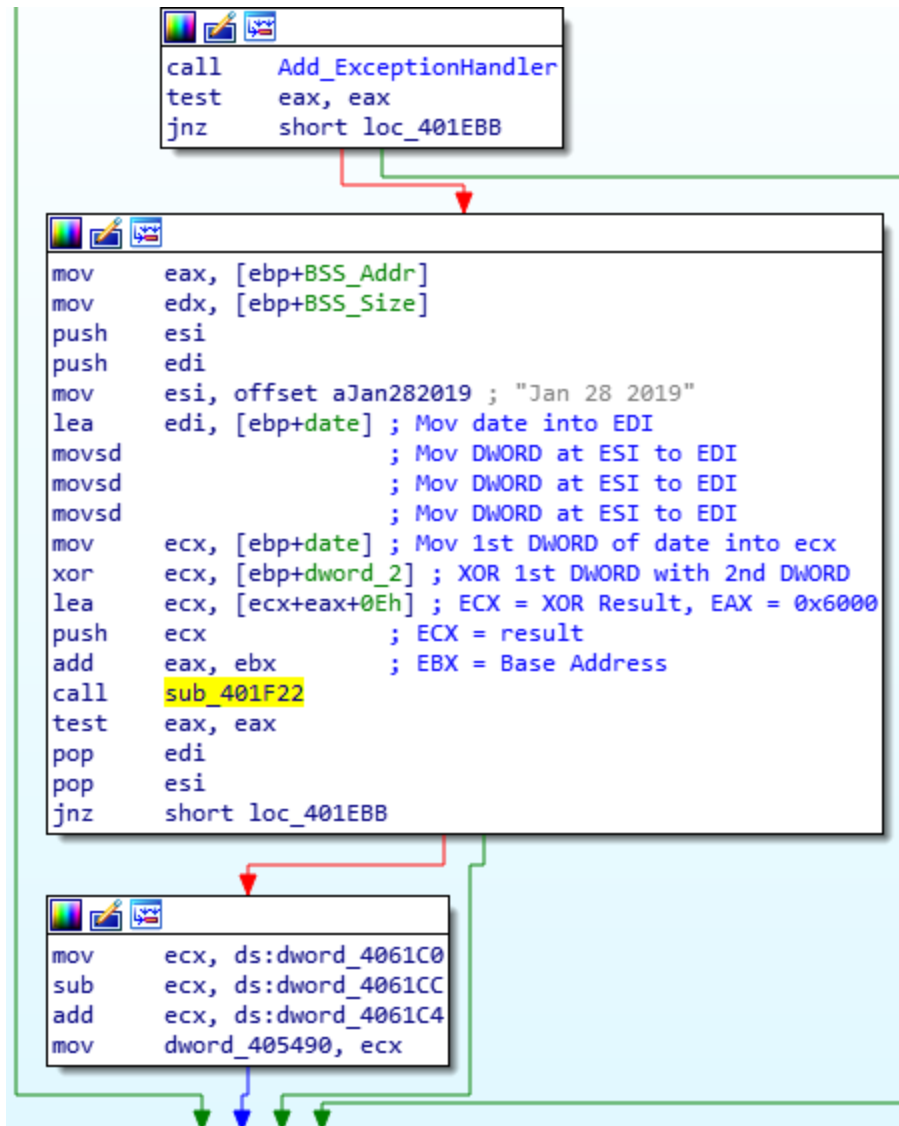
    v1 = 0;
    v2 = lpCriticalSection;
    if ( lpCriticalSection )
    {
        v3 = *a1;
        v4 = **a1;
        if ( v4 == 0xC0000005 )
        {
            v5 = (void *)v3[6];
            if ( !sub_401B36(lpCriticalSection, (LPVOID)v3[6], 1) )
            {
                TlsSetValue((DWORD)v2[1].OwningThread, v5);
                a1[1][48] |= 0x100u;
                return -1;
            }
        }
        else if ( v4 == 0x80000004 )
        {
            v6 = (DWORD *)&lpCriticalSection[1].OwningThread;
            v7 = TlsGetValue((DWORD)lpCriticalSection[1].OwningThread);
            if ( v7 )
            {
                if ( !sub_401B36(v2, v7, 0) )
                {
                    TlsSetValue(*v6, 0);
                    return -1;
                }
            }
        }
    }
    return v1;
}

```

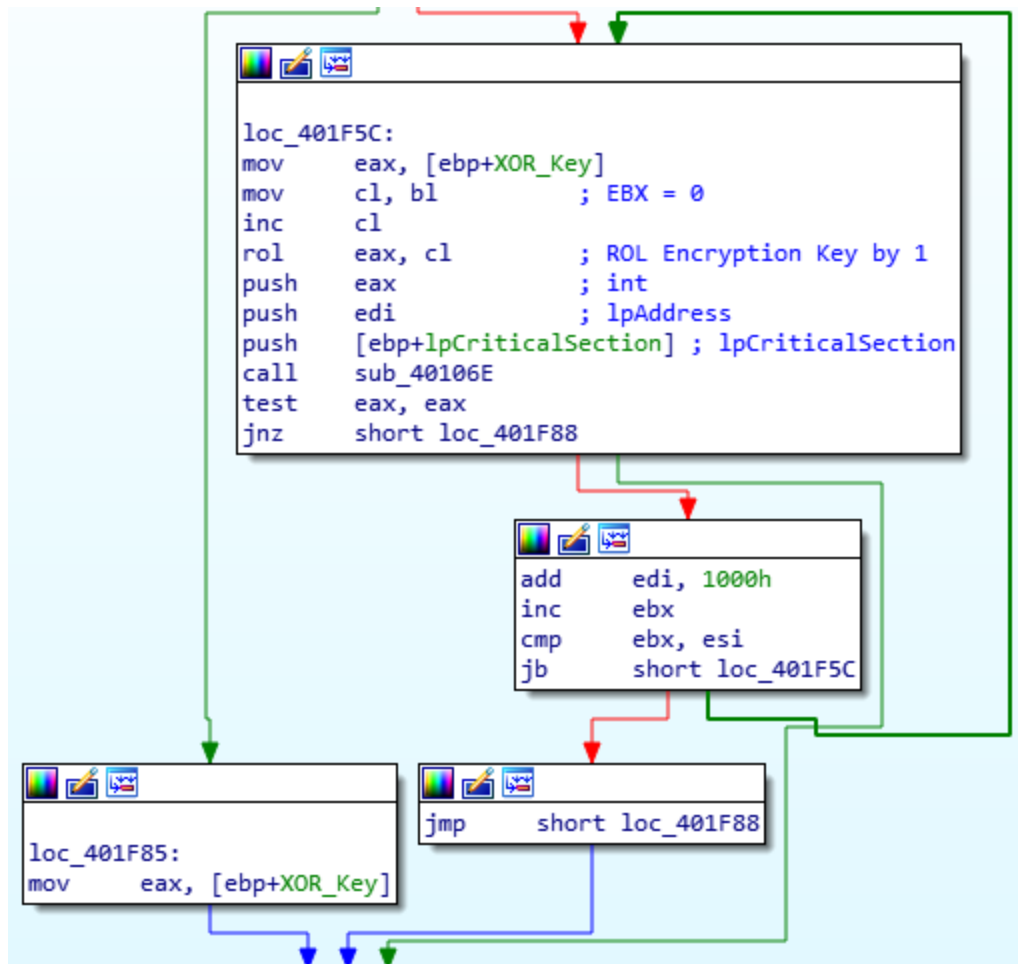
The important part of this function happens soon after it is executed. In the image below, you'll be able to make out a loop, as well as two XOR instructions and a ROR instruction. This is the BSS section decryption – the XOR key that is created later on is actually **arg_8**, so take note of that. So, 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, that's where we need to look. First, the address of the BSS section (**0x6000**) is moved into EAX, and the size of it (**0x1000**) 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 (**0x6000** here) and the value **0xE**. This is then pushed as the first argument for the next function that will be called.



Taking a look at the function, **arg_0** (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.



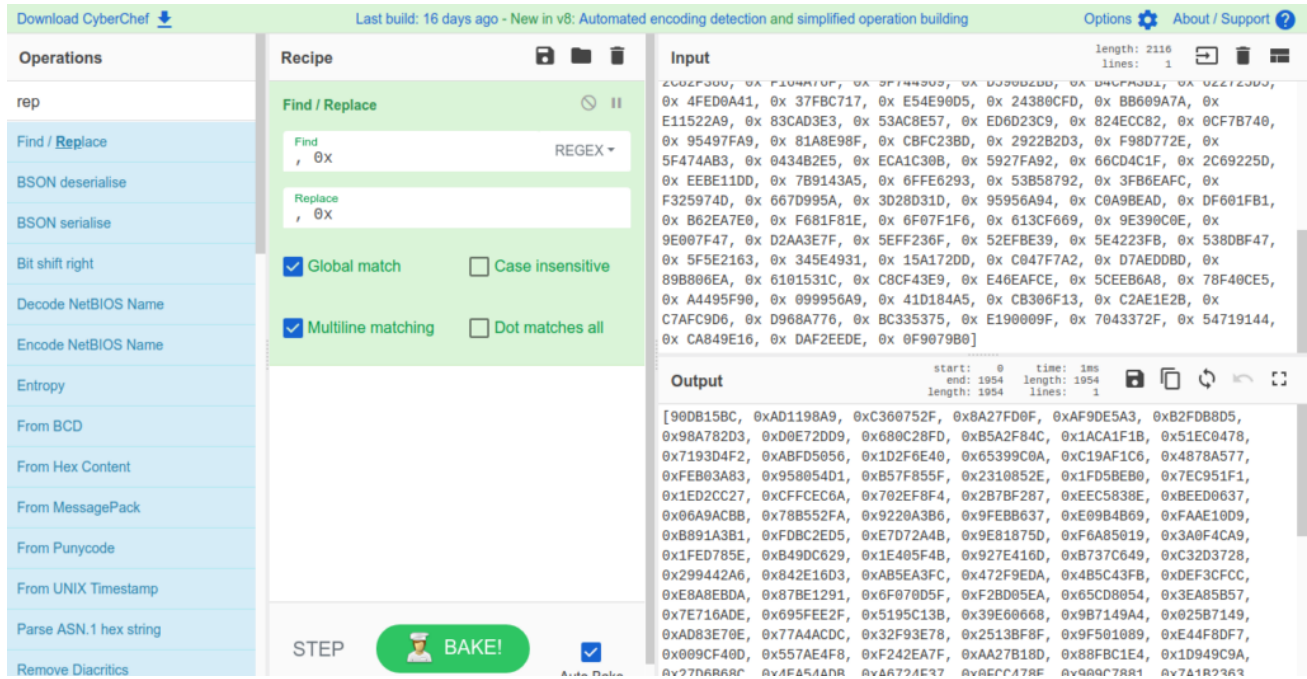
Once we have performed these calculations, we get this as the key: **0x249d730c**. Whilst it is possible to get the strings from a debugger and copy it over to an IDA instance, I prefer to replicate custom routines using Python. I'm currently polishing up the decryption script I have been using, and 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.


```

root@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 = ""90DB15BCAD1198A9C360752F8A27FD0FAF95E5A3B2FD88D598A782D3D0E72D09680C28F0B5A2F84C1ACA1F1B51EC04787193D4F2ABFD50561D2F6E4065399C0AC1
9AF1C6408A577FE0B3A83958054D1B57F855F2310852E1FD5BE807EC951F11ED2CC27CFCEC6A702EF8F42B7F287E5C838EBE063706A9ACBB78B552FA9220A3B69FEB8637E0
9B4B69FAAE10D9B891A3B1FD8C2ED5E7D72A48E081875DF6A850193A0F4CA91FED75EB49DC6291E405F48927E4160B737C649C32D3728299442A6842E1603AB5EA3FC472F9EDA4B
5C43F8DF3CFCE8A8EBD0A87B12916F0705FF28D05EA65CD08543E8A58577E7160DE695FEE2F5195C13B39E60689B7149A4025B7149AD083E70E77A4ACDC32F937825138F8F9F
561089E44F8D7009CF40D557AE4F8F242EA7FAA27B18D088FBC1E41D949CA27D6868C4E54AD8A6724F370CC478E909CF8817A1B236388443369AF6A711594194F06EA33FC82F
82F386F164A76F9F744969D590B2BB84CFA3B1622725D54FED0A4137FBC717E54E90D524380CFDBB609A7AE1152A983CAD3E353AC8E57ED62C39824ECC820CF7B74095497FA981
A8E98FCBFC23BD2922B2D3F980772E5F474AB30434B2E5ECA1C30B5927FA9266CD41F2C69225D0EBE110D7B9143A56FF6E29353B587923FB6EAF325974D6670995A3C28D31D95
956A94C0A9BEADDF601FB1862EA7E0F681F81E6F07F1F6613CF6699E390C0E9E007F47D2AA3E7F5EFF7236F52FEBE395E423FB5380BF475F5E2163345E493115A172DDC0477A2D7
AEDDBD89B806EA6101531CC8CF43E9E46EAFCE5CEEB6A878F40CE5A4495F9009956A941D184A5CB306F13C2AE1E2BC7AFC9D06968A7768C335375E190009F7043372F54719144CA
849E16DAF2EED0F9079B0""
>>> len(data)
1304
>>> n = 8
>>> [data[i:i+n] for i in range(0, len(data), n)]
['90DB15BC', 'AD1198A9', 'C360752F', '8A27FD0F', 'AF9DE5A3', 'B2FD88D5', '98A782D3', 'D0E72D09', '680C28FD', 'B5A2F84C', '1ACA1F1B', '51EC0478',
'7193D4F2', 'ABFD5056', '1D2F6E40', '65399C0A', 'C19AF1C6', '4878A577', 'FEB03A83', '958054D1', 'B57F855F', '2310852E', '1FD5BE80', '7EC951F1',
'1ED2CC27', 'CFCEC6A', '702EF8F4', '2B7BF287', 'EEC5838E', 'BEE0637', '06A9ACBB', '78B552FA', '9220A3B6', '9FEBB637', 'E09B4B69', 'FAAE10D9',
'B891A3B1', 'FD8C2ED5', 'E7D72A4B', '9E81875D', 'F6A85019', '3A0F4CA9', '1FED785E', 'B49DC629', '1E405F48', '927E4160', 'B737C649', 'C32D3728',
'299442A6', '842E16D3', 'AB5EA3FC', '472F9EDA', '485C43FB', 'DEF3CFCC', 'E8A8EBDA', '87BE1291', '6F070D5F', 'F2B0D05E', '65CD0805', '3EA85B57',
'7E716ADE', '695FEE2F', '5195C13B', '39E60668', '987149A4', '025B7149', 'A083E70E', '77A4ACDC', '32F93E78', '25138F8F', '9F501089', 'E44F8D7F',
'009CF40D', '557AE4F8', 'F242EA7F', 'AA27B18D', '88FBC1E4', '1D949C9A', '27D6868C', '4EA54AD8', 'A6724F37', '0FCC478E', '909C7881', '7A1B2363',
'BB443369', 'AF6A7415', '94194F06', 'EA3FBC46', '2C82F386', 'F164A76F', '9F744969', 'D590828B', 'B4CFA3B1', '622725D0', '4FED0A41', '37BC717',
'E54E90D5', '24380CFD', '8B609A7A', 'E11522A9', '83CAD3E3', '53AC8E57', 'ED6D23C9', '824ECC82', '0CFB7840', '95497FA9', '81A8E98F', 'CBFC23BD',
'2922B2D3', 'F98D772E', '5F474AB3', '0434B2E5', 'ECA1C30B', '5927FA92', '66C04C1F', '2C69225D', 'EEBE11DD', '789143A5', '6FFE6293', '53B58792',
'3FB6EAF3', 'F325974D', '667D995A', '3D28D31D', '95956A94', 'C0A9BEAD', 'DF601FB1', 'B62EA7E0', 'F681F81E', '6F07F1F6', '613CF669', '9E390C0E',
'9E007F47', 'D2AA3E7F', '5EFF236F', '52EFBE39', '5E4223FB', '538DBF47', '5F5E2163', '345E4931', '15A172DD', 'C0477A2', 'D7AEDDBD', '89B806EA',
'6101531C', 'C8CF43E9', 'E46EAFCE', '5CEEB6A8', '78F40CE5', 'A4495F90', '099956A9', '41D184A5', 'CB306F13', 'C2AE1E2B', 'C7AFC9D6', 'D968A776',
'BC335375', 'E190009F', '7043372F', '54719144', 'CA849E16', 'DAF2EEDE', '0F9079B0']
>>>

```

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 **0x** before every DWORD, and so Python won't treat the list as a list of strings.



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.

```

for i in range(len(encrypt)):
    ...
    mov eax, encrypt
    mov ecx, eax
    |
    xor eax, ebx          - 4 bytes of encrypt are used as xor?
    xor eax, arg_4
    add encrypt, 4
    inc ror_counter
    mov ebx, ecx
    mov cl, byte ror_counter
    ror eax, cl
    mov decrypt, eax
    add decrypt, 4
    -----
    xor encrypt[i], encrypt[i]
    xor encrypt[i], arg_4
    inc ror_counter
    mov cl, byte ror_counter
    ror encrypt[i], cl
    mov decrypt, encrypt[i]
    ...
try:
    encrypt[i] = swap32(encrypt[i])
except Exception as E:
    print encrypt[i]
    print "Error!. Stopping Decryption."
    print E
    break
test = encrypt[i] ^ ebx
ebx = encrypt[i]
test2 = test ^ arg_4
ror_counter = ror_counter + 1
test2 = ror(test2, ror_counter, 32)
final data = pack("<i", test2)

```

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

```

;BG)
(D;0
ICI;
GA;;
;AN)
(A;0
ICI;
GA;;
;AU)
(A;0
ICI;
GA;;
;BA)
Decrypted Strings:
[78, 84, 68, 76, 76, 46, 68, 76, 76, 0, 78, 84, 68, 83, 65, 80, 73, 46, 68, 76, 76, 0, 75, 69, 82, 78, 69, 76, 51, 50, 46, 68, 76, 76, 0, 90, 11
9, 83, 101, 116, 67, 111, 110, 116, 101, 120, 116, 84, 104, 114, 101, 97, 100, 0, 90, 119, 71, 101, 116, 67, 111, 110, 116, 101, 120, 116, 84, 1
04, 114, 101, 97, 100, 0, 90, 119, 87, 111, 119, 54, 52, 82, 101, 97, 100, 86, 105, 114, 116, 117, 97, 108, 77, 101, 109, 111, 114, 121, 54, 52,
0, 90, 119, 87, 111, 119, 54, 52, 81, 117, 101, 114, 121, 73, 110, 102, 111, 114, 109, 97, 116, 105, 111, 110, 80, 114, 111, 99, 101, 115, 115,
54, 52, 0, 46, 98, 105, 110, 0, 76, 111, 97, 100, 76, 105, 98, 114, 97, 114, 121, 65, 0, 37, 48, 50, 117, 45, 37, 48, 50, 117, 45, 37, 48, 50,
117, 32, 37, 48, 50, 117, 58, 37, 48, 50, 117, 58, 37, 48, 50, 117, 13, 10, 0, 54, 52, 0, 82, 116, 108, 69, 120, 105, 116, 85, 115, 101, 114, 84
, 104, 114, 101, 97, 100, 0, 42, 0, 40, 0, 0, 76, 100, 114, 71, 101, 116, 80, 114, 111, 99, 101, 100, 117, 114, 101, 65, 100, 100, 114
, 101, 115, 115, 0, 67, 114, 101, 97, 116, 101, 82, 101, 109, 111, 116, 101, 84, 104, 114, 101, 97, 100, 0, 90, 119, 87, 114, 105, 116, 101, 86,
105, 114, 116, 117, 97, 108, 77, 101, 109, 111, 114, 121, 0, 76, 100, 114, 76, 111, 97, 100, 68, 108, 108, 0, 90, 119, 80, 114, 111, 116, 101,
99, 116, 86, 105, 114, 116, 117, 97, 108, 77, 101, 109, 111, 114, 121, 0, 107, 101, 114, 110, 101, 108, 98, 97, 115, 101, 0, 76, 100, 114, 82, 1
01, 103, 105, 115, 116, 101, 114, 68, 108, 108, 78, 111, 116, 105, 102, 105, 99, 97, 116, 105, 111, 110, 0, 76, 100, 114, 85, 110, 114, 101, 103
, 105, 115, 116, 101, 114, 68, 108, 108, 78, 111, 116, 105, 102, 105, 99, 97, 116, 105, 111, 110, 0, 67, 114, 101, 97, 116, 101, 80, 114, 111, 9
9, 101, 115, 115, 65, 0, 67, 114, 101, 97, 116, 101, 80, 114, 111, 99, 101, 115, 115, 87, 0, 67, 114, 101, 97, 116, 101, 80, 114, 111, 99, 101,
115, 115, 65, 115, 85, 115, 101, 114, 65, 0, 67, 114, 101, 97, 116, 101, 80, 114, 111, 99, 101, 115, 115, 65, 115, 85, 115, 101, 114, 87, 0, 118
, 101, 114, 115, 105, 111, 110, 61, 37, 117, 38, 115, 111, 102, 116, 61, 37, 117, 38, 117, 115, 101, 114, 61, 37, 48, 56, 120, 37, 48, 56, 120,
37, 48, 56, 120, 37, 48, 56, 120, 38, 115, 101, 114, 118, 101, 114, 61, 37, 117, 38, 105, 100, 61, 37, 117, 38, 116, 121, 112, 101, 61, 37, 117,
38, 110, 97, 109, 101, 61, 37, 115, 0, 73, 115, 87, 111, 119, 54, 52, 80, 114, 111, 99, 101, 115, 115, 0, 77, 0, 105, 0, 99, 0, 114, 0, 111, 0,
115, 0, 111, 0, 102, 0, 116, 0, 0, 0, 87, 111, 119, 54, 52, 69, 110, 97, 98, 108, 101, 87, 111, 119, 54, 52, 70, 115, 82, 101, 100, 105, 114, 1
01, 99, 116, 105, 111, 110, 0, 68, 58, 40, 68, 59, 79, 73, 67, 73, 59, 71, 65, 59, 59, 59, 66, 71, 41, 40, 68, 59, 79, 73, 67, 73, 59, 71, 65, 5
9, 59, 59, 65, 78, 41, 40, 65, 59, 79, 73, 67, 73, 59, 71, 65, 59, 59, 59, 65, 85, 41, 40, 65, 59, 79, 73, 67, 73, 59, 71, 65, 59, 59, 66, 6
5, 41]
NTDLL.DLLNTDSAPI.DLLKERNEL32.DLLZwSetContextThreadZwGetContextThreadZwWow64ReadVirtualMemory64ZwWow64QueryInformationProcess64.binLoadLibraryA%0
2u-%02u-%02u-%02u-%02u
64RtlExitUserThread*.LdrGetProcedureAddressCreateRemoteThreadZwWriteVirtualMemoryLdrLoadDllZwProtectVirtualMemorykernelbaseLdrRegisterDllNotifi
cationLdrUnregisterDllNotificationCreateProcessACreateProcessWCreateProcessAsUserCreateProcessAsUserWrsion=%u&soft=%u&user=%08x%08x%08x%08x&s
erver=%u&id=%u&type=%u&name=%sIsWow64ProcessMicrosoftWow64EnableWow64FsRedirectionD:(D;OICI;GA;;BG)(D;OICI;GA;;AN)(A;OICI;GA;;AU)(A;OICI;GA;;
;BA)
DONE

```

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.

```

import idc

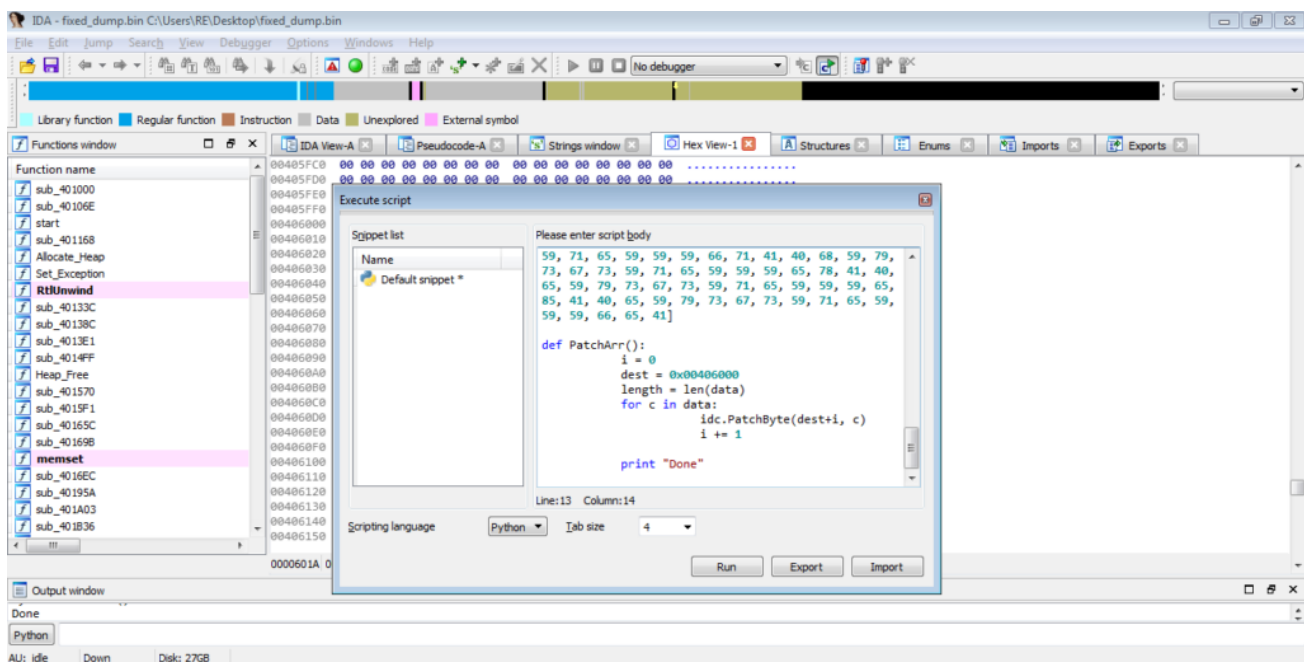
data = []

def PatchArr():
    i = 0
    dest = 0x00406000 # BSS Start offset
    length = len(data)
    for c in data:
        idc.PatchByte(dest+i, c)
        i += 1

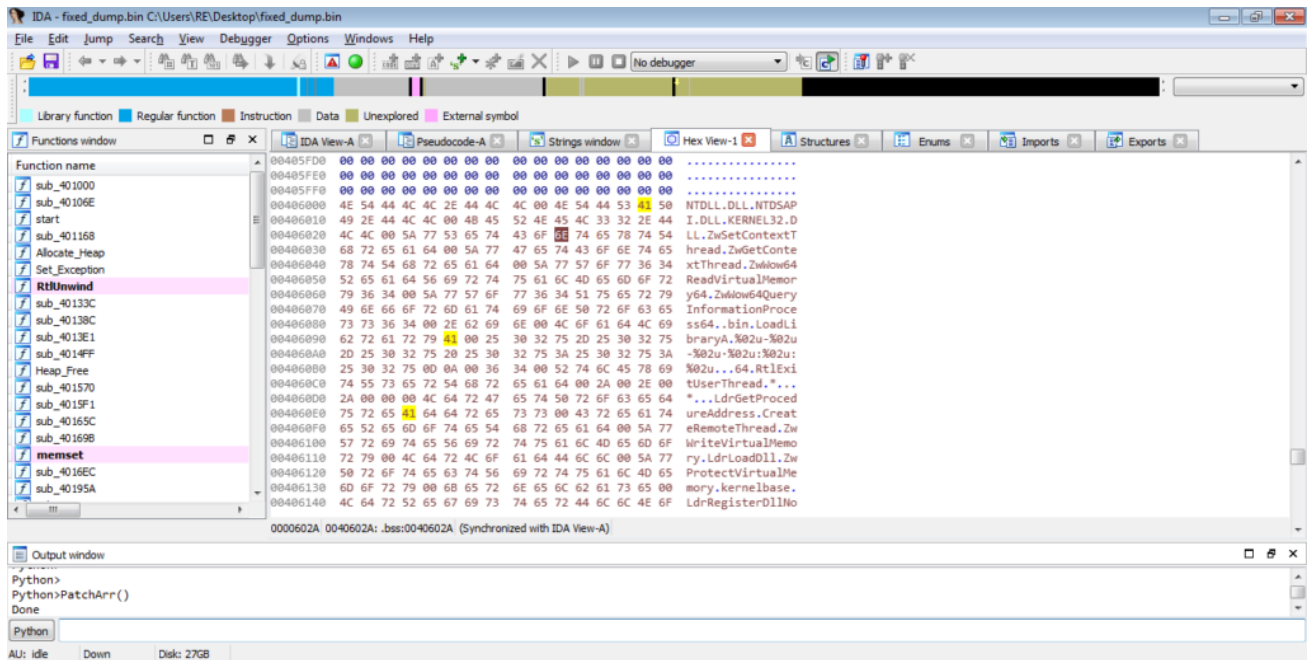
print "Done"

```

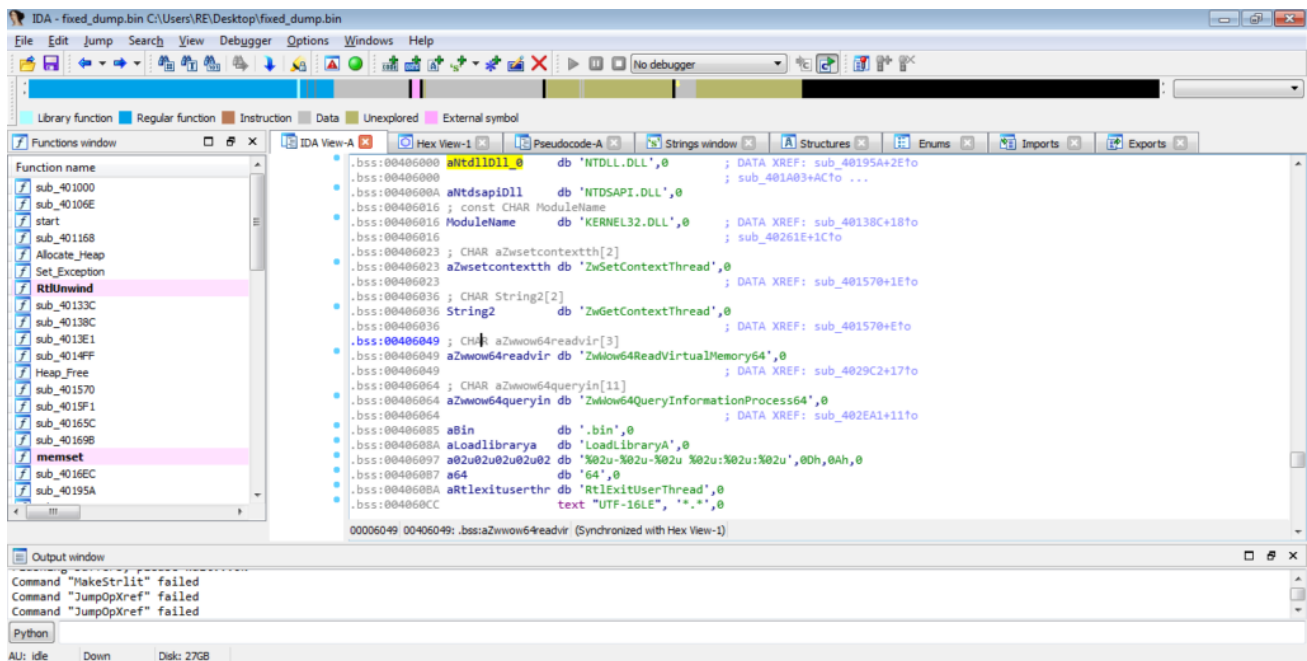
From there, we need to import it as a script/module in IDA, so you can do that either by importing the file, or by copying the text and pasting it into the box as shown below. All you need to do then is click **Run**, and it should be imported – although we physically have to call the script, **Run** 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.



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.

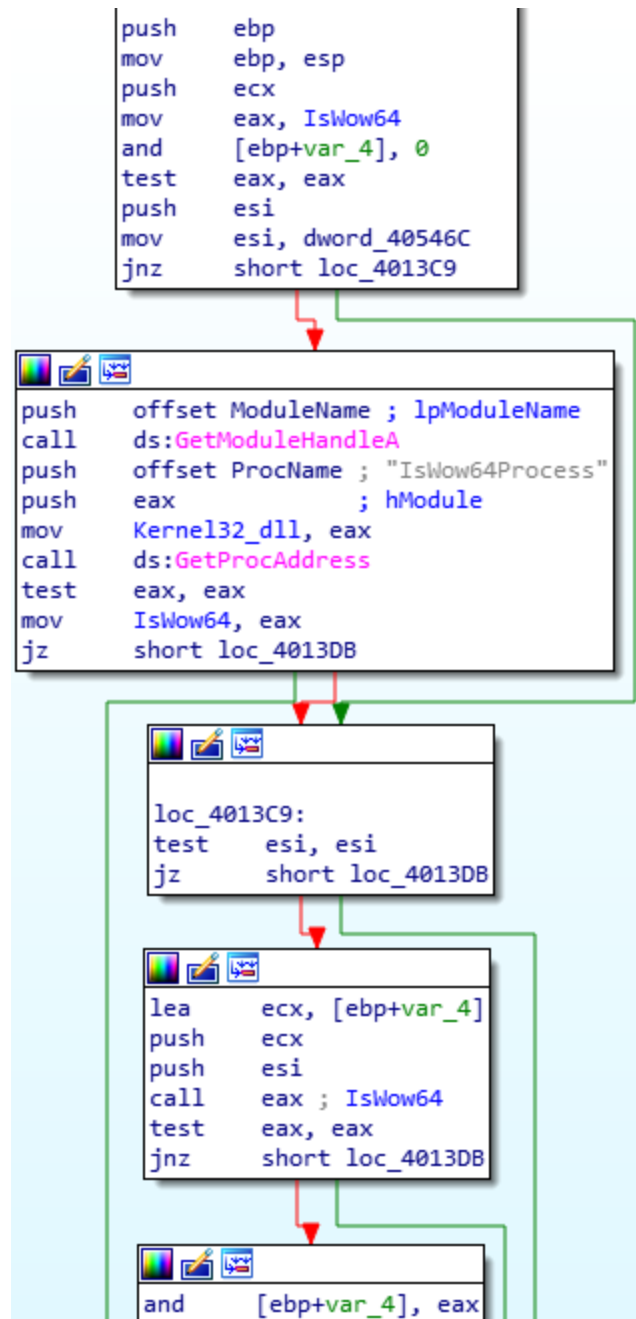


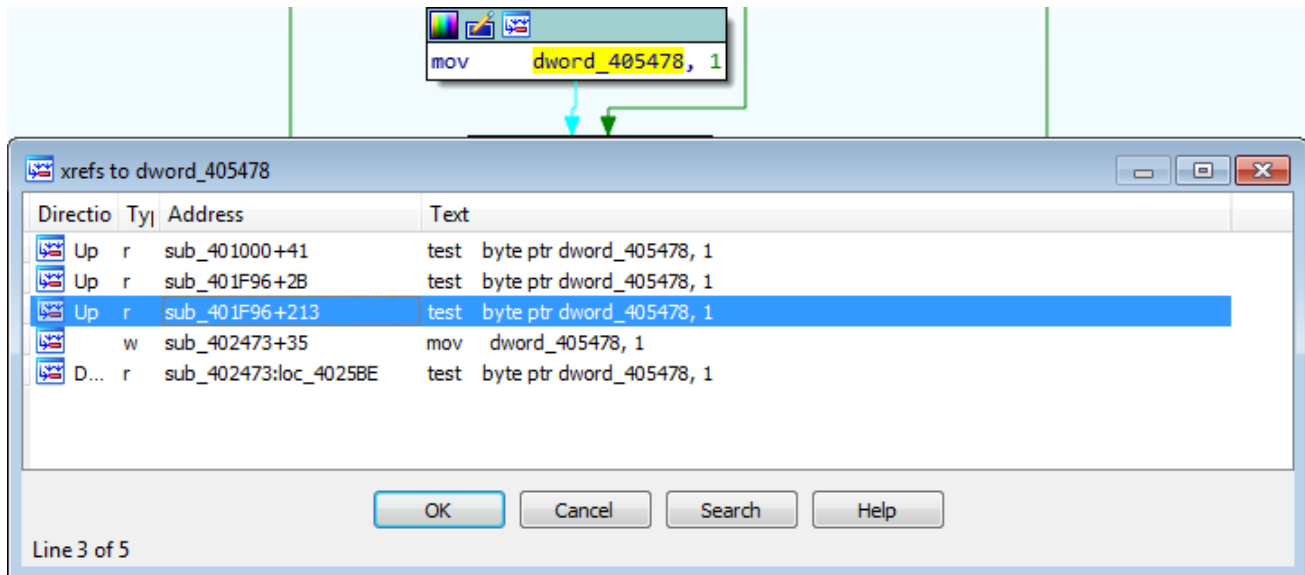
With the newly decrypted BSS section, we should be able to analyse the rest of the payload without issues.

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 result is not 64 bit, the variable **var_4** in the graph below is used in

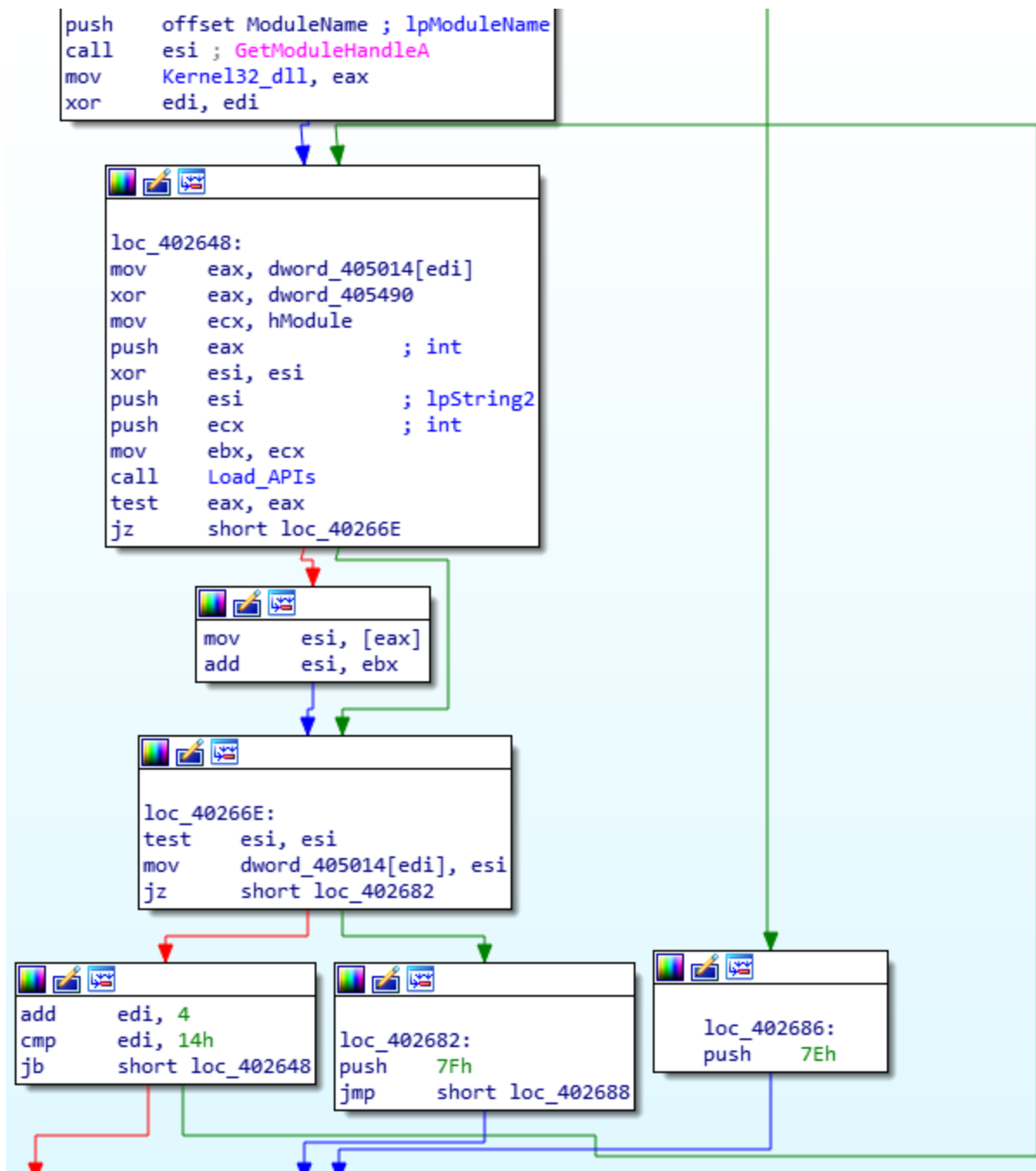
an AND operation with EAX, which would be equal to **0**, meaning the value in **var_4** would be **0**. If the system is 64 bit, this is skipped. Then, regardless of the system architecture, the value in **var_4** is moved into EAX, which is the return value.

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.

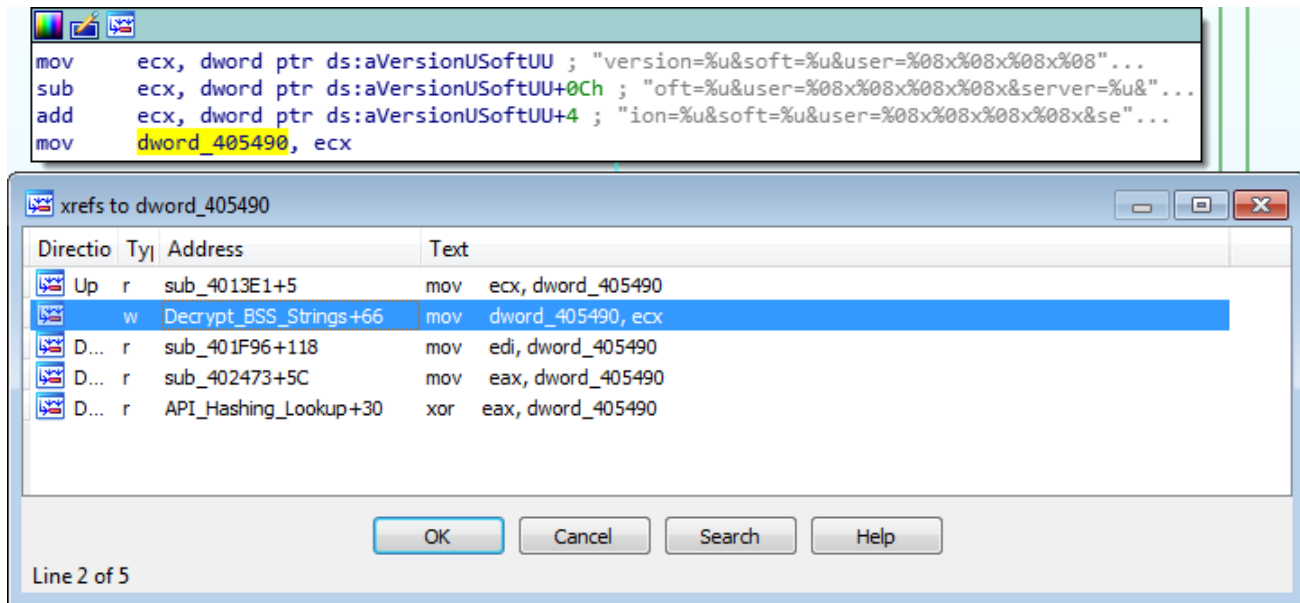




The next function that is called is quite complex. In a nutshell, this function is responsible for getting the address to a select few API calls. To do this, it uses a list of predefined values, a “key”, and a hashing routine. First, let’s take a look at the key and find out what it is. Looking at the 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.



```

int __thiscall Decrypt_BSS_Strings(void *this)
{
    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;
}

```

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="
```

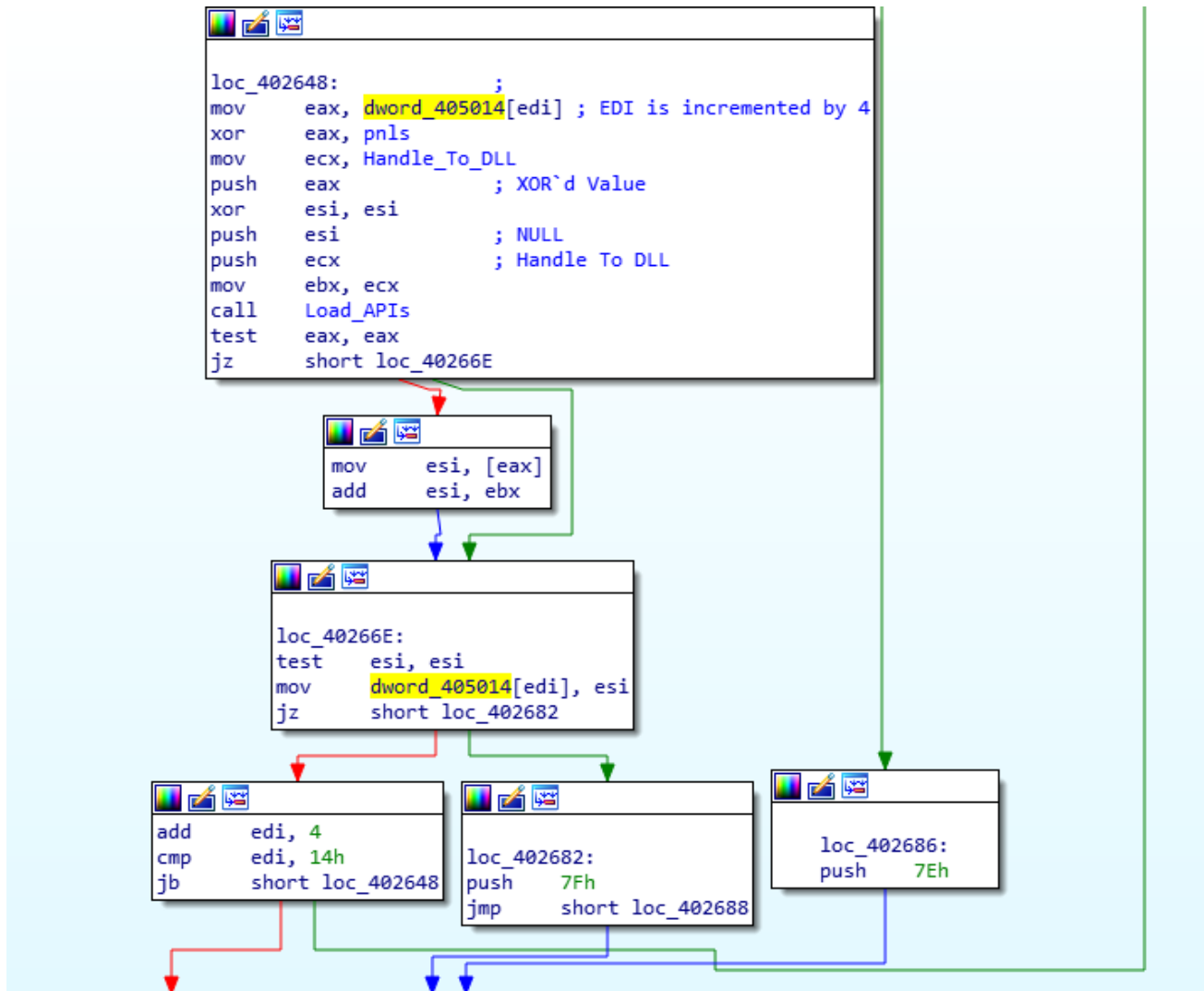
We need to convert these values to hex to perform addition and subtraction:

```
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.

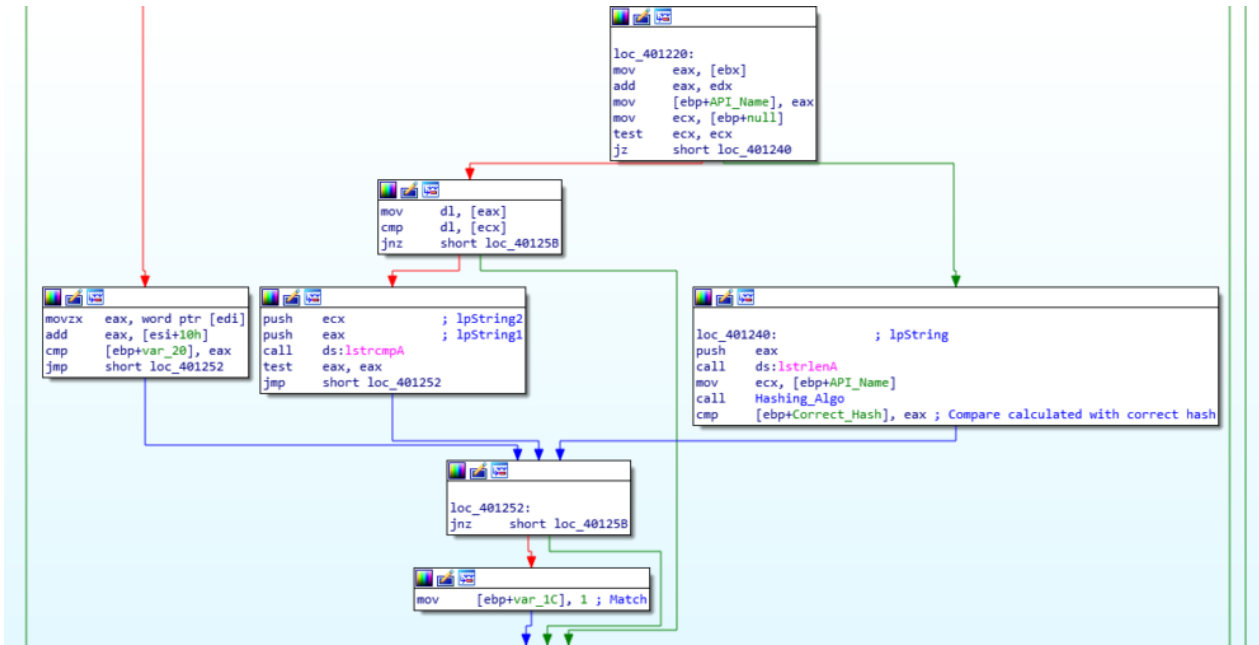
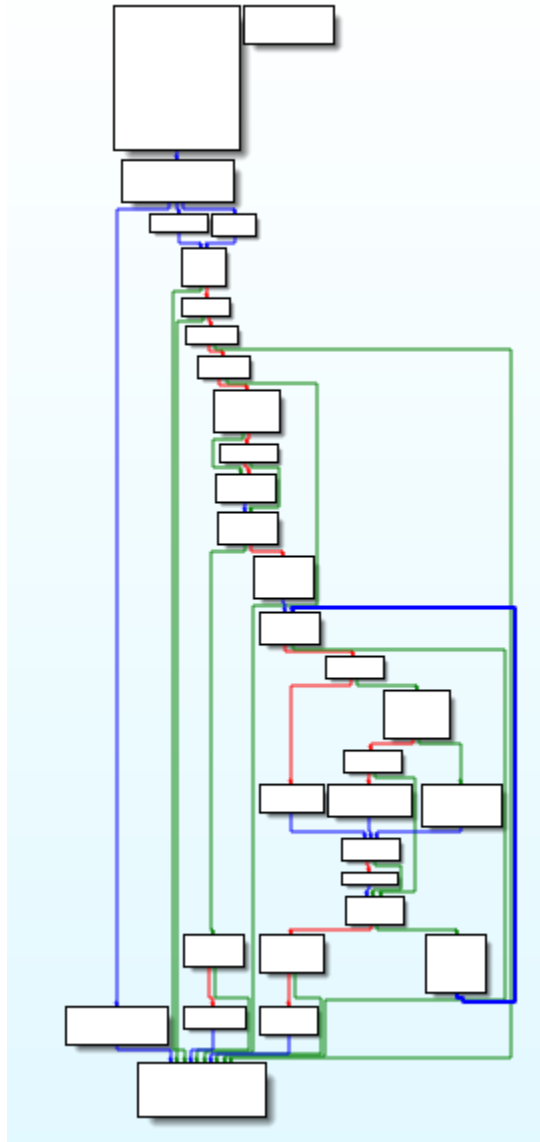


```

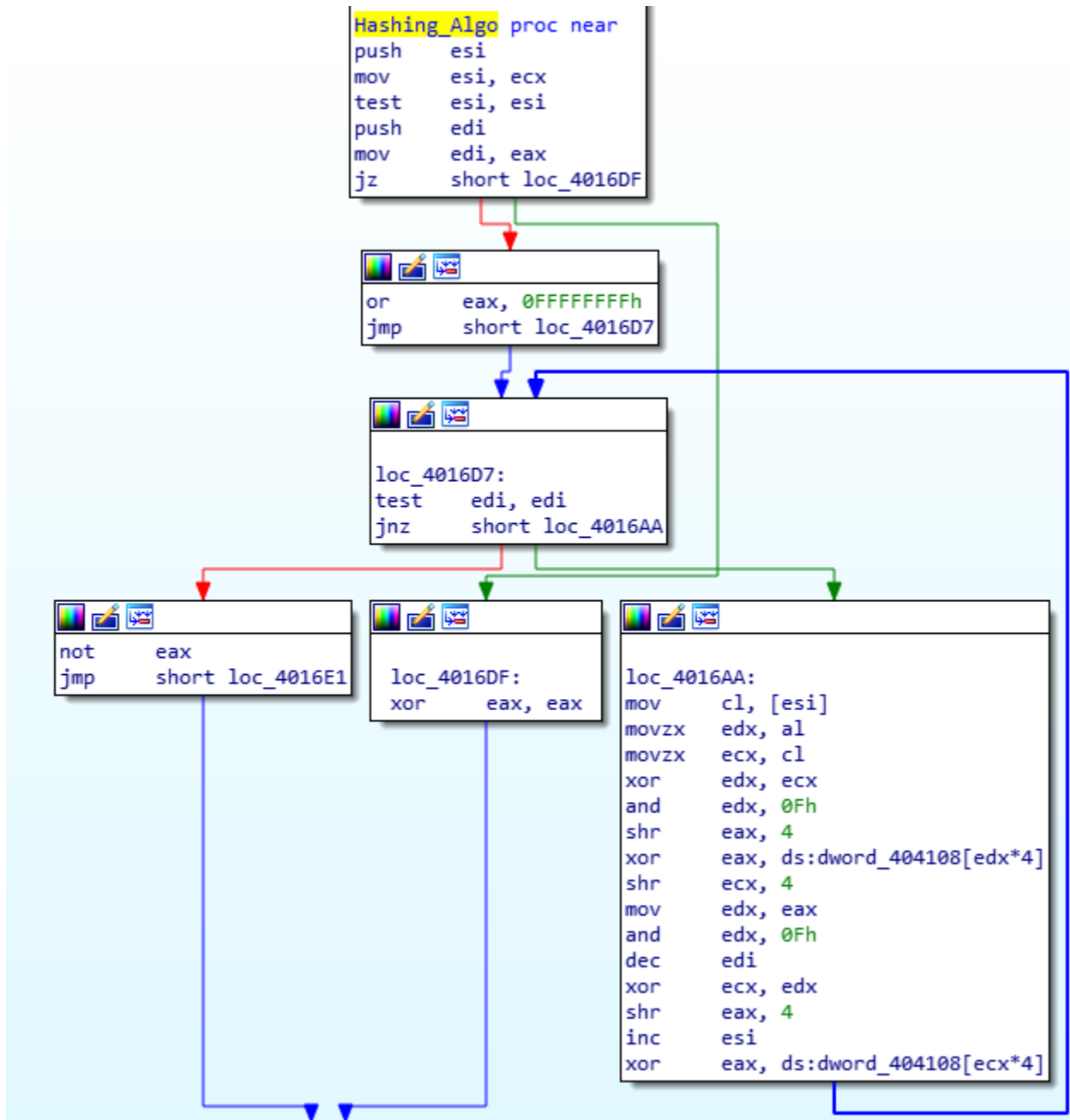
.data:00405011 db 0
.data:00405012 db 0
.data:00405013 db 0
.data:00405014 ; int dword_405014[]
.data:00405014 dword_405014 dd 29510894h ; DATA XREF: sub_401CDD↑r
.data:00405018 dword_405018 dd 1B477A8Dh ; API_Hashing_Lookup:loc_402648↑r ...
.data:0040501C dword_40501C dd 1D1D4817h ; DATA XREF: sub_40234A+D7↑r
.data:00405020 dword_405020 dd 6569CE63h ; DATA XREF: sub_401EDF↑r
.data:00405024 dword_405024 dd 54B334FBh ; DATA XREF: sub_40133C+5↑r
.data:00405028 db 80h ; €
.data:00405029 db 0

```

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 0, 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.

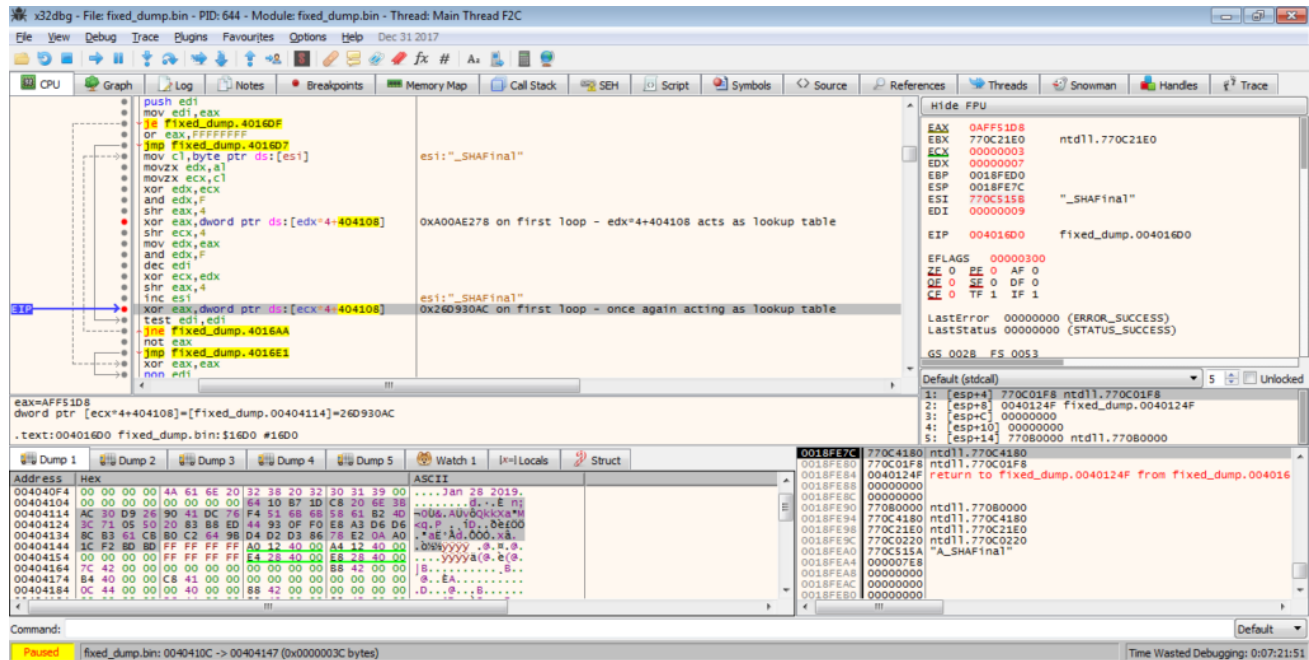


Taking a look at the actual hashing function, it is quite difficult to come to the conclusion (especially for a beginner) that the function is responsible for hashing, until it is run in a debugger. There is definitely something happening in this function from a static analysis perspective, as there 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 pseudocode), 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: 0xA00AE278. 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> ▼

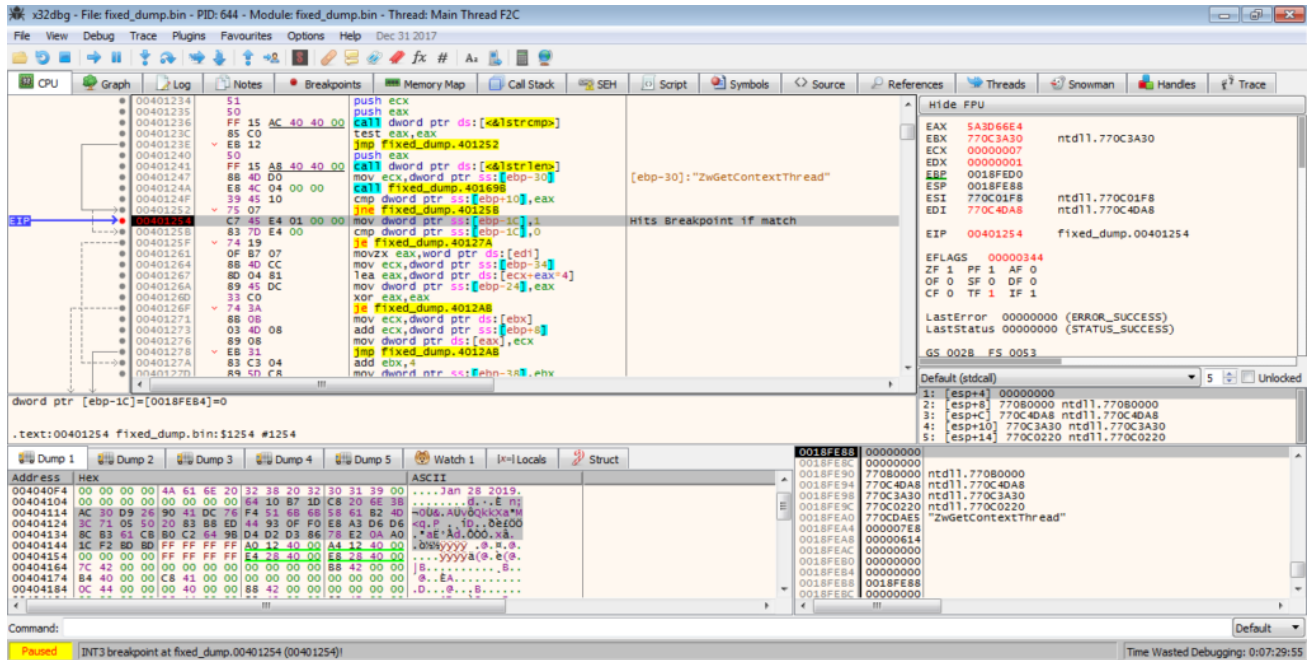
14 Feb 2019 - ... 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, ...

crc32.c

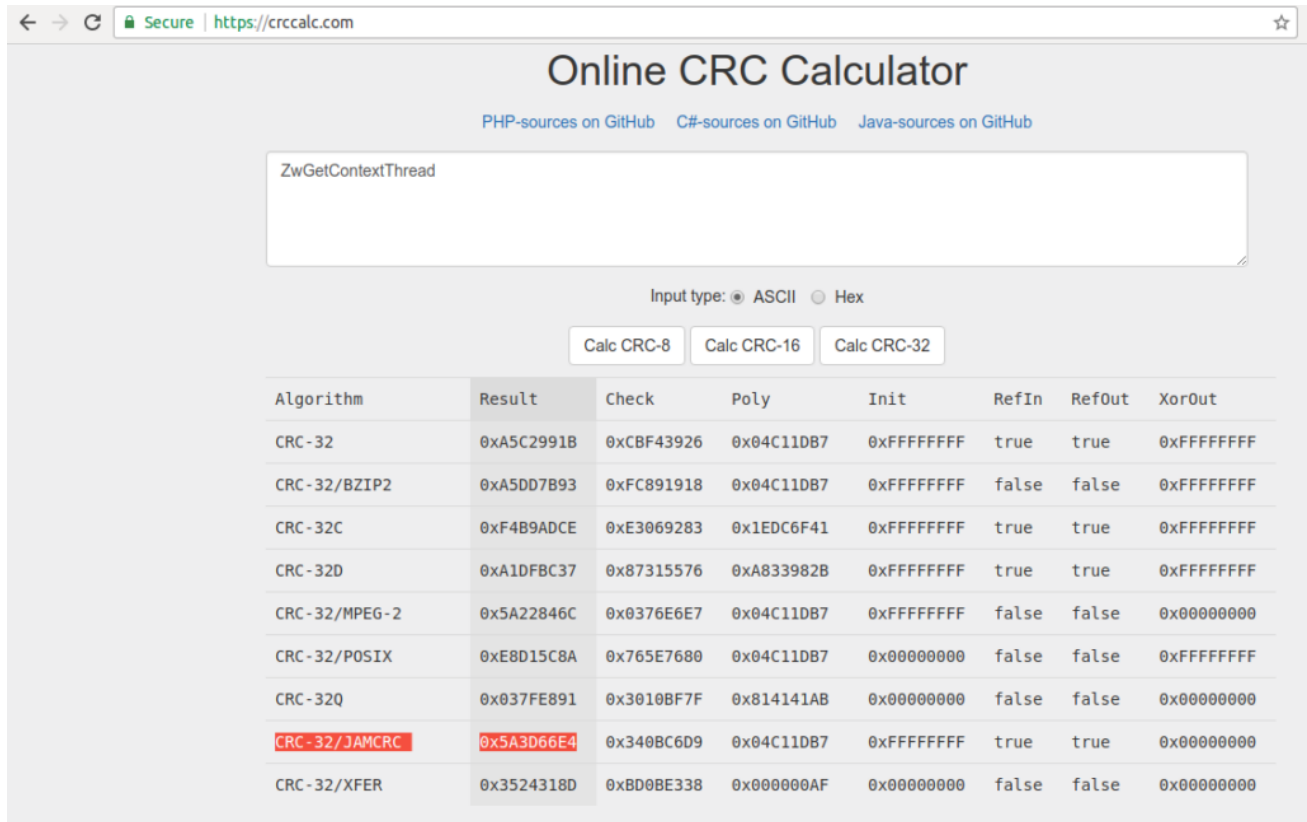
<ftp://ftp.ncdc.noaa.gov/pub/data/software/nexrad2/crc32.c> ▼

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

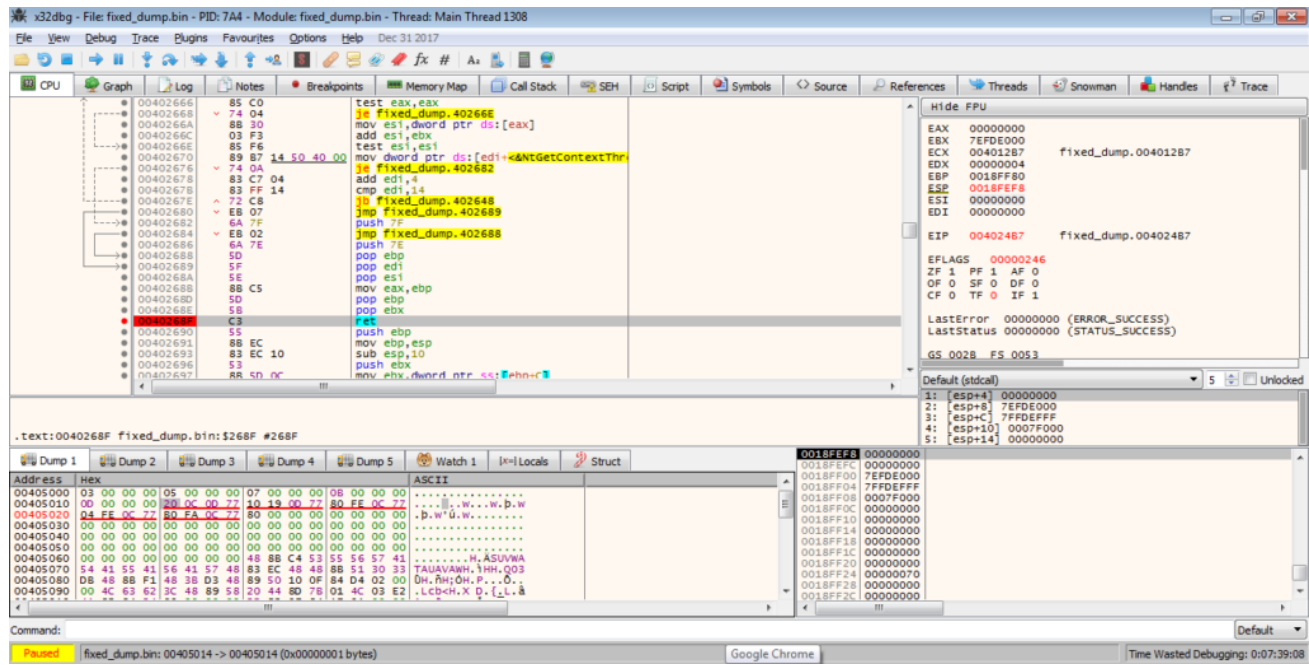
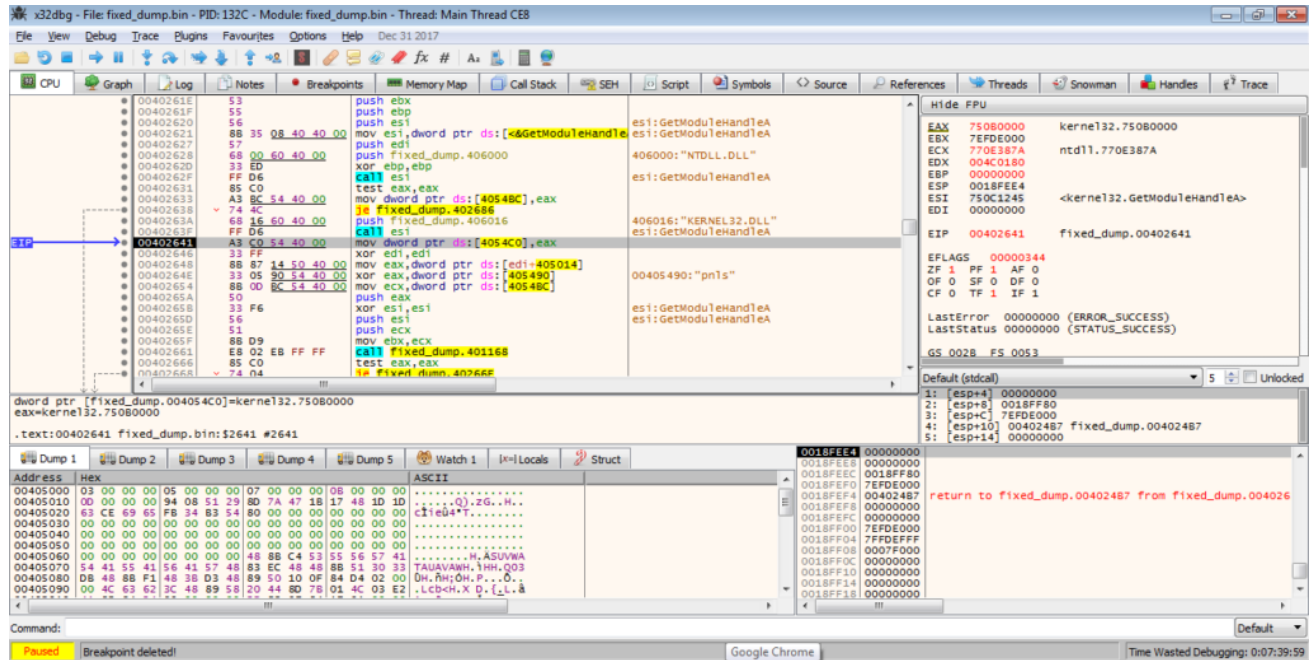
In this specific sample, the program loops through the NTDLL.DLL export functions and hashes them all until it finds a matching hash. What makes hashing worse is that hashes are irreversible, meaning the only way you can find the matching hash is by brute forcing – to do this you could hash each API 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 [site](#), 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.



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 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
000001E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001F0 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 .....text...
00000200 6F 26 00 00 00 10 00 00 00 30 00 00 00 10 00 00 o&.....0.....
00000210 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 .....
00000220 2E 72 64 61 74 61 00 00 9E 06 00 00 00 40 00 00 .rdata..Z...@..
00000230 00 10 00 00 00 40 00 00 00 00 00 00 00 00 00 00 .....@.....
00000240 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 .....@.@.data...
00000250 14 05 00 00 00 50 00 00 00 10 00 00 00 50 00 00 .....P.....P..
00000260 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 .....@..@..@..@
00000270 2E 62 73 73 00 00 00 00 8D 02 00 00 00 60 00 00 .bss.....`..
00000280 00 10 00 00 00 60 00 00 00 00 00 00 00 00 00 00 .....
00000290 00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00 .....@..@.rsrc...
000002A0 10 00 00 00 00 70 00 00 00 10 00 00 00 70 00 00 .....P.....P..
000002B0 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 .....@..@..@..@
000002C0 2E 72 65 6C 6F 63 00 00 00 70 00 00 00 80 00 00 .reloc...p...@..
000002D0 00 6E 00 00 00 80 00 00 00 00 00 00 00 00 00 00 .....n...@.....
000002E0 00 00 00 00 40 00 00 40 00 00 00 00 00 00 00 00 .....@..@.....
000002F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000310 4A 4A 00 01 8F C6 B1 D4 0C 4A 15 9E 00 84 00 00 JJ...Z+O.J.Z...
00000320 00 B6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000350 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000360 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000370 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000380 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000390 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000003D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
Offset(h): 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.

```
loc_4024B2:
call  API_Hashing_Lookup
cmp   eax, esi
jnz   loc_40260C

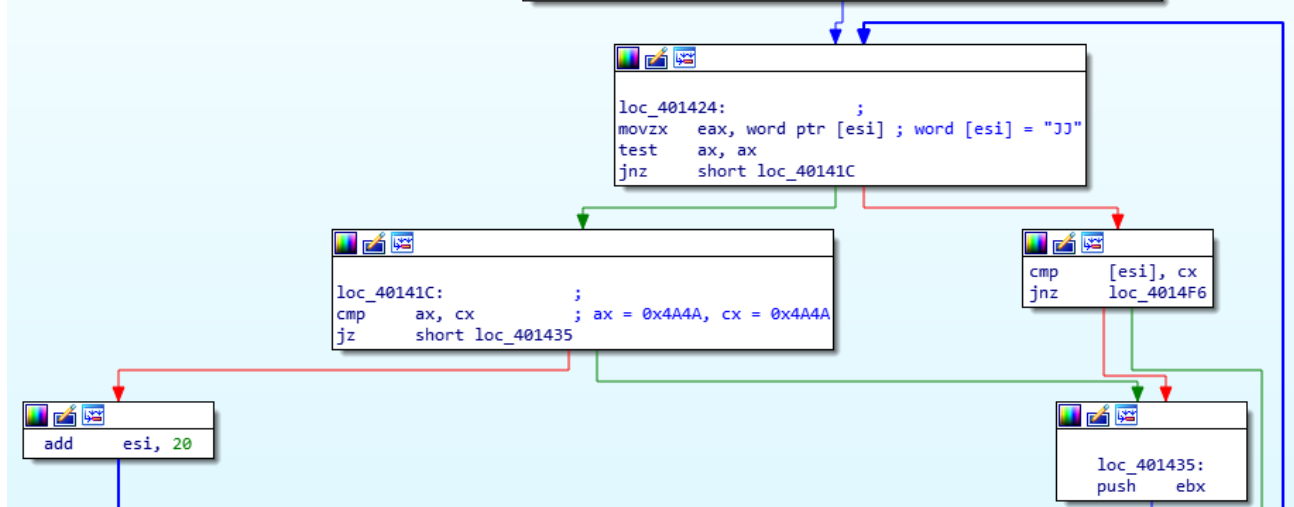
call  Get_FileName
push  6
xor   eax, eax
pop   ecx
lea   edi, [esp+88h+var_58]
rep  stosd
mov   eax, pnls
xor   eax, 0ED79247Ch ; 0x9E154A0C
push  eax             ; hash
lea   eax, [esp+8Ch+var_48]
push  eax             ; int
lea   eax, [esp+90h+var_58]
push  eax             ; int
call  sub_4013E1      ; Parse Structure
test  eax, eax
jz    loc_402609
```

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.

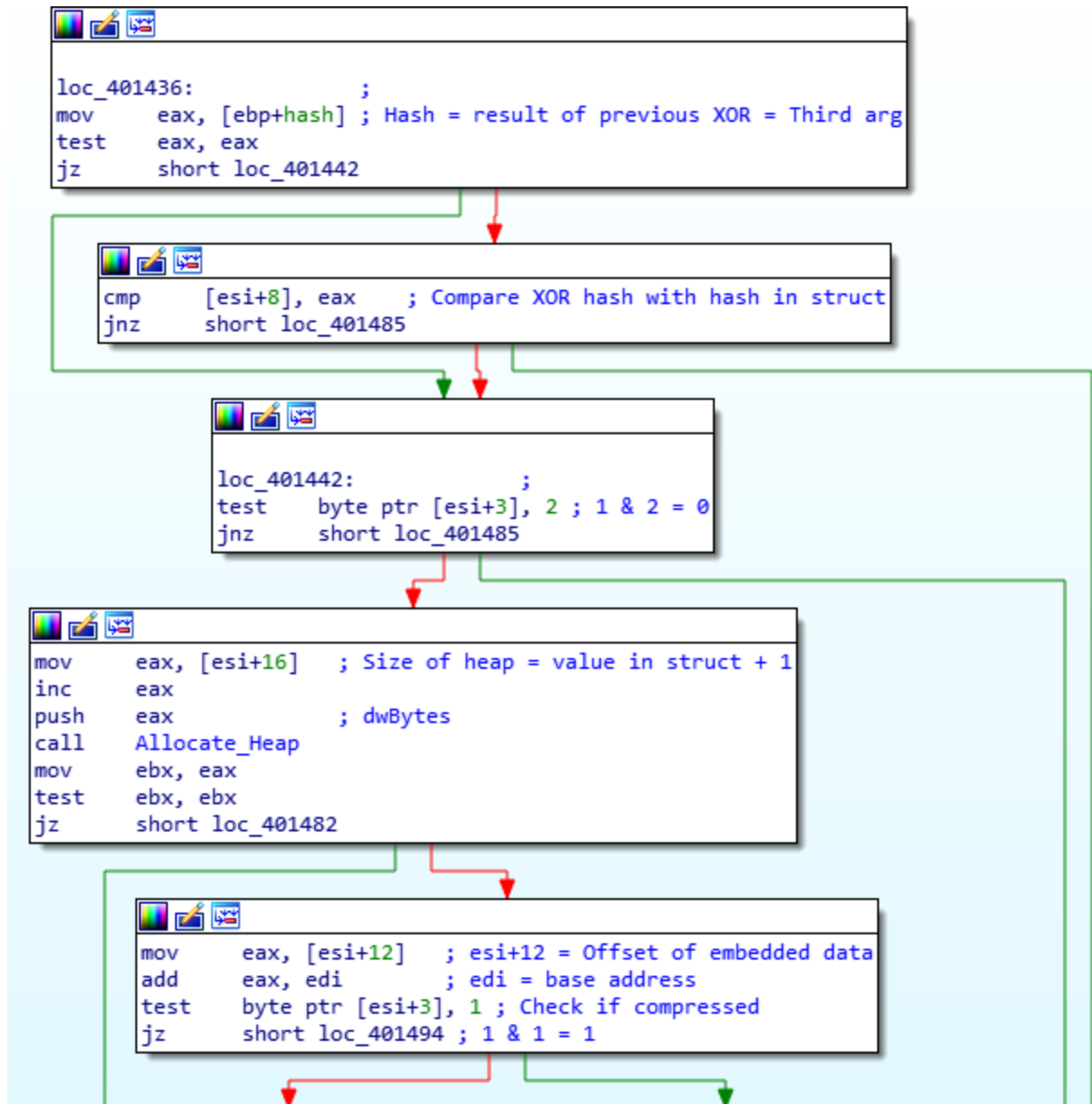
```

push    ebp
mov     ebp, esp
push   ecx
push   ecx
mov    ecx, pnls
and    [ebp+var_4], 0
push   esi
push   edi
mov    edi, base_addr
mov    eax, [edi+3Ch] ; Value from MZ Header
add    eax, edi      ; Base Addr + 0x100 = Start of PE Header
movzx  edx, word ptr [eax+6] ; 0x06
movzx  esi, word ptr [eax+14h] ; 0xE0
imul   edx, 28h     ; 0xF0
xor    ecx, 243Ah   ; 0X736C4A4A
movzx  ecx, cx      ; 0X4A4A
add    edx, eax     ; Start of section table
mov    [ebp+var_8], ecx ; 0x4A4A
lea    esi, [esi+edx+40h] ; 0xE0 + 0x4001F0 + 0x40 = 0x400310
jmp    short loc_401424

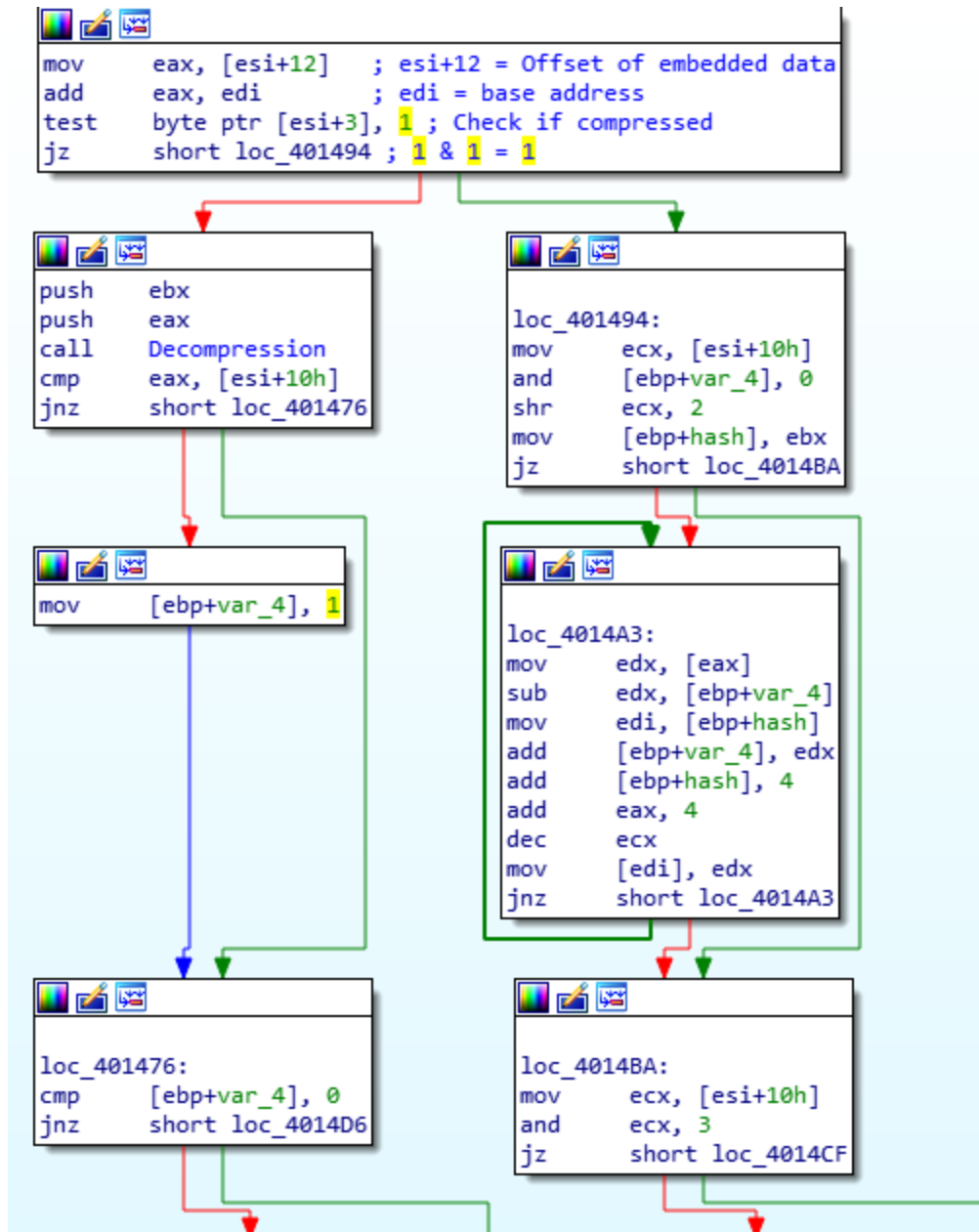
```



Once the structure has been located successfully, the function compares the third argument (0x9E154A0C) with the embedded CRC-32 hash, which is also 0x9E154A0C. 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.

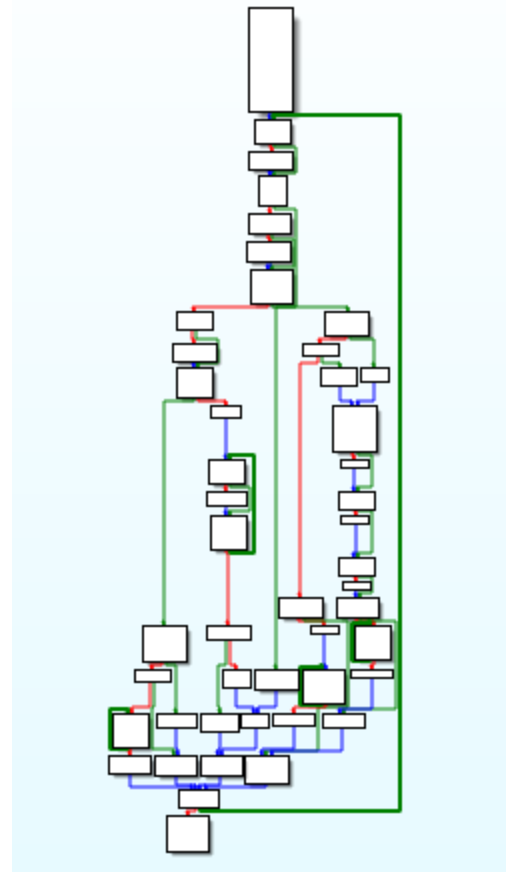


Looking at an overview of the function, you can see it is quite intricate and difficult to follow. Remember what I was saying about crypto constants earlier? They are also present in some, if not all, (de)compression routines, although it is much more difficult to identify the algorithm compared to encryption 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 0x7D00. Using this, we can search “0x7d00 compression” on Google and after scrolling down a bit, come across this [site](#). Comparing this python code:


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

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.

```
mov    esi, eax
call   sub_4030C0
cmp    esi, 7D00h
jb     short loc_402D6A
```

```
add    eax, 1
```

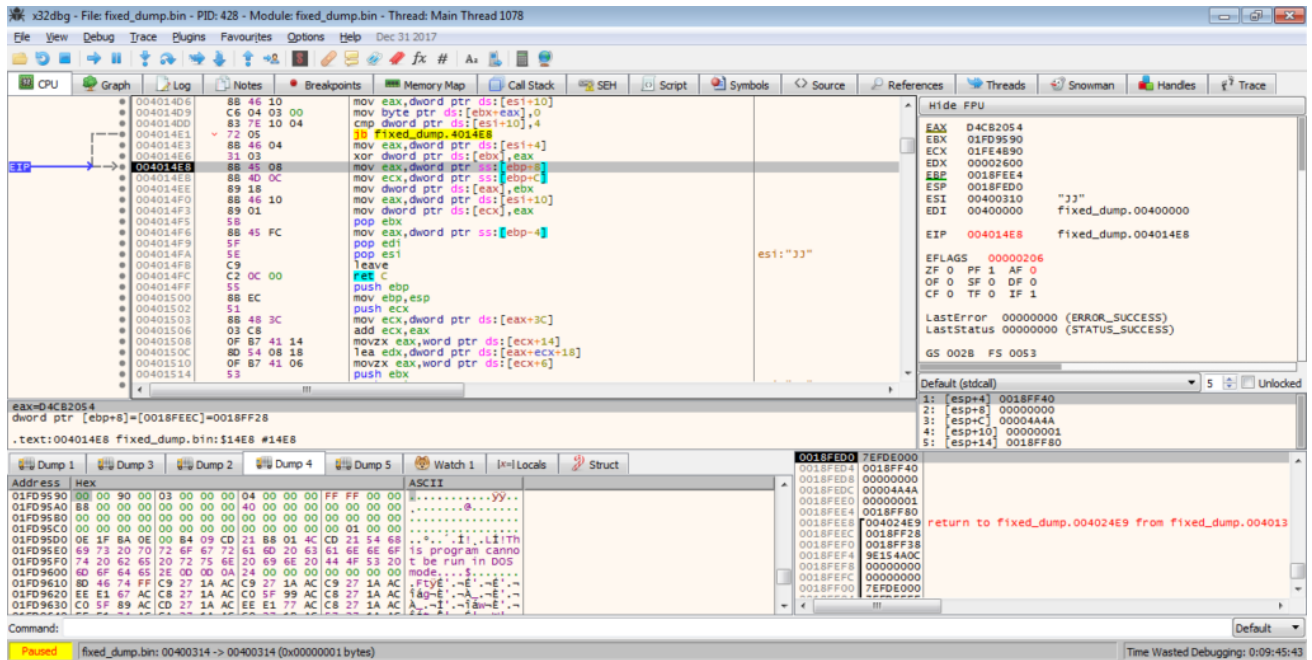
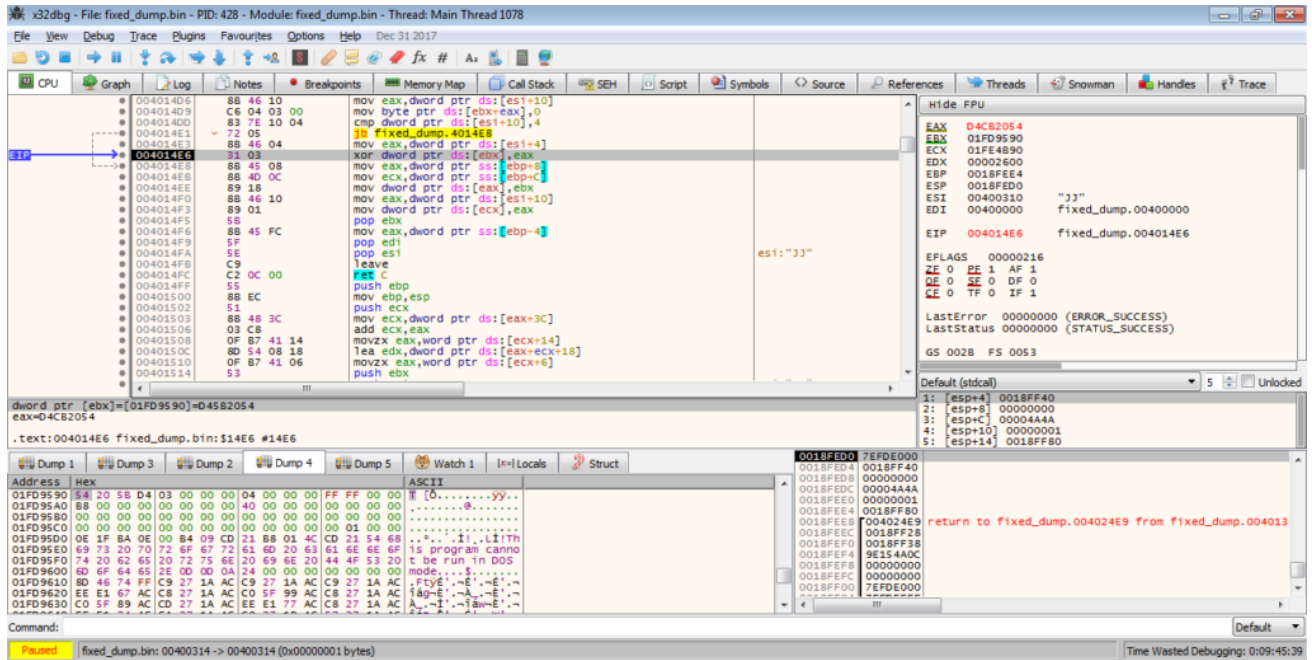
```
loc_402D6A:
cmp    esi, 500h
jb     short loc_402D75
```

```
add    eax, 1
```

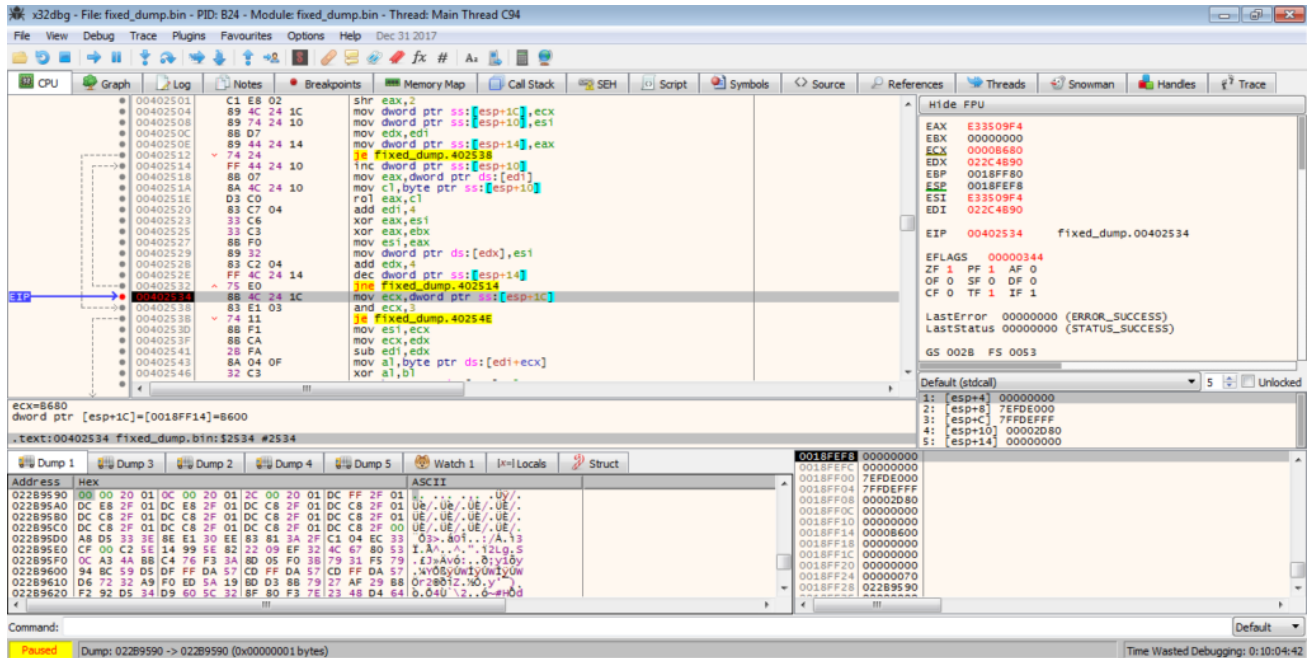
```
loc_402D75:
cmp    esi, 80h
jnb    short loc_402D80
```

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	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	A4	..E!Ô...æ.ÿ`p,
00008410	01	70	40	85	79	01	87	06	0E	1F	BA	3E	00	B4	09	CD	.p@...y.+...°>.'Í
00008420	21	B8	FA	4C	C0	0A	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`Íu
00008440	5F	98	69	06	44	4F	7E	53	03	6D	6F	64	65	2E	0D	89	_~i.DO~S.mode..%
00008450	0A	24	4C	43	8D	01	46	74	FF	C9	27	1A	AC	58	04	EE	.\$LC..FtyÉ'.-X.i
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	0A	28	7C	47	+Ï.tŠÊ,ÿ..W..(G
00008480	43	CA	3E	45	21	C8	95	15	10	67	50	6B	C5	EC	4A	08	CÊ>E!È*..gPkÁiJ.
00008490	60	38	65	62	41	08	52	69	63	68	44	68	68	C7	4C	38	`8ebA.RichDhhÇL8
000084A0	01	05	05	10	1B	4F	5C	63	10	E0	80	02	21	0B	33	01O\c.à€!.!3.
000084B0	08	12	8A	29	1B	26	14	0C	A5	5A	0B	10	CA	09	A0	0F	..Š).&..¥Z..Ê. .
000084C0	BD	54	0C	02	92	38	86	34	F1	4A	3D	47	0D	29	02	2B	T..'8+4fiJ=G.)+.
000084D0	B4	58	39	08	0D	47	12	82	33	11	A6	02	50	4F	64	44	`X9..G.,3. .POdD
000084E0	D0	0D	C4	BA	1A	01	A4	49	A8	40	23	A0	B4	A2	29	08	Ð.Ä°..I`@# `c).
000084F0	C0	1F	58	2E	74	39	65	78	E2	11	0F	88	BA	91	12	EC	À.X.t9exá..^°`.i
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	A4	27	C8	D8	02	95	æ..VŽ(€@..'Èø.*
00008520	B0	FD	CA	9C	28	F8	C0	2E	38	62	73	D0	0C	DD	70	0A	°ýÊæ(zÀ.8bsÐ.ÿp.
00008530	AC	99	81	58	9E	4A	28	C0	72	65	6C	6F	63	AE	9C	53	-.XžJ(Àreloc@æS
00008540	D0	28	21	AA	5B	9C	8E	0F	A3	4A	8F	00	11	D9	CF	B1	Ð(!*[æŽ.£J...ÛI±
00008550	07	D4	64	5E	28	E1	04	D8	0A	84	14	41	41	9C	C8	01	.Ôd^(á.ø...AAæÈ.
00008560	E1	DD	D1	8F	C1	DA	01	B0	01	C2	BD	01	FE	01	56	57	áyŇ.ÁÚ.°.Â.p.vW
00008570	6A	28	E8	55	4B	80	12	8B	F0	85	F6	74	4F	B0	8F	1D	j(èUK€.<ð...øt°..
00008580	18	56	89	1C	1C	EE	0C	0E	15	A0	3C	19	10	0C	88	11	.V%..i...<...^.
00008590	83	F8	FB	03	28	24	74	1E	68	5A	AE	1A	32	6A	01	2A	føù.(ſt.hZ@.2j.*
000085A0	8C	23	85	C0	0E	28	20	74	0A	A3	35	54	B2	06	1A	33	€#..À.(t.£5T*..3
000085B0	FF	EB	17	2E	44	3C	8B	3E	F8	85	01	74	0B	56	E8	9F	ÿè..D<<>ø...t.Vèÿ
000085C0	59	80	A6	EB	03	6A	08	5F	8B	73	C7	1D	5E	C3	55	8E	Y€ é.j._<sç.^ÁUŽ
000085D0	EC	83	46	14	53	DB	7D	27	7D	8D	61	F0	5E	E8	ED	4C	ìfF.SÛ}')..aš^èiL
000085E0	BB	F9	18	DC	4A	F3	D8	85	3F	DB	74	00	FF	75	0C	8D	»ù.ÛJóø?...?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.



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.



First, it creates a new file mapping, and calls `MapViewOfFile`, and then copies its 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.

```

call    ds:GetCurrentProcessId
push    eax
lea     eax, [esp+8Ch+Name]
push    offset asc_4040F0 ; "%x"
push    eax                ; LPWSTR
call    ds:wsprintfW
add     esp, 0Ch
lea     eax, [esp+88h+Name]
push    eax                ; lpName
push    lpString2         ; lpString
call    ds:lstrlenW
lea     eax, [eax+eax+2]
push    eax                ; dwMaximumSizeLow
push    esi                ; dwMaximumSizeHigh
push    4                  ; flProtect
push    esi                ; lpFileMappingAttributes
push    0FFFFFFFFh        ; hFile
call    ds:CreateFileMappingW
cmp     eax, esi
jz      short loc_4025BE

```

```

push    esi                ; dwNumberOfBytesToMap
push    esi                ; dwFileOffsetLow
push    esi                ; dwFileOffsetHigh
push    6                  ; dwDesiredAccess
push    eax                ; hFileMappingObject
call    ds:MapViewOfFile
mov     edi, eax
cmp     edi, esi
jz      short loc_4025BE

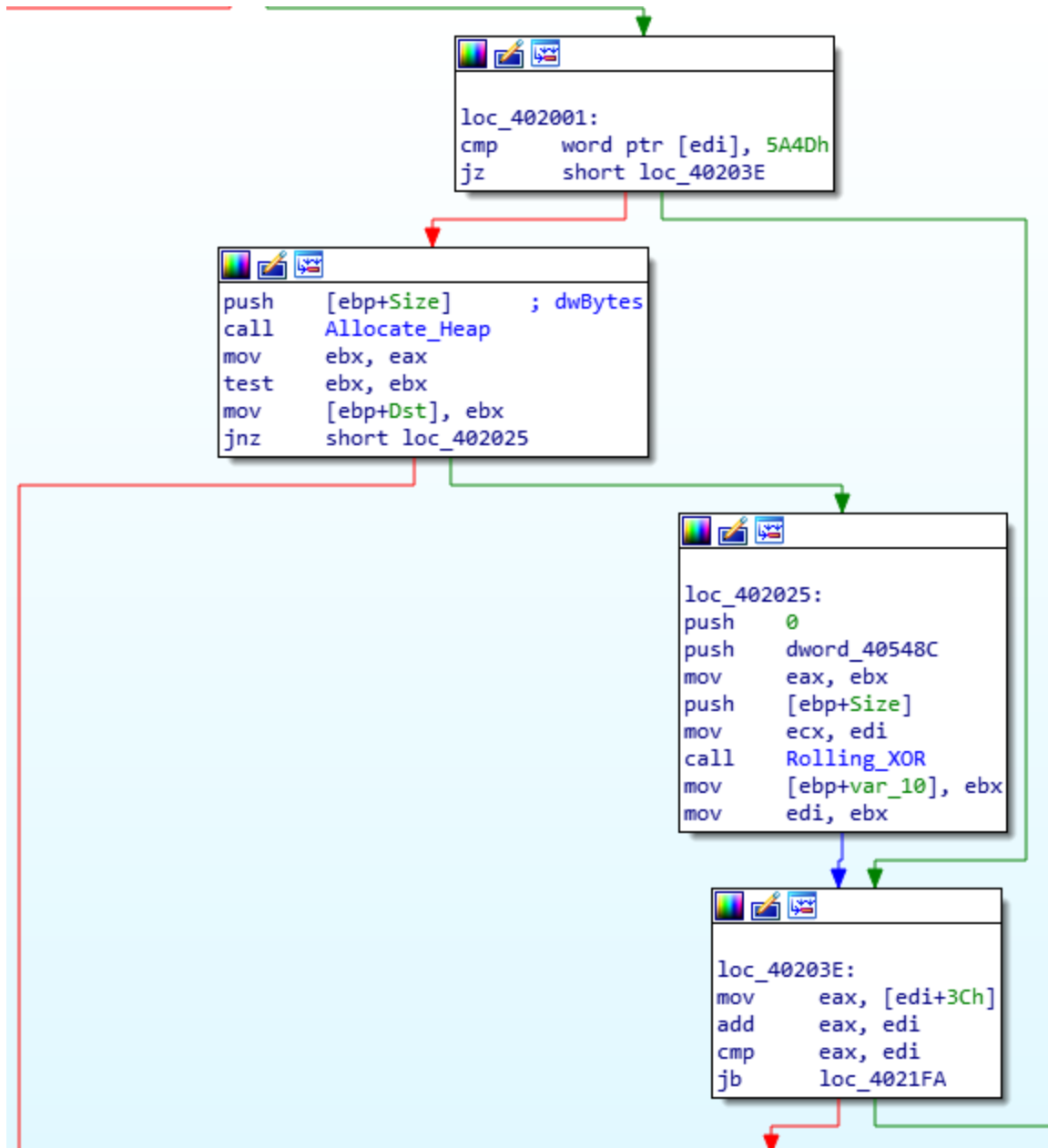
```

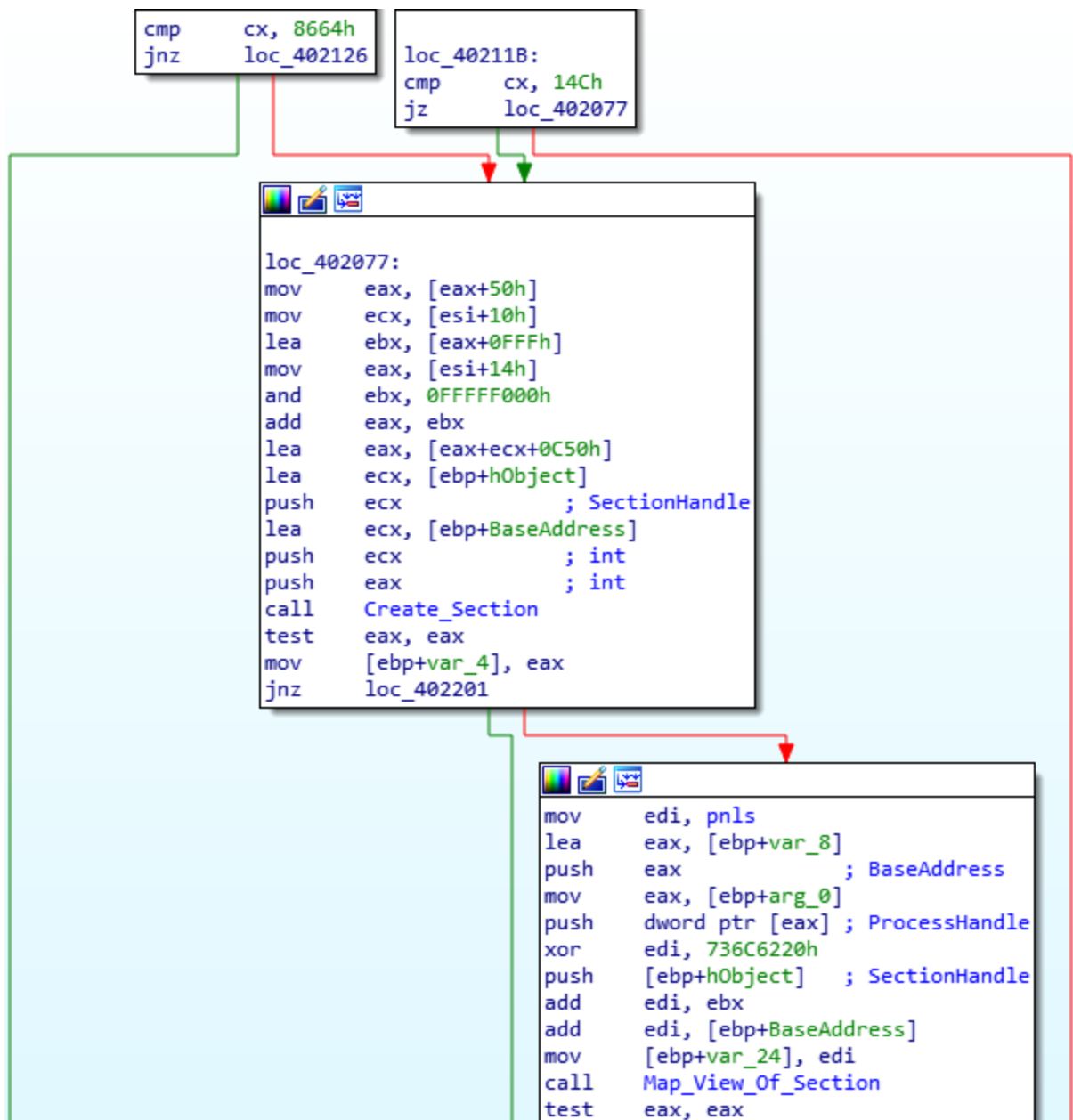
```

push    lpString2         ; lpString2
push    edi                ; lpString1
call    ds:lstrcpyW
push    edi                ; lpBaseAddress
call    ds:UnmapViewOfFile

```

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]

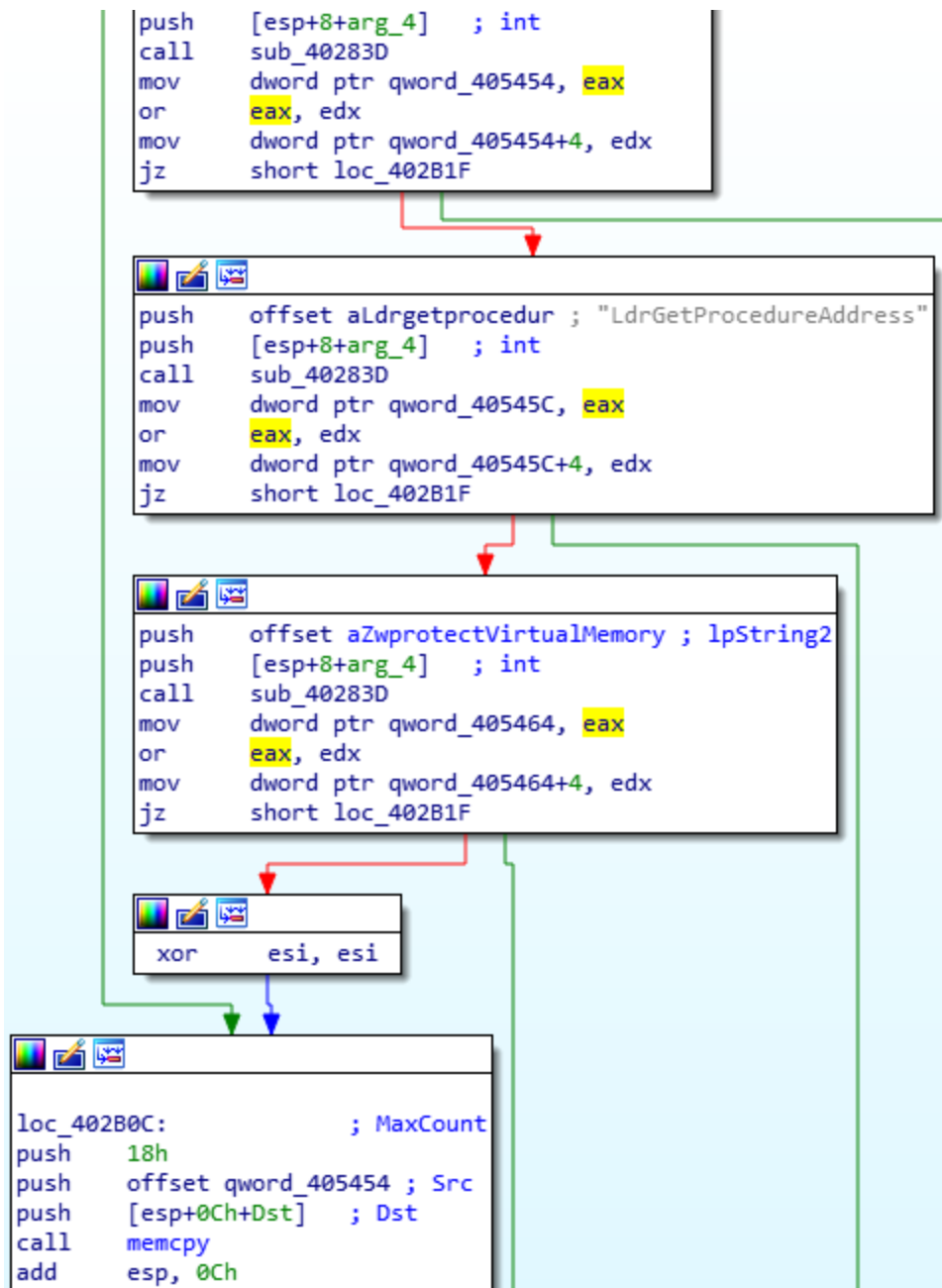
    v3 = SectionHandle;
    MaximumSize.QuadPart = (unsigned int)a1;
    ObjectAttributes.Attributes = 64;
    SectionHandle = 0;
    Dst = 0;
    ObjectAttributes.Length = 24;
    ObjectAttributes.RootDirectory = 0;
    ObjectAttributes.ObjectName = 0;
    ObjectAttributes.SecurityDescriptor = 0;
    ObjectAttributes.SecurityQualityOfService = 0;
    v4 = NtCreateSection(&SectionHandle, 0xF001Fu, &ObjectAttributes, &MaximumSize, 0x40u, 0x8000000u, 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;
}

```

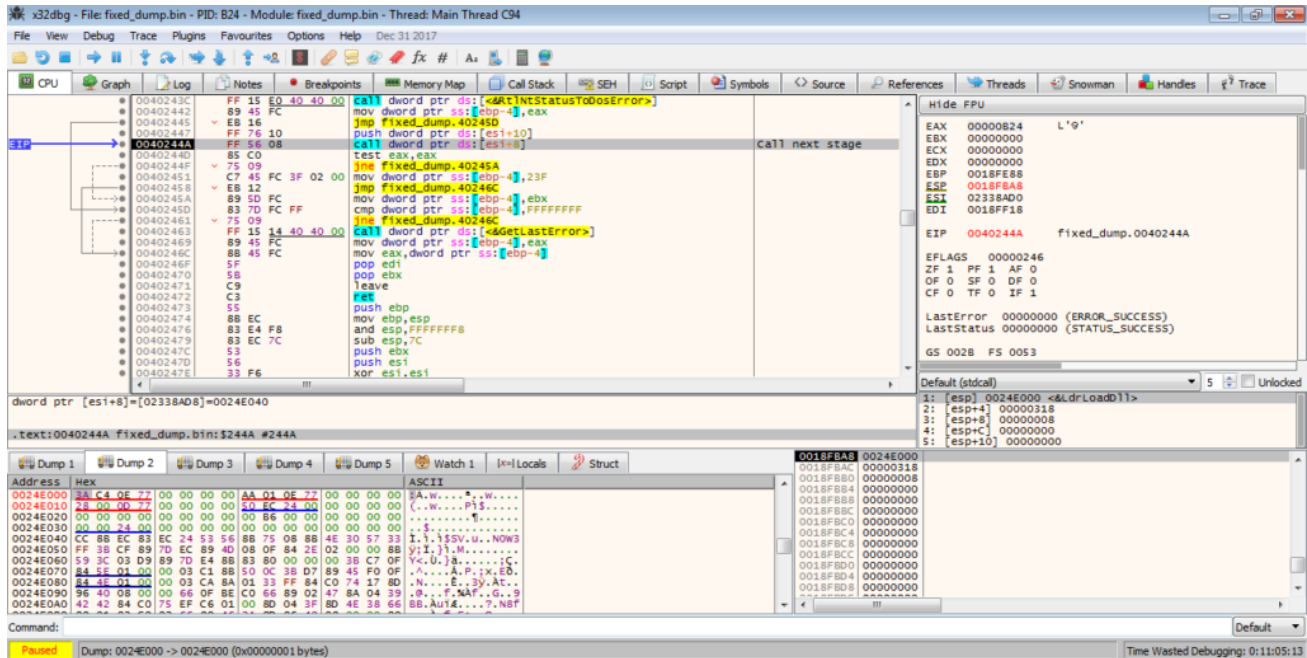
From there, an address located inside the newly created section of memory will be formed, pointing to 0x0022EC50, which will be used later. The created section of memory is replicated, to the address 0x00240000. Next, the program begins to copy over the executable to the new addresses, however it skips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

```
fixed_dump.bin (2852) (0x220000 - 0x23b000)
00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000010 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 .....
00000030 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 .....
00000050 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 .....
00000070 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 .....
00000090 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 .....
000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000d0 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 .....
000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000100 00 00 00 4c 01 05 00 10 1b 4f 5c 00 00 00 00 .....L.....0\.....
00000110 00 00 00 e0 00 02 21 0b 01 08 00 00 8a 00 00 .....!.
00000120 00 26 00 00 00 00 00 00 a5 5a 00 00 10 00 00 .....6.....Z.....
00000130 00 a0 00 00 00 00 00 10 00 10 00 00 02 00 00 .....
00000140 04 00 00 00 00 00 00 04 00 00 00 00 00 00 .....
00000150 00 e0 00 00 00 04 00 00 00 00 00 02 00 00 .....
00000160 00 00 10 00 00 10 00 00 00 10 00 00 10 00 .....
00000170 00 00 00 10 00 00 00 a0 ac 00 00 33 00 00 .....3...
00000180 00 a6 00 00 50 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 .....
000001a0 00 d0 00 00 c4 04 00 00 00 00 00 00 00 00 .....
000001b0 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 .....
000001d0 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 .....
000001f0 00 00 00 00 00 00 00 2e 74 65 78 74 00 00 .....text...
00000200 0f 88 00 00 10 00 00 00 8a 00 00 04 00 00 .....
00000210 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 .....
00000220 2e 72 64 61 74 61 00 00 d3 0c 00 00 a0 00 00 .rdata.....
00000230 00 0e 00 00 00 8e 00 00 00 00 00 00 00 00 .....
00000240 00 00 00 00 40 00 00 40 2e 64 61 74 61 00 00 ...@..@.data...
00000250 c0 02 00 00 00 b0 00 00 00 02 00 00 9c 00 00 .....
```

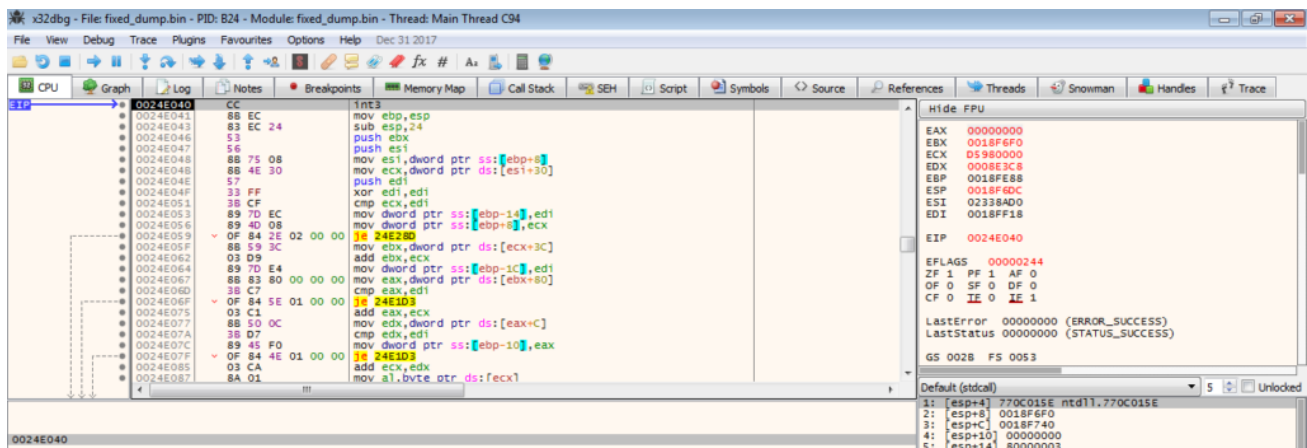
Then, a function is called that imports some more API calls from NTDLL, however this time it is not done using CRC hashes, and the name is simply passed as an argument. The imported API calls are; LdrLoadDll, LdrGetProcedureAddress, and ZwProtectVirtualMemory. The pointers to the addresses are stored 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.



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`.

Immunity Debugger - fixed_dump.bin - [CPU - main thread]

File View Debug Plugins ImmLib Options Window Help Jobs

Immunity Consulting Services Manager

```

004E077E 8F4 4E1000  MOV EDI,EDI
004E0780 8A1  MOV DL,BYTE PTR DS:[EAX]
004E0782 8A  YES EDI,EDI
004E0784 C9  MOV AL,AL
004E0786 74 17  MOV AL,AL
004E0788 09 40000000  LEA EDX,DWORD PTR DS:[ESI+040]
004E0790 668FEC0  MOV WORD PTR DS:[EDI],AX  Write DLL name to memory
004E0792 47  INC EDI
004E0794 8A439  MOV AL,BYTE PTR DS:[EAX+EDI]
004E0796 42  INC EDI
004E0798 8A  MOV AL,AL
004E079A 8A  YES EDI,EDI
004E079C 75 00  MOV BYTE PTR DS:[EDI],0
004E079E 0D43F  LEA EDX,DWORD PTR DS:[EDI+EDI]
004E07A0 0D43  LEA ECX,DWORD PTR DS:[ESI+33]
004E07A2 661941  MOV WORD PTR DS:[EDI],AX
004E07A4 8D  MOV EDI,EDI
004E07A6 66194E  MOV WORD PTR DS:[ESI+30],AX
004E07A8 661953  LEA EDI,DWORD PTR DS:[ESI+040]
004E07AA 661958  MOV WORD PTR DS:[ESI+3C],EAX
004E07AC 0D45 E4  LEA EDX,DWORD PTR SS:[EBP-1C]
004E07AE 59  PUSH EDI
004E07B0 51  PUSH ECX
004E07B2 57  PUSH EDI
004E07B4 57  PUSH EDI
004E07B6 57  PUSH EDI
004E07B8 FF16  MOV EDI,DWORD PTR DS:[ESI]  ntdll.LdrLoadDll
004E07BA 8E  MOV EDI,EDI
004E07BC 66194C  MOV WORD PTR SS:[EBP-4],EAX
004E07BE 66194E  MOV EDI,DWORD PTR SS:[EBP-10]
004E07C0 8B10  MOV EDI,DWORD PTR DS:[EAX]
004E07C2 8D  INC EDI,EDI
004E07C4 66194E  MOV WORD PTR SS:[EBP-10],EDI
004E07C6 66194E  MOV EDI,DWORD PTR DS:[EAX+10]
004E07C8 75 00  JNZ

```

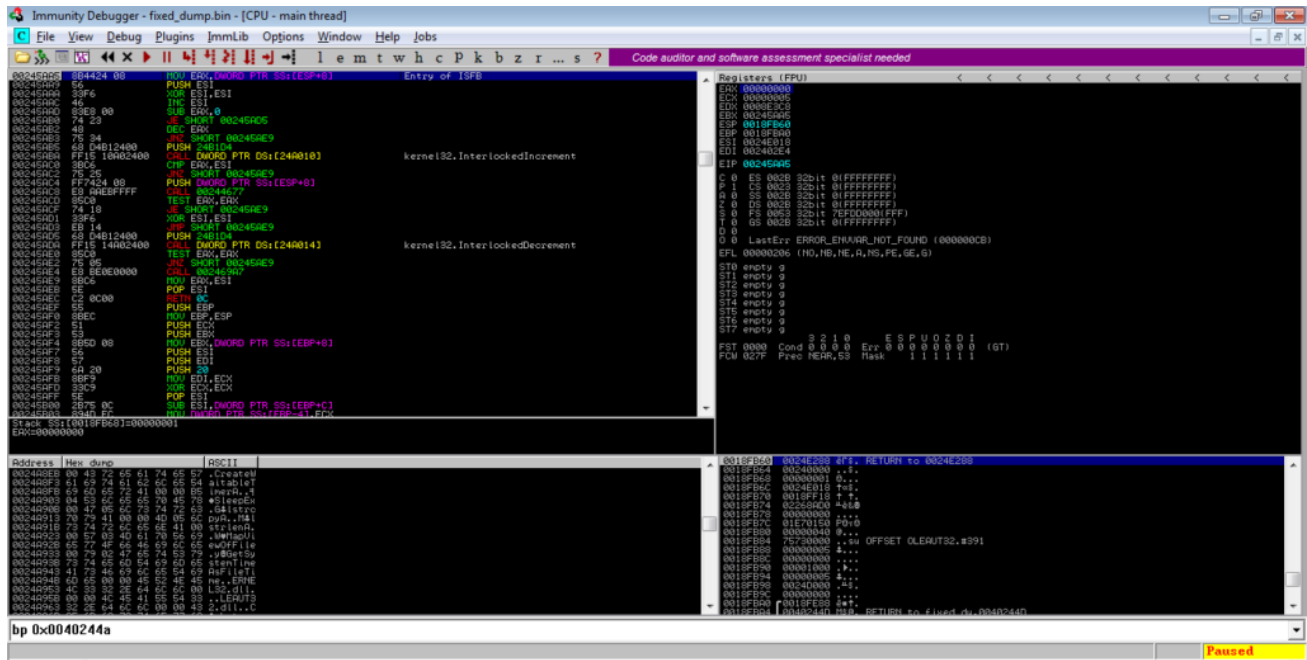
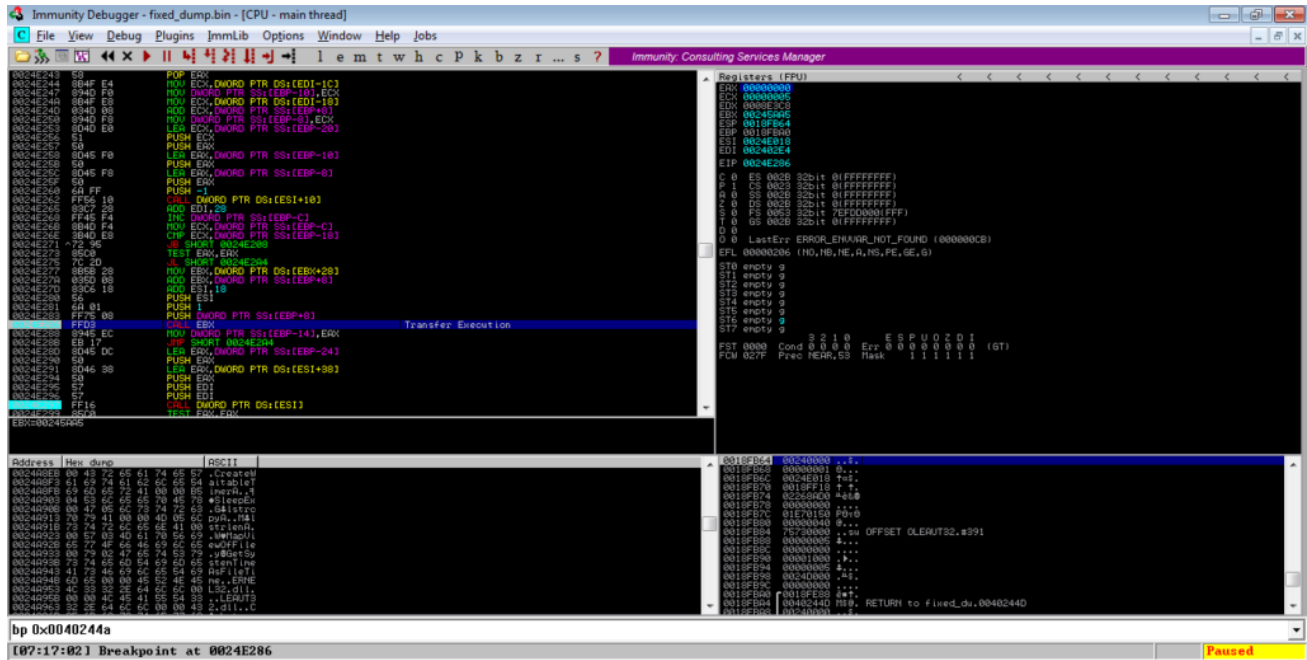
Registers (FPU)

```

EAX 001F660
ECX 0024E30
EDX 0024E30
ESI 0024100
ESP 001F660
EBP 001F660
ESI 0024E00
EDI 0000000
EIP 0024E0B
C 0 ES 002B 32bit (0FFFFFFF)
P 1 CS 002B 32bit (0FFFFFFF)
D 0 DS 002B 32bit (0FFFFFFF)
I 0 FS 002B 32bit (7E000000)
T 0 GS 002B 32bit (0FFFFFFF)
0 0
0 0 LastErr: ERROR_SUCCESS (00000000)
EFL 00000246 (NO,HB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
FPU Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FPU Prec NEAR,SS Task 1 1 1 1

```

Address	Hex dump	ASCII
004E071E	16 56 80 00 72 00 30 00	SUI+1(0)
004E0720	74 24 58 00 73 04 38 12	12+1(4)
004E0722	C1 75 EF 58 C2 CE 04	40 17 77
004E0724	80 A1 50 00 00 35	17+1(0)
004E0726	F6 95 C0 74 1E FF 6E	00 3A 4 A 8
004E0728	2C 00 64 00 00 00	00 00 00 00 00 00
004E072A	00 00 00 00 00 00	00 00 00 00 00 00
004E072C	00 00 00 00 00 00	00 00 00 00 00 00
004E072E	00 00 00 00 00 00	00 00 00 00 00 00
004E0730	00 00 00 00 00 00	00 00 00 00 00 00
004E0732	00 00 00 00 00 00	00 00 00 00 00 00
004E0734	00 00 00 00 00 00	00 00 00 00 00 00
004E0736	00 00 00 00 00 00	00 00 00 00 00 00
004E0738	00 00 00 00 00 00	00 00 00 00 00 00
004E073A	00 00 00 00 00 00	00 00 00 00 00 00
004E073C	00 00 00 00 00 00	00 00 00 00 00 00
004E073E	00 00 00 00 00 00	00 00 00 00 00 00
004E0740	00 00 00 00 00 00	00 00 00 00 00 00
004E0742	00 00 00 00 00 00	00 00 00 00 00 00
004E0744	00 00 00 00 00 00	00 00 00 00 00 00
004E0746	00 00 00 00 00 00	00 00 00 00 00 00
004E0748	00 00 00 00 00 00	00 00 00 00 00 00
004E074A	00 00 00 00 00 00	00 00 00 00 00 00
004E074C	00 00 00 00 00 00	00 00 00 00 00 00
004E074E	00 00 00 00 00 00	00 00 00 00 00 00
004E0750	00 00 00 00 00 00	00 00 00 00 00 00
004E0752	00 00 00 00 00 00	00 00 00 00 00 00
004E0754	00 00 00 00 00 00	00 00 00 00 00 00
004E0756	00 00 00 00 00 00	00 00 00 00 00 00
004E0758	00 00 00 00 00 00	00 00 00 00 00 00
004E075A	00 00 00 00 00 00	00 00 00 00 00 00
004E075C	00 00 00 00 00 00	00 00 00 00 00 00
004E075E	00 00 00 00 00 00	00 00 00 00 00 00
004E0760	00 00 00 00 00 00	00 00 00 00 00 00
004E0762	00 00 00 00 00 00	00 00 00 00 00 00
004E0764	00 00 00 00 00 00	00 00 00 00 00 00
004E0766	00 00 00 00 00 00	00 00 00 00 00 00
004E0768	00 00 00 00 00 00	00 00 00 00 00 00
004E076A	00 00 00 00 00 00	00 00 00 00 00 00
004E076C	00 00 00 00 00 00	00 00 00 00 00 00
004E076E	00 00 00 00 00 00	00 00 00 00 00 00
004E0770	00 00 00 00 00 00	00 00 00 00 00 00
004E0772	00 00 00 00 00 00	00 00 00 00 00 00
004E0774	00 00 00 00 00 00	00 00 00 00 00 00
004E0776	00 00 00 00 00 00	00 00 00 00 00 00
004E0778	00 00 00 00 00 00	00 00 00 00 00 00
004E077A	00 00 00 00 00 00	00 00 00 00 00 00
004E077C	00 00 00 00 00 00	00 00 00 00 00 00
004E077E	00 00 00 00 00 00	00 00 00 00 00 00
004E0780	00 00 00 00 00 00	00 00 00 00 00 00
004E0782	00 00 00 00 00 00	00 00 00 00 00 00
004E0784	00 00 00 00 00 00	00 00 00 00 00 00
004E0786	00 00 00 00 00 00	00 00 00 00 00 00
004E0788	00 00 00 00 00 00	00 00 00 00 00 00
004E078A	00 00 00 00 00 00	00 00 00 00 00 00
004E078C	00 00 00 00 00 00	00 00 00 00 00 00
004E078E	00 00 00 00 00 00	00 00 00 00 00 00
004E0790	00 00 00 00 00 00	00 00 00 00 00 00
004E0792	00 00 00 00 00 00	00 00 00 00 00 00
004E0794	00 00 00 00 00 00	00 00 00 00 00 00
004E0796	00 00 00 00 00 00	00 00 00 00 00 00
004E0798	00 00 00 00 00 00	00 00 00 00 00 00
004E079A	00 00 00 00 00 00	00 00 00 00 00 00
004E079C	00 00 00 00 00 00	00 00 00 00 00 00
004E079E	00 00 00 00 00 00	00 00 00 00 00 00
004E07A0	00 00 00 00 00 00	00 00 00 00 00 00
004E07A2	00 00 00 00 00 00	00 00 00 00 00 00
004E07A4	00 00 00 00 00 00	00 00 00 00 00 00
004E07A6	00 00 00 00 00 00	00 00 00 00 00 00
004E07A8	00 00 00 00 00 00	00 00 00 00 00 00
004E07AA	00 00 00 00 00 00	00 00 00 00 00 00
004E07AC	00 00 00 00 00 00	00 00 00 00 00 00
004E07AE	00 00 00 00 00 00	00 00 00 00 00 00
004E07B0	00 00 00 00 00 00	00 00 00 00 00 00
004E07B2	00 00 00 00 00 00	00 00 00 00 00 00
004E07B4	00 00 00 00 00 00	00 00 00 00 00 00
004E07B6	00 00 00 00 00 00	00 00 00 00 00 00
004E07B8	00 00 00 00 00 00	00 00 00 00 00 00
004E07BA	00 00 00 00 00 00	00 00 00 00 00 00
004E07BC	00 00 00 00 00 00	00 00 00 00 00 00
004E07BE	00 00 00 00 00 00	00 00 00 00 00 00
004E07C0	00 00 00 00 00 00	00 00 00 00 00 00
004E07C2	00 00 00 00 00 00	00 00 00 00 00 00
004E07C4	00 00 00 00 00 00	00 00 00 00 00 00
004E07C6	00 00 00 00 00 00	00 00 00 00 00 00
004E07C8	00 00 00 00 00 00	00 00 00 00 00 00
004E07CA	00 00 00 00 00 00	00 00 00 00 00 00
004E07CC	00 00 00 00 00 00	00 00 00 00 00 00
004E07CE	00 00 00 00 00 00	00 00 00 00 00 00
004E07D0	00 00 00 00 00 00	00 00 00 00 00 00
004E07D2	00 00 00 00 00 00	00 00 00 00 00 00
004E07D4	00 00 00 00 00 00	00 00 00 00 00 00
004E07D6	00 00 00 00 00 00	00 00 00 00 00 00
004E07D8	00 00 00 00 00 00	00 00 00 00 00 00
004E07DA	00 00 00 00 00 00	00 00 00 00 00 00
004E07DC	00 00 00 00 00 00	00 00 00 00 00 00
004E07DE	00 00 00 00 00 00	00 00 00 00 00 00
004E07E0	00 00 00 00 00 00	00 00 00 00 00 00
004E07E2	00 00 00 00 00 00	00 00 00 00 00 00
004E07E4	00 00 00 00 00 00	00 00 00 00 00 00
004E07E6	00 00 00 00 00 00	00 00 00 00 00 00
004E07E8	00 00 00 00 00 00	00 00 00 00 00 00
004E07EA	00 00 00 00 00 00	00 00 00 00 00 00
004E07EC	00 00 00 00 00 00	00 00 00 00 00 00
004E07EE	00 00 00 00 00 00	00 00 00 00 00 00
004E07F0	00 00 00 00 00 00	00 00 00 00 00 00
004E07F2	00 00 00 00 00 00	00 00 00 00 00 00
004E07F4	00 00 00 00 00 00	00 00 00 00 00 00
004E07F6	00 00 00 00 00 00	00 00 00 00 00 00
004E07F8	00 00 00 00 00 00	00 00 00 00 00 00
004E07FA	00 00 00 00 00 00	00 00 00 00 00 00
004E07FC	00 00 00 00 00 00	00 00 00 00 00 00
004E07FE	00 00 00 00 00 00	00 00 00 00 00 00
004E0800	00 00 00 00 00 00	00 00 00 00 00 00
004E0802	00 00 00 00 00 00	00 00 00 00 00 00
004E0804	00 00 00 00 00 00	00 00 00 00 00 00
004E0806	00 00 00 00 00 00	00 00 00 00 00 00
004E0808	00 00 00 00 00 00	00 00 00 00 00 00
004E080A	00 00 00 00 00 00	00 00 00 00 00 00
004E080C	00 00 00 00 00 00	00 00 00 00 00 00
004E080E	00 00 00 00 00 00	00 00 00 00 00 00
004E0810	00 00 00 00 00 00	00 00 00 00 00 00
004E0812	00 00 00 00 00 00	00 00 00 00 00 00
004E0814	00 00 00 00 00 00	00 00 00 00 00 00
004E0816	00 00 00 00 00 00	00 00 00 00 00 00
004E0818	00 00 00 00 00 00	00 00 00 00 00 00
004E081A	00 00 00 00 00 00	00 00 00 00 00 00
004E081C	00 00 00 00 00 00	00 00 00 00 00 00
004E081E	00 00 00 00 00 00	00 00 00 00 00 00
004E0820	00 00 00 00 00 00	00 00 00 00 00 00
004E0822	00 00 00 00 00 00	00 00 00 00 00 00
004E0824	00 00 00 00 00 00	00 00 00 00 00 00
004E0826	00 00 00 00 00 00	00 00 00 00 00 00
004E0828	00 00 00 00 00 00	00 00 00 00 00 00
004E082A	00 00 00 00 00 00	00 00 00 00 00 00
004E082C	00 00 00 00 00 00	00 00 00 00 00 00
004E082E	00 00 00 00 00 00	00 00 00 00 00 00
004E0830	00 00 00 00 00 00	00 00 00 00 00 00
004E0832	00 00 00 00 00 00	00 00 00 00 00 00
004E0834	00 00 00 00 00 00	00 00 00 00 00 00
004E0836	00 00 00 00 00 00	00 00 00 00 00 00
004E0838	00 00 00 00 00 00	00 00 00 00 00 00
004E083A	00 00 00 00 00 00	00 00 00 00 00 00
004E083C	00 00 00 00 00 00	00 00 00 00 00 00
004E083E	00 00 00 00 00 00	00 00 00 00 00 00
004E0840	00 00 00 00 00 00	00 00 00 00 00 00
004E0842	00 00 00 00 00 00	00 00 00 00 00 00
004E0844	00 00 00 00 00 00	00 00 00 00 00 00
004E0846	00 00 00 00 00 00	00 00 00 00 00 00
004E0848	00 00 00 00 00 00	00 00 00 00 00 00
004E084A	00 00 00 00 00 00	00 00 00 00 00 00
004E084C	00 00 00 00 00 00	00 00 00 00 00 00
004E084E	00 00 00 00 00 00	00 00 00 00 00 00
004E0850	00 00 00 00 00 00	00 00 00 00 00 00
004E0852	00 00 00 00 00 00	00 00 00 00 00 00
004E0854	00 00 00 00 00 00	00 00 00 00 00 00
004E0856	00 00 00 00 00 00	00 00 00 00 00 00
004E0858	00 00 00 00 00 00	00 00 00 00 00 00
004E085A	00 00 00 00 00 00	00 00 00 00 00 00
004E085C	00 00 00 00 00 00	00 00 00 00 00 00
004E085E	00 00 00 00 00 00	00 00 00 00 00 00
004E0860	00 00 00 00 00 00	00 00 00 00 00 00
004E0862	00 00 00 00 00 00	00 00 00 00 00 00
004E0864	00 00 00 00 00 00	00 00 00 00 00 00
004E0866	00 00 00 00 00 00	00 00 00 00 00 00
004E0868	00 00 00 00 00 00	00 00 00 00 00 00
004E086A	00 00 00 00 00 00	00 00 00 00 00 00
004E086C	00 00 00 00 00 00	00 00 00 00 00 00
004E086E	00 00 00 00 00 00	00 00 00 00 00 00
004E0870	00 00 00 00 00 00	00 00 00 00 00 00
004E0872	00 00 00 00 00 00	00 00 00 00 00 00
004E0874	00 00 00 00 00 00	00 00 00 00 00 00
004E0876	00 00 00 00 00 00	00 00 00 00 00 00
004E0878	00 00 00 00 00 00	00 00 00 00 00 00
004E087A	00 00 00 00 00 00	00 00 00 00 00 00
004E087C	00 00 00 00 00 00	00 00 00 00 00 00
004E087E	00 00 00 00 00 00	00 00 00 00 00 00
004E0880	00 00 00 00 00 00	00 00 00 00 00 00
004E0882	00 00 00 00 00 00	00 00 00 00 00 00
004E0884	00 00 00 00 00 00	00 00 00 00



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!

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 4D 5A 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 .....
00000020 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 .....
00000040 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 .....
00000060 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 .....
00000080 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 .....
000000A0 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 .....
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000D0 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 .....
000000F0 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 PE..L....O\....
00000110 00 00 00 00 E0 00 02 21 0B 01 08 00 00 8A 00 ....à..!.....š..
00000120 00 26 00 00 00 00 00 00 A5 5A 00 00 00 10 00 .&.....ÿZ.....
00000130 00 A0 00 00 00 00 00 10 00 10 00 00 00 02 00 . .....
00000140 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 .à.....
00000150 00 E0 00 00 00 04 00 00 00 00 00 00 02 00 00 .à.....
00000160 00 00 10 00 00 10 00 00 00 10 00 00 10 00 00 .....
00000170 00 00 00 00 10 00 00 A0 AC 00 00 33 00 00 00 ..... 3...
00000180 00 A6 00 00 50 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 .....
000001A0 00 D0 00 00 C4 04 00 00 00 00 00 00 00 00 00 .D..Ä.....
000001B0 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 .....
000001D0 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 'c..Ä.....
000001F0 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 .....text...

```

The screenshot displays the Immunity Debugger interface. At the top, a memory dump shows hexadecimal values and their decoded ASCII representation. Below the dump, the 'Imports' tab is selected, showing a table of imported functions. The table includes columns for Offset, Name, Func. Count, Bound?, OriginalFirstThunk, TimeDateStamp, Forwarder, NameRVA, and FirstThunk. The imported functions listed are ntdll.dll, KERNEL32.dll, and OLEAUT32.dll. Below the imports table, the 'Details' tab is visible, showing a table of function pointers with columns for Call via, Name, Ordinal, Original Thunk, Thunk, Forwarder, and Hint.

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	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

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
A100	ZwOpenProces...	-	A99A	A99A	-	449
A104	ZwOpenProcess	-	A98A	A98A	-	448
A108	ZwClose	-	A980	A980	-	3E0
A10C	strcpy	-	A976	A976	-	552
A110	wcstombs	-	A9B0	A9B0	-	575
A114	ZwQueryInfor...	-	A9BC	A9BC	-	46B
A118	_snprintf	-	A9D6	A9D6	-	50E
A11C	sprintf	-	A9E2	A9E2	-	54C

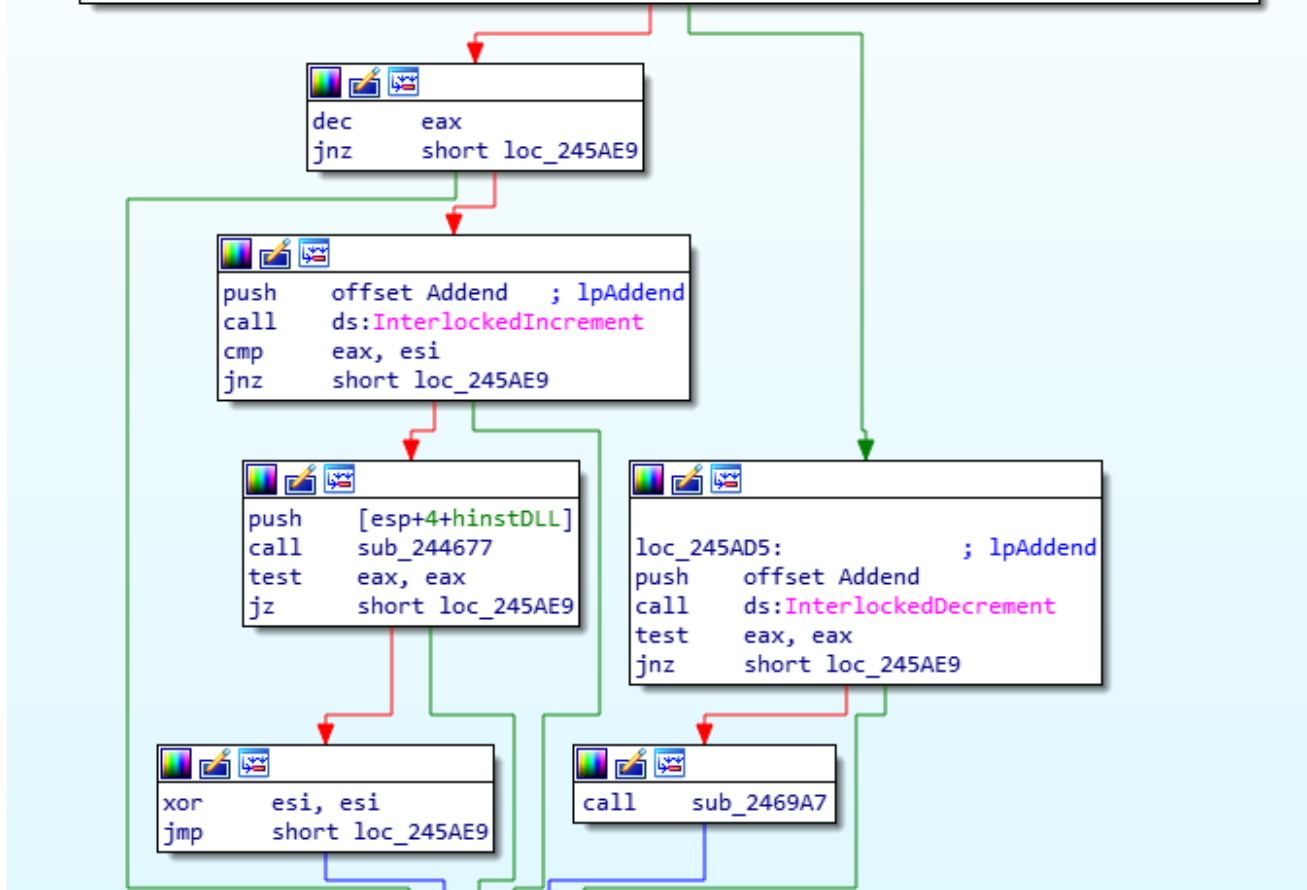

```

; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
public DllEntryPoint
DllEntryPoint proc near

hinstDLL= dword ptr 4
fdwReason= dword ptr 8
lpReserved= dword ptr 0Ch

mov     eax, [esp+fdwReason]
push   esi
xor     esi, esi
inc     esi
sub     eax, 0
jz      short loc_245AD5

```



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.