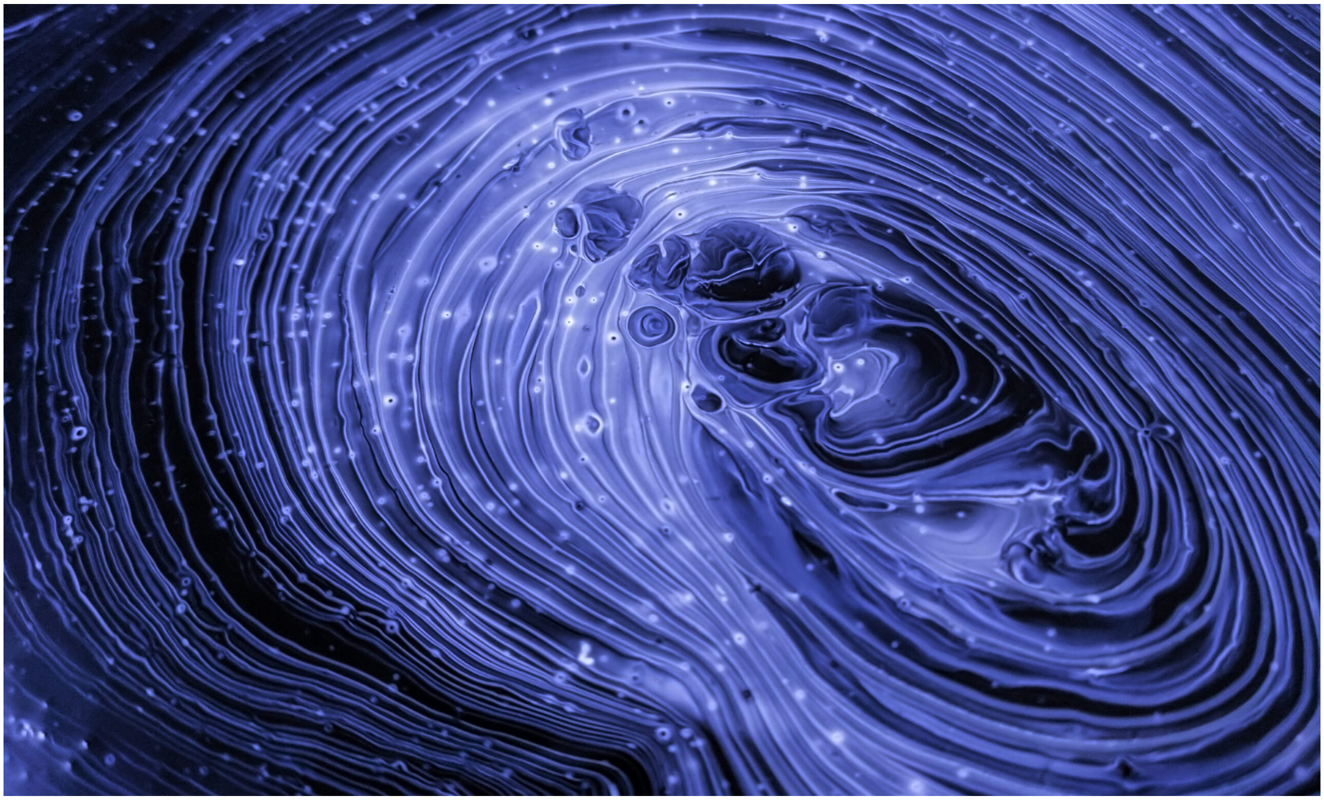
Hidden menace: Peeling back the secrets of OnionCrypter

decoded.avast.io/jakubkaloc/onion-crypter/

March 17, 2021



by Jakub KaločMarch 17, 202120 min read

One of the goals of malware authors is to keep their creation undetected by antivirus software. One possible solution for this are crypters. A crypter encrypts a program, so it looks like meaningless data and it creates an envelope for this encrypted program also called a stub. This stub looks like an innocent program, it may also perform some tasks which are not harmful at all but its primary task is to decrypt a payload and run it.

Why is this one intriguing?

The crypter discussed in this blogpost uses a combination of multiple interesting techniques that make it hard for analysts and for proper detection. One of the key techniques this crypter uses is multiple layers of encryption. Because of this we are calling it “OnionCrypter”. It’s important to note the name reflects the many layers this crypter uses, it’s in no way related to the TOR browser or network.

This blogpost covers most of the techniques OnionCrypter used to complicate analysis and breaks down its structure. This can help malware analysts because seeing samples like these might get confusing and overwhelming at first not only for humans but also for dynamic analysis sandboxes.

Most interestingly, we have found that OnionCrypter has been used by over 30 different malware families since 2016. This includes some of the best known-most prevalent families such as Ursnif, Lokibot, Zeus, AgentTesla, and Smokeloader among others. In the last three years we have protected almost 400,000 users around the world from malware protected by this crypter. Its widespread use and length of time in use make it a key malware infrastructure component. We believe that likely the authors of OnionCrypter offer it as an encrypting service. Based on the uniqueness of the first layer it is also safe to assume that authors of OnionCrypter offer the option of a unique stub file to ensure that encrypted malware will be undetectable. A service like this is frequently advertised as a **FUD** (fully undetectable) crypter.

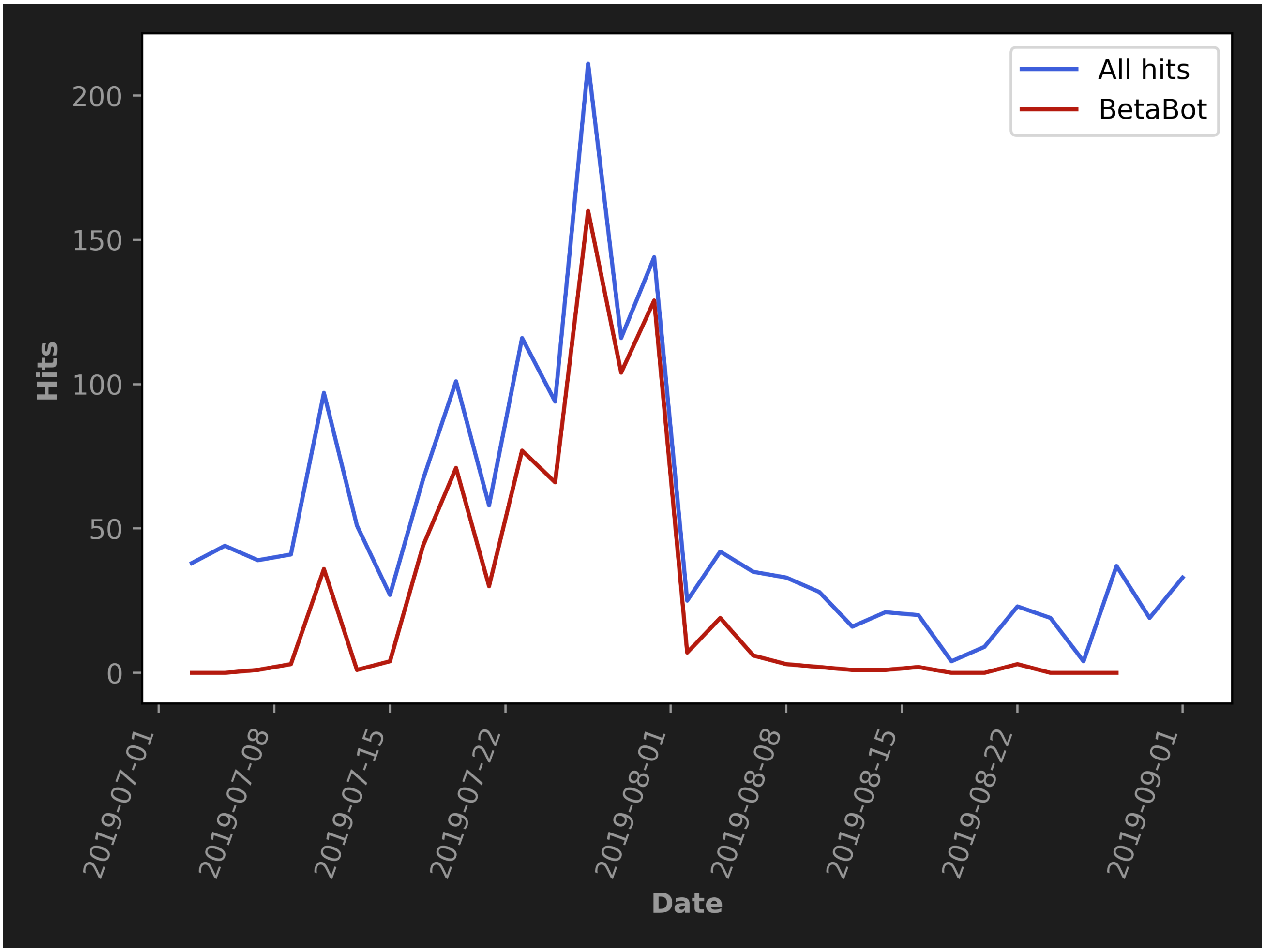
OnionCrypter forms a malware family on its own, even though it is used to protect malware from many different families. OnionCrypter has been around for several years so it is not something entirely new, however it is interesting that because of the multiple layers and uniqueness of the first layer, nobody was detecting this crypter as one malware family. After downloading thousands of samples of this crypter from VirusTotal, we were able to confirm that most of the detections from all AVs are based on detecting what’s encrypted inside this crypter. Even when AVs are recognizing the samples as a crypter with some other malware packed inside, they are detecting the samples as tens of different malware families.

Statistics

With the data from more than 15,000 samples (where oldest samples date back to 2016) it was possible to create a statistic on malware families which are using this crypter. The chart below shows that OnionCrypter is used by multiple malware authors.

Occurrence of malware families in samples
With the same data it was possible to create graphical insight on prevalence of the crypter during its existence.

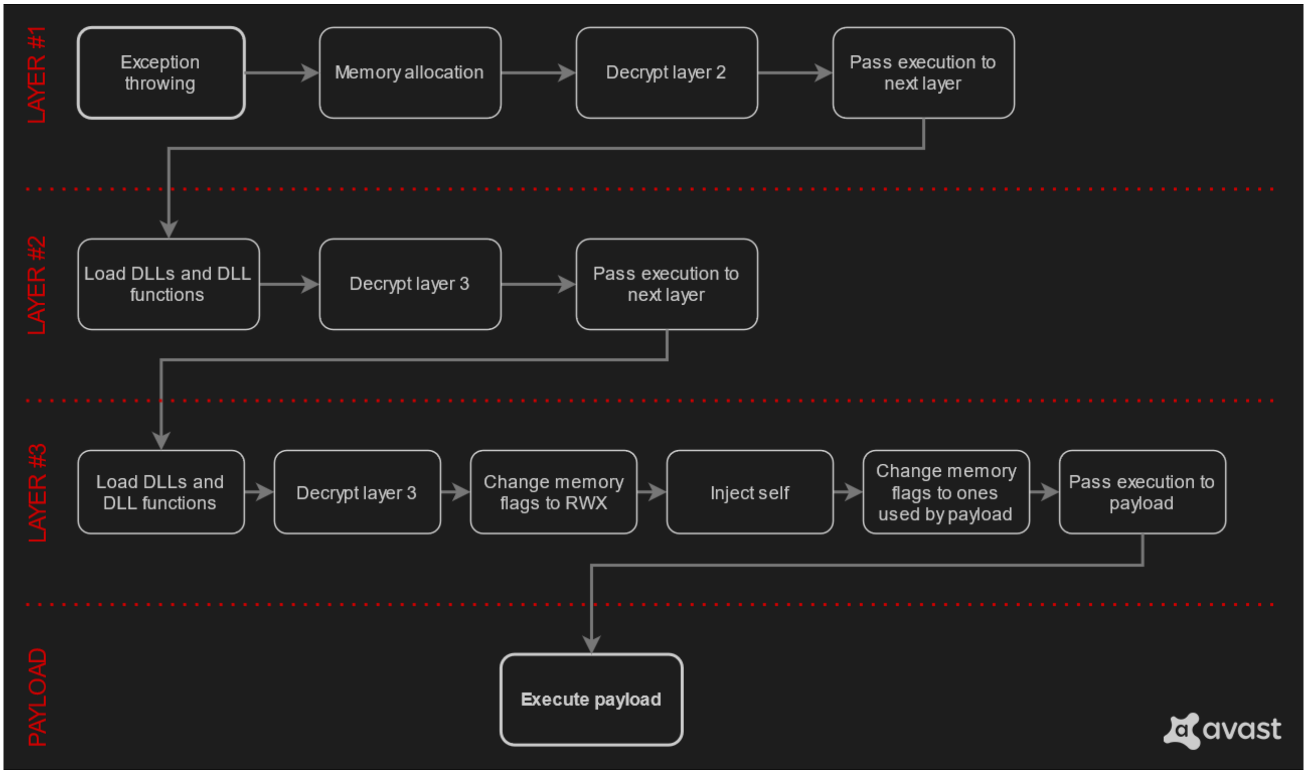
Prevalence of the OnionCrypter
This data can be further interpreted. The peaks suggest that in that time period there could have emerged a new malware campaign which was using services of the OnionCrypter and was spreading widely through the world. After a closer look at the highest peak and identification of malware families inside the OnionCrypter encrypted samples, it was possible to confirm that this peak corresponds to the spread of BetaBot malware family, a family that spreads ransomware and other malware, during the summer of 2019.



BetaBot campaign using the OnionCrypter during the summer of 2019

Analysis

OnionCrypter is 32-bit software written in C++. Architecture of OnionCrypter consists of three layers. Each layer will be discussed in a separate section along with techniques which can be found there.



OnionCrypter Program structure

Layer 1

This is the outer layer of OnionCrypter. Even though the first layer includes usually at least a few hundred functions, there is always one long function (let's call it main function) with a lot of junk code but it also includes following functionalities which are important parts of OnionCrypter:

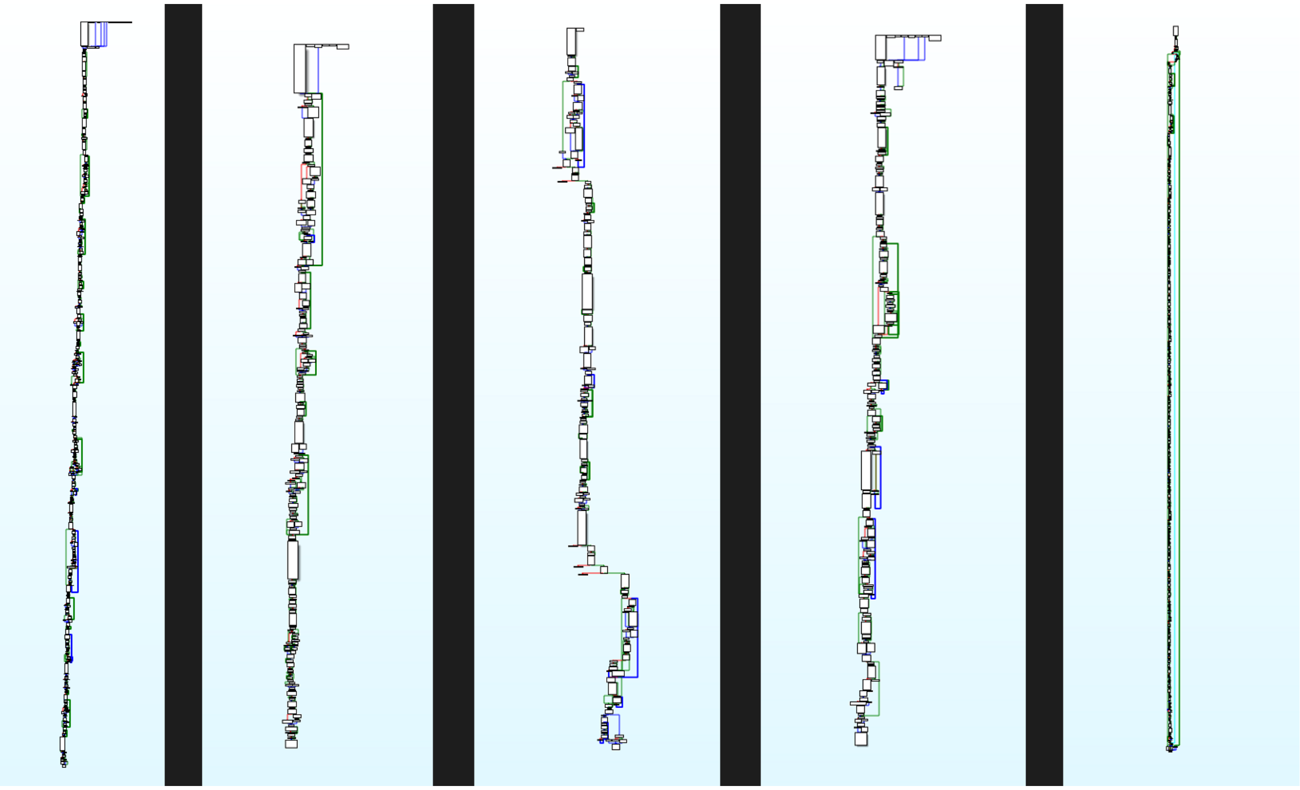
- Creation of a named event object
- Allocation of a memory
- Load data to memory
- Decrypt of the loaded data
- Pass execution to decrypted data

The easiest way to find this function is to check cross references to the `CreateEventA` API function.

Uniqueness

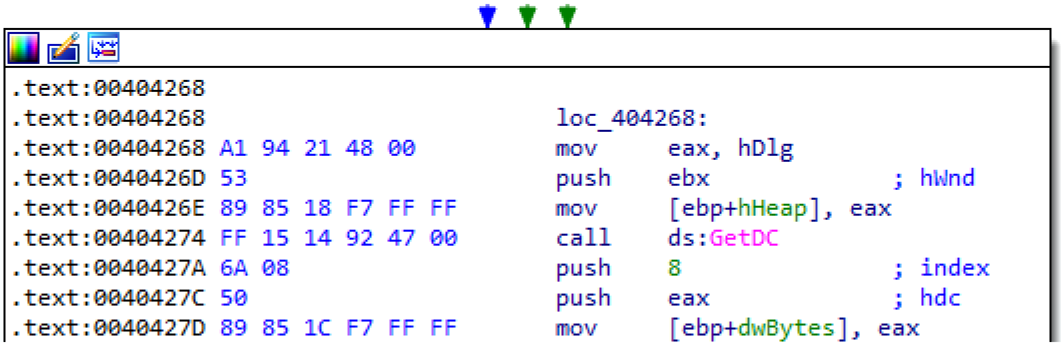
After finding this main function in multiple samples there is the first obstacle – uniqueness. Each one of the analyzed samples had a unique main function. Differences vary between big ones like completely different API function calls in the junk part of code or small ones like those that use different registers and local variables in a cycle which seem the same. As a consequence, creation of static rules for detection gets quite complicated if someone wants to cover the majority of samples.

After seeing some samples it is possible to quite easily estimate which function is the main function. The main function is always quite long, because of junk code and often because of loop unrolling. It may happen that memory allocation or decryption happens in a small part of code between unrolled iterations of loops full of junk code.

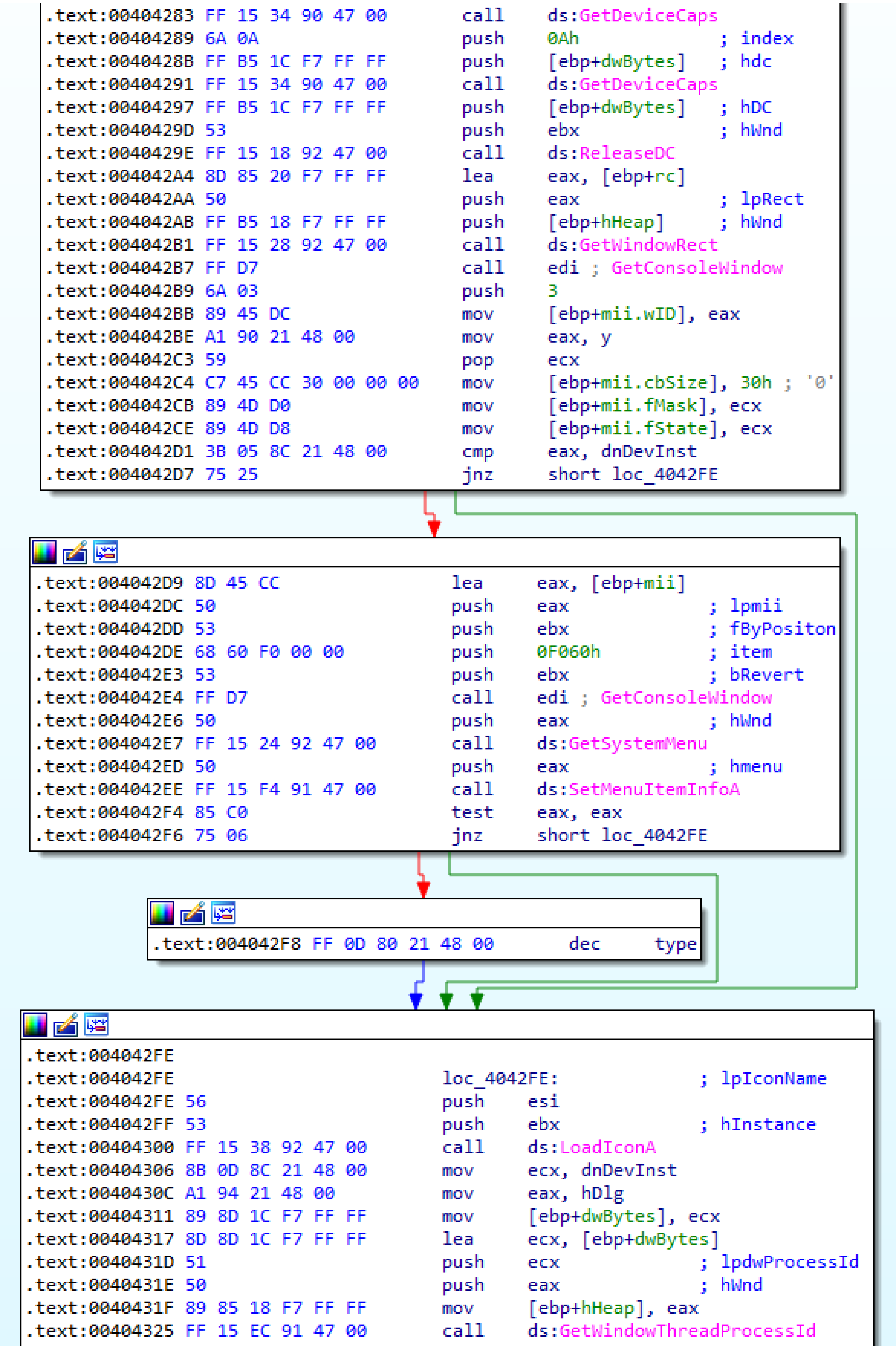


Overview of main function in IDA Pro
From left to right
260003293D1785571FEF5A2CF54E89B7AF0C1FBD5B970D2285F21BFC65E2981C
05AAB2F7D5D432CBEB970BC5471B3FAE1E45F23E0933CC673BE923F7609F53AE
17C2E36EE4387365AC00A84E91B59CE4D31D3BA04624902512810B7797A2356B
81C479BF71196724055F1AF30CA05C9162B7D32E7B3363B7F93D1AAF0161E760
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

In many cases one or more sleep calls (`sleep` function from `synchapi.h`) are included in the junk code. These sleep calls along with loops that have many iterations can increase execution time by a few minutes. This can cause some simple dynamic analysis sandboxes to fail. Even when a sandbox is able to detect the final payload and scan it with Yara rules, it is often necessary to increase timeouts to 3 or more minutes.



```
.text:00404268
.text:00404268                          loc_404268:
.text:00404268 A1 94 21 48 00           mov     eax, hDlg
.text:0040426D 53                       push    ebx             ; hWnd
.text:0040426E 89 85 18 F7 FF FF        mov     [ebp+hHeap], eax
.text:00404274 FF 15 14 92 47 00        call    ds:GetDC
.text:0040427A 6A 08                    push    8               ; index
.text:0040427C 50                       push    eax             ; hdc
.text:0040427D 89 85 1C F7 FF FF        mov     [ebp+dwBytes], eax
```



```
.text:00404283 FF 15 34 90 47 00        call    ds:GetDeviceCaps
.text:00404289 6A 0A                    push    0Ah             ; index
.text:0040428B FF B5 1C F7 FF FF        push    [ebp+dwBytes]   ; hdc
.text:00404291 FF 15 34 90 47 00        call    ds:GetDeviceCaps
.text:00404297 FF B5 1C F7 FF FF        push    [ebp+dwBytes]   ; hDC
.text:0040429D 53                       push    ebx             ; hWnd
.text:0040429E FF 15 18 92 47 00        call    ds:ReleaseDC
.text:004042A4 8D 85 20 F7 FF FF        lea     eax, [ebp+rc]
.text:004042AA 50                       push    eax             ; lpRect
.text:004042AB FF B5 18 F7 FF FF        push    [ebp+hHeap]     ; hWnd
.text:004042B1 FF 15 28 92 47 00        call    ds:GetWindowRect
.text:004042B7 FF D7                    call    edi ; GetConsoleWindow
.text:004042B9 6A 03                    push    3
.text:004042BB 89 45 DC                 mov     [ebp+mii.wID], eax
.text:004042BE A1 90 21 48 00           mov     eax, y
.text:004042C3 59                       pop     ecx
.text:004042C4 C7 45 CC 30 00 00 00     mov     [ebp+mii.cbSize], 30h ; '0'
.text:004042CB 89 4D D0                 mov     [ebp+mii.fMask], ecx
.text:004042CE 89 4D D8                 mov     [ebp+mii.fState], ecx
.text:004042D1 3B 05 8C 21 48 00        cmp     eax, dnDevInst
.text:004042D7 75 25                    jnz     short loc_4042FE

.text:004042D9 8D 45 CC                 lea     eax, [ebp+mii]
.text:004042DC 50                       push    eax             ; lpmii
.text:004042DD 53                       push    ebx             ; fByPositon
.text:004042DE 68 60 F0 00 00           push    0F060h          ; item
.text:004042E3 53                       push    ebx             ; bRevert
.text:004042E4 FF D7                    call    edi ; GetConsoleWindow
.text:004042E6 50                       push    eax             ; hWnd
.text:004042E7 FF 15 24 92 47 00        call    ds:GetSystemMenu
.text:004042ED 50                       push    eax             ; hmenu
.text:004042EE FF 15 F4 91 47 00        call    ds:SetMenuItemInfoA
.text:004042F4 85 C0                    test    eax, eax
.text:004042F6 75 06                    jnz     short loc_4042FE

.text:004042F8 FF 0D 80 21 48 00        dec     type

.text:004042FE
.text:004042FE                          loc_4042FE:             ; lpIconName
.text:004042FE 56                       push    esi
.text:004042FF 53                       push    ebx             ; hInstance
.text:00404300 FF 15 38 92 47 00        call    ds:LoadIconA
.text:00404306 8B 0D 8C 21 48 00        mov     ecx, dnDevInst
.text:0040430C A1 94 21 48 00           mov     eax, hDlg
.text:00404311 89 8D 1C F7 FF FF        mov     [ebp+dwBytes], ecx
.text:00404317 8D 8D 1C F7 FF FF        lea     ecx, [ebp+dwBytes]
.text:0040431D 51                       push    ecx             ; lpdwProcessId
.text:0040431E 50                       push    eax             ; hWnd
.text:0040431F 89 85 18 F7 FF FF        mov     [ebp+hHeap], eax
.text:00404325 FF 15 EC 91 47 00        call    ds:GetWindowThreadProcessId
```

Example of junk code in IDA Pro

8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

UPX impostors

One of the most common packers is the UPX packer which can compress programs and also hide their original code. A few samples have the first layer modified to look like they are UPX packed even when they are not. At the first glance it is possible to see that the sample has sections exactly like UPX, even when you analyze the sample with tools like “Detect It Easy”, the tool will incorrectly tell you that the sample is UPX packed.

This can lead to the confusion of an inexperienced analyst, but what is even worse it can confuse analytical tools. There are multiple tools for automatic and static unpack of UPX packed programs and for extraction of original code for further analysis. When a tool like this unpacks an UPX impostor sample the result will be random corrupted data. On data like this any static detection will not be possible and a corrupted sample won’t run in dynamic analytical boxes.

Exceptions

The majority of samples raise exceptions during debugging. In most cases it happens at the beginning of the main function. Dealing with these exceptions can slow down manual analysis and definitely make dynamic analysis more difficult. It’s a good idea to identify the place where exceptions are raised, because even if some samples are throwing only a few exceptions, others do it in a loop and passing them one by one may be too time consuming.

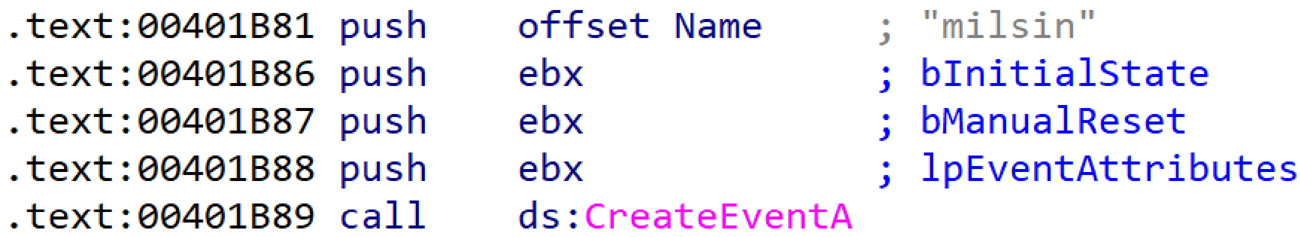
The most common exceptions which could appear are:

- Microsoft C++ exception with code `0xE06D7363`
  This exception is usually thrown by some exotic functions used in junk code.
  Some of the functions causing this exception are:
    - `SCardEstablishContext`
    - `SCardConnectA`
    - `SCardTransmit`
- Instruction referenced memory at `XYZ`. Memory could not be read. Exception code `0xC0000005`
- Unknown exception code `0x6EF`
  From function `GetServiceDisplayNameA`

We have also found that OnionCrypter combines functions that throw exceptions with the data about the position of the mouse cursor. OnionCrypter uses a loop where it finds out the cursor position (**X** and **Y** coordinates) using the function `GetCursorPos` and compares it with the position values from the previous iteration of the loop. If the **X** or **Y** coordinate didn’t change, the program calls more functions that throw the exceptions, waits for a few seconds and starts the next iteration of the loop. It is expected from a normal user that he will move his mouse during this timeframe, but it is not expected from a sandbox or analyst who is pressing the `F9` key repeatedly to pass the throwing exception part of the program. Because of that we believe that throwing the exceptions is an anti-debug trick to make the manual work of analysts harder.

Named event object

OnionCrypter uses named event objects, which are hardcoded into the code and created in the main function to avoid multiple executions of the payload. This feature is important for the malware hidden inside, because many times can multiple simultaneous executions of particular malware on one device cause some unexpected or unwanted behavior (e.g. there is no need to run the same ransomware twice on one device). After deeper analysis it was possible to connect multiple event objects to this particular software.



```
.text:00401B81 push    offset Name     ; "milsin"
.text:00401B86 push    ebx             ; bInitialState
.text:00401B87 push    ebx             ; bManualReset
.text:00401B88 push    ebx             ; lpEventAttributes
.text:00401B89 call    ds:CreateEventA
```

Creation of named event object
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE
To facilitate extraction of new names of the event object and to automate processing, an IDAPython script was created. Among most common names of event objects are:

- milsin
- svet
- lifecicled
- parames
- cueevn
- Strolls
- Menulapkievent
- doroga

Allocation of memory

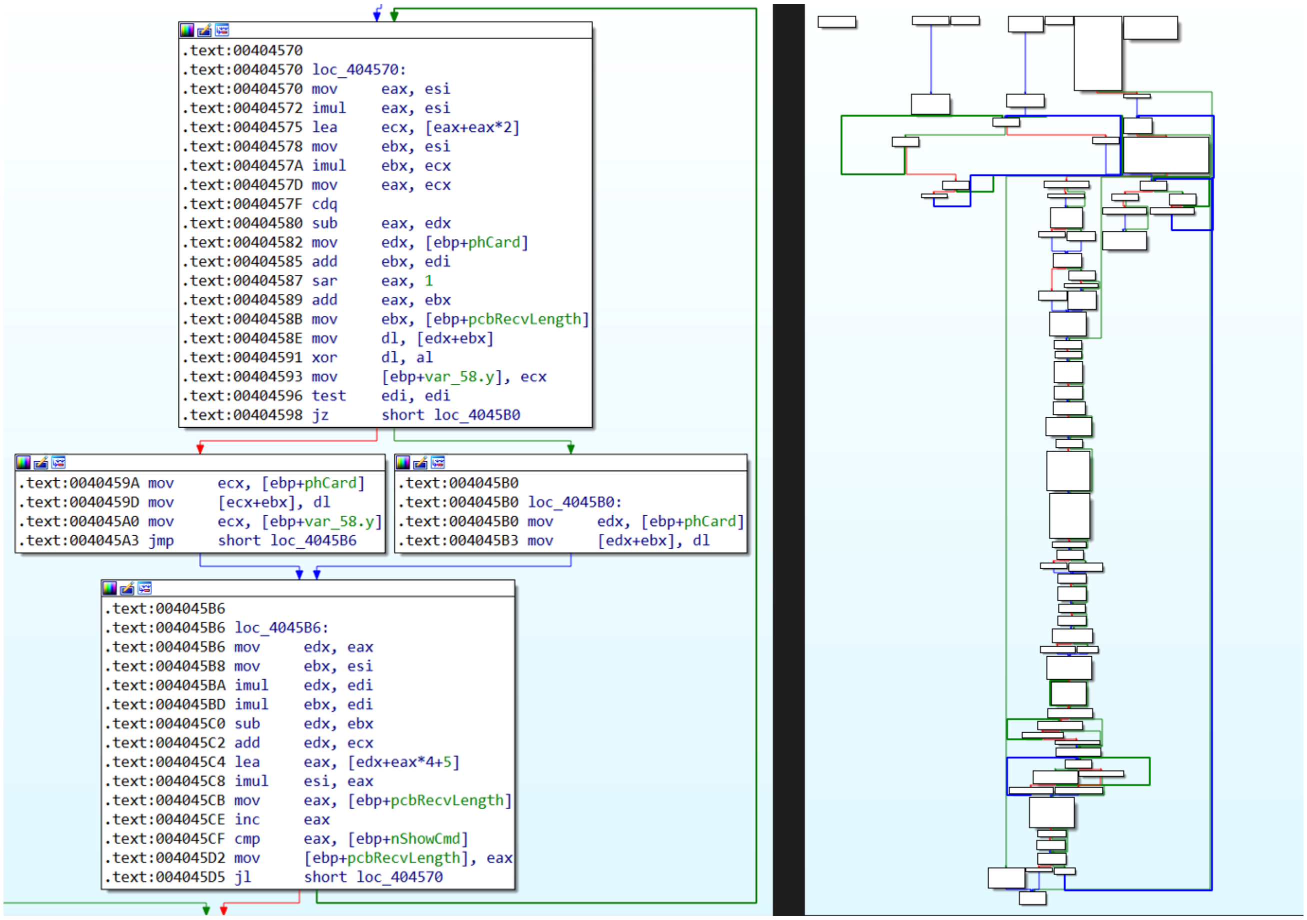
At some point during the execution of the main function OnionCrypter has to create the memory space where it loads and decrypts data. Another aspect of uniqueness is demonstrated here. For allocation OnionCrypter uses one of the following functions:

1. `GlobalAlloc`
2. `VirtualAlloc`
3. `HeapAlloc`

In other malware families it is normal that samples of a crypter belonging to the same family use the same memory allocation function across all samples. In this case there are three different functions. This complicates analysis and it is another anti-analysis trick to hide the payload, because it is not enough to hook one function and monitor allocated memory in order to find the payload. What is even worse, hooking all these functions may be a very slow way to find allocated memory, because the important allocation happens in some part of the junk code. At the same time, during execution of the junk code, allocation functions may be called many times to allocate insignificant memory. Especially when these functions are used in a loop, monitoring all allocated places will be overwhelming. One possible solution to solve this is the knowledge that the allocated memory for the encrypted data has all three of the `read/write/execute` flags set to `true`. With some cleverly placed breakpoints in main function and monitoring of memory segments it is possible to find a moment when a segment with `read/write/execute` flags was created.

Decryption of the second layer

After memory allocation, data is moved into created space and decrypted. Either a decrypt loop is implemented inline in the main function or a separate function is called. Finding the decrypt loop is easy with an `R/W` breakpoint for allocated memory. Even here every sample is quite unique. Even though all samples read data byte by byte and xor it with another value, implementation of the decrypt algorithm is totally different, as can be seen in the images below.



```
.text:00404570
.text:00404570 loc_404570:
.text:00404570 mov     eax, esi
.text:00404572 imul    eax, esi
.text:00404575 lea     ecx, [eax+eax*2]
.text:00404578 mov     ebx, esi
.text:0040457A imul    ebx, ecx
.text:0040457D mov     eax, ecx
.text:0040457F cdq
.text:00404580 sub     eax, edx
.text:00404582 mov     edx, [ebp+phCard]
.text:00404585 add     ebx, edi
.text:00404587 sar     eax, 1
.text:00404589 add     eax, ebx
.text:0040458B mov     ebx, [ebp+pcbRecvLength]
.text:0040458E mov     dl, [edx+ebx]
.text:00404591 xor     dl, al
.text:00404593 mov     [ebp+var_58.y], ecx
.text:00404596 test    edi, edi
.text:00404598 jz      short loc_4045B0
```

```
.text:0040459A mov     ecx, [ebp+phCard]
.text:0040459D mov     [ecx+ebx], dl
.text:004045A0 mov     ecx, [ebp+var_58.y]
.text:004045A3 jmp     short loc_4045B6
```

```
.text:004045B0
.text:004045B0 loc_4045B0:
.text:004045B0 mov     edx, [ebp+phCard]
.text:004045B3 mov     [edx+ebx], dl
```

```
.text:004045B6
.text:004045B6 loc_4045B6:
.text:004045B6 mov     edx, eax
.text:004045B8 mov     ebx, esi
.text:004045BA imul    edx, edi
.text:004045BD imul    ebx, edi
.text:004045C0 sub     edx, ebx
.text:004045C2 add     edx, ecx
.text:004045C4 lea     eax, [edx+eax*4+5]
.text:004045C8 imul    esi, eax
.text:004045CB mov     eax, [ebp+pcbRecvLength]
.text:004045CE inc     eax
.text:004045CF cmp     eax, [ebp+nShowCmd]
.text:004045D2 mov     [ebp+pcbRecvLength], eax
.text:004045D5 jl      short loc_404570
```

Structure of decrypt loop in IDA Pro
left – 75E692519607C2E58A3E4F5606D17262D4387D8EEA92FAB9C11C64C4A6035FBC
right – 8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

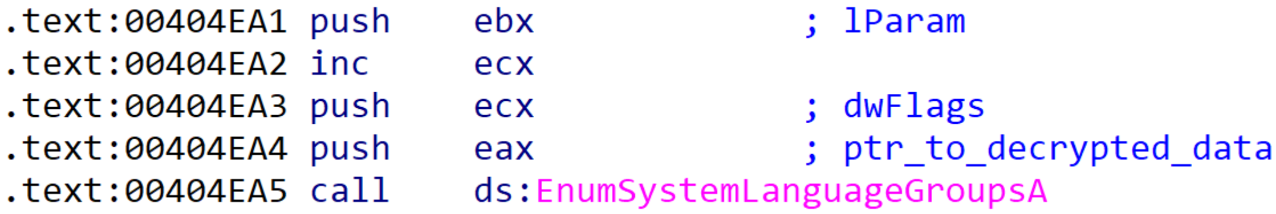
On the left side the decrypt algorithm of layer 2 is implemented as a part of the main function. This algorithm is quite simple – it uses one byte as a key value and does XOR operation on all bytes of encrypted data. What is even more interesting, this algorithm is so naive, that if the key was originally set to zero, layer 2 would not be (de/en)crypted at all.

On the other hand the decrypt algorithm on the right side is quite complicated. It is a standalone function, which receives as parameters pointer to the encrypted data, length of the encrypted data and key seed value. Decryption goes from the beginning of the encrypted data and it does XOR operation of key value and each encrypted byte. Unlike the previous decrypt algorithm, this one is a stream cipher, which generates a key stream. Key stream consists of key values where a new key value is generated from a key value used in the previous iteration.

Passing execution to the second layer

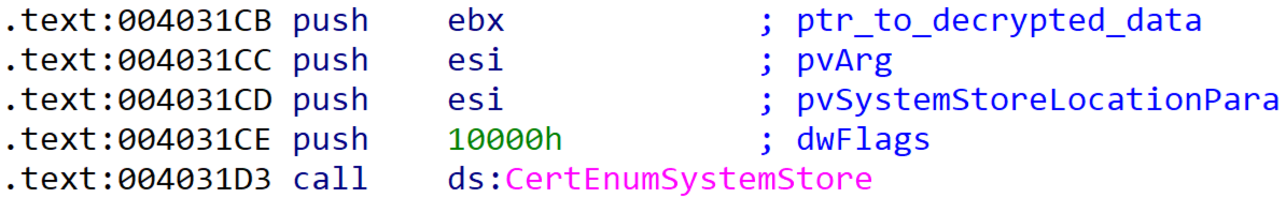
Even here are some creative ways of how to start the execution of the decrypted code. The simplest, which is also the most frequent one, is to load a pointer to the decrypted code into the register and call it.

Things can get more interesting when there is no call to a register. Some samples use “**Enum**” functions like `EnumSystemLanguageGroupsA` to pass execution. Originally this function enumerates the language groups that are either installed on or supported by an operating system, but one of the parameters of this function is a pointer to an application-defined callback function. This callback function should process the enumerated language group information provided by the `EnumSystemLanguageGroupsA` function. Instead of providing a pointer to the callback function a pointer to the decrypted code is given as parameter and as a result decrypted code gets executed.



```
.text:00404EA1 push    ebx             ; lParam
.text:00404EA2 inc     ecx
.text:00404EA3 push    ecx             ; dwFlags
.text:00404EA4 push    eax             ; ptr_to_decrypted_data
.text:00404EA5 call    ds:EnumSystemLanguageGroupsA
```

Passing execution to second layer
909A94BCB5C0354D85B8BDB64D4EE49093CCA070653F73B99C201136B72CB94A
A similar technique is used with all kinds of “**Enum**” functions e.g. `CertEnumSystemStore` or `EnumDisplayMonitors`. Because of the amount of these functions and possibility of their legitimate use, it is not feasible to detect OnionCrypter by this technique.



```
.text:004031CB push    ebx             ; ptr_to_decrypted_data
.text:004031CC push    esi             ; pvArg
.text:004031CD push    esi             ; pvSystemStoreLocationPara
.text:004031CE push    10000h          ; dwFlags
.text:004031D3 call    ds:CertEnumSystemStore
```

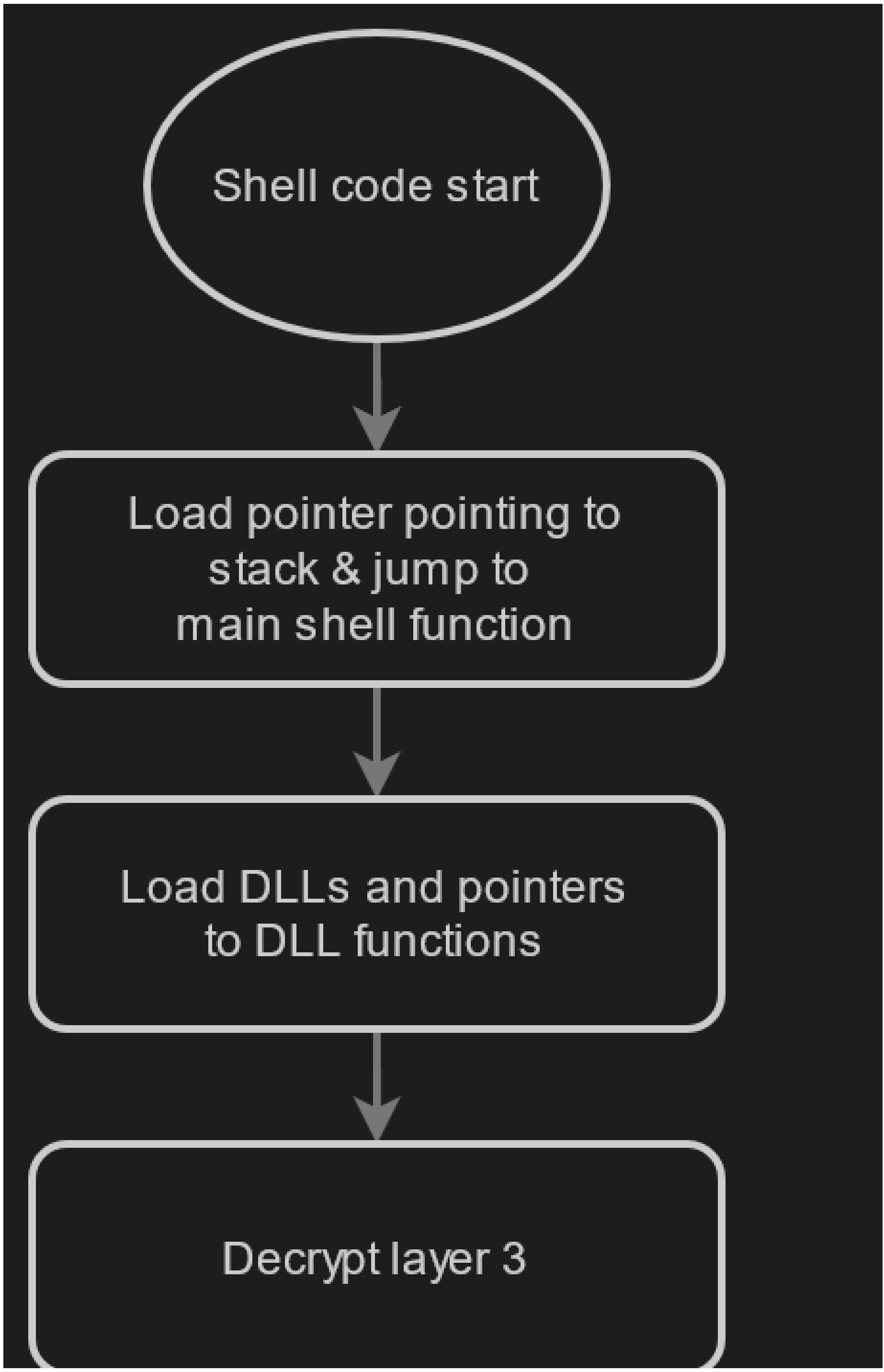
Passing execution to second layer no.2
846DCC9BCDC5C6103B2979FF93F4E1789B63827413B2FE56B1362129DF069DAF
List of functions known to be used by OnionCrypter:

- `EnumSystemLanguageGroupsA`
- `CertEnumSystemStore`
- `EnumDisplayMonitors`
- `EnumObjects`
- `EnumFontFamiliesA`
- `EnumTimeFormatsA`
- `EnumDesktopsA`
- `EnumerateLoadedModules`
- `EnumDateFormatsA`
- `EnumPropsA`
- `EnumFontsA`
- `EnumSystemGeoID`
- `EnumWindowStationsW`
- `EnumResourceTypesA`
- `acmFormatEnumA`

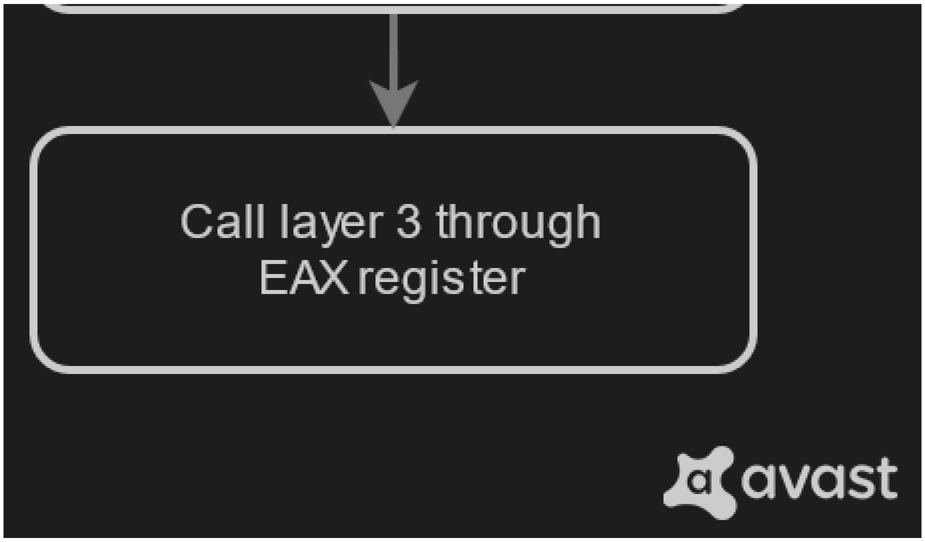
- `EnumSystemCodePagesW`

Layer 2

Layer 2 is a shell code whose ultimate task is to decrypt another layer. This process is not straightforward at all. The overview of what happens on layer 2 can be seen on image below, but the “Decrypt layer 3” bubble hides quite a complicated process of decryption. The layer 3 is decrypted in parts, but the decryption happens on another sublayer of the layer 2, in shell codes. As if it’s not enough, even these shell codes are decrypted in small parts and then put together to form a decrypt sequence.



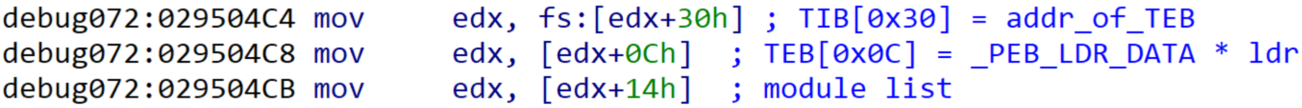
Main structure of layer 2



shell code

Finding DLLs and functions

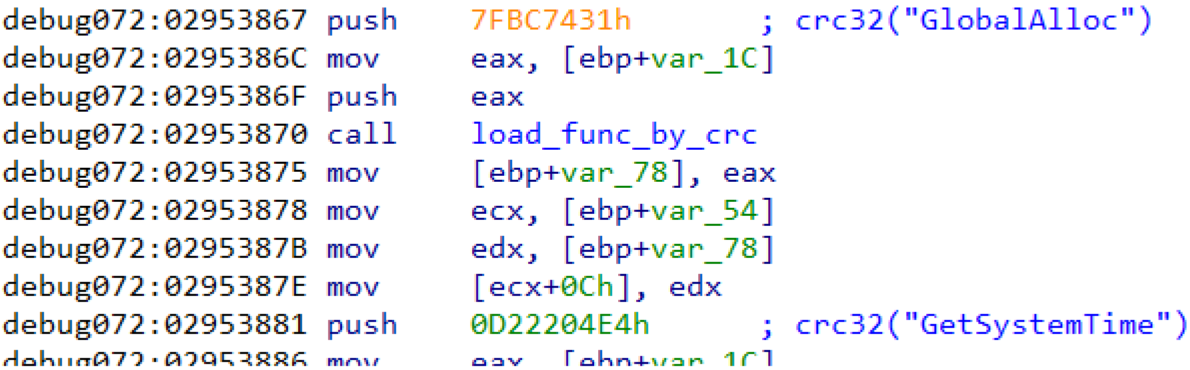
As a first thing, OnionCrypter loads pointers to `kernel32.dll`. It uses `TIB` (Thread Information Block) to find the Process Information Block and there is a pointer to a structure (`PEB_LDR_DATA`) that contains information about all of the loaded modules in the current process. By searching this structure, OnionCrypter finds the base address of `kernel32.dll`.



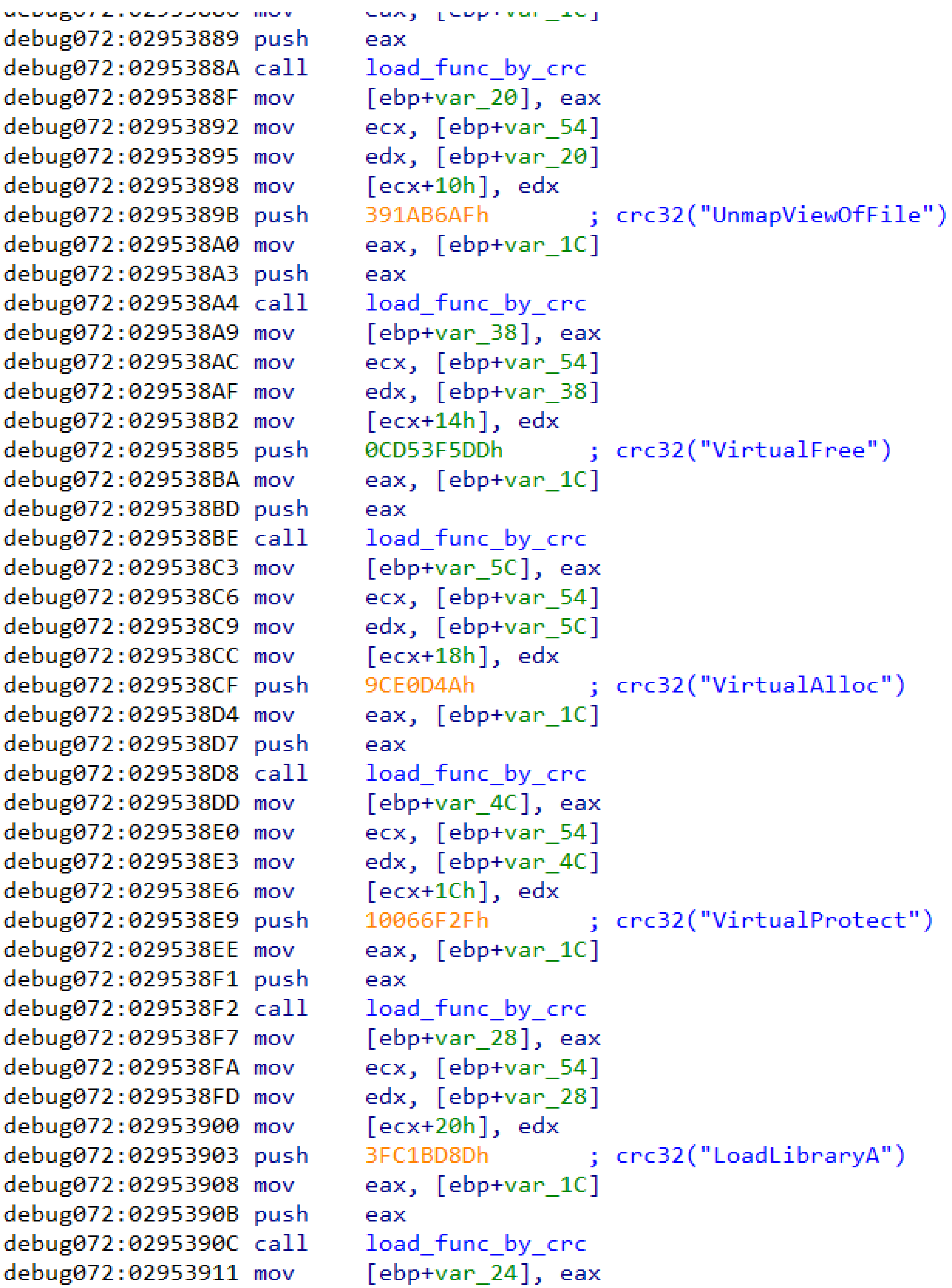
```
debug072:029504C4 mov     edx, fs:[edx+30h] ; TIB[0x30] = addr_of_TEB
debug072:029504C8 mov     edx, [edx+0Ch]  ; TEB[0x0C] = _PEB_LDR_DATA * ldr
debug072:029504CB mov     edx, [edx+14h]  ; module list
```

Loading list of modules
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

When OnionCrypter has the base address of `kernell32.dll`, it loads the address of the Export Table, which is well known. Then OnionCrypter iterates through the Name Pointer Table containing names of DLL functions. OnionCrypter calculates the `CRC32` from every function name and compares that number to one received as a hard-coded parameter. When there is a match, an iterator value is used to find the function's ordinal number in the Ordinal Table. With this number it is possible to look up the function's address in the Export Address Table. Even if this method is known, OnionCrypter tries to hide what it's loading by using pre-calculated `CRC32` numbers instead of strings with function names.



```
debug072:02953867 push    7FBC7431h       ; crc32("GlobalAlloc")
debug072:0295386C mov     eax, [ebp+var_1C]
debug072:0295386F push    eax
debug072:02953870 call    load_func_by_crc
debug072:02953875 mov     [ebp+var_78], eax
debug072:02953878 mov     ecx, [ebp+var_54]
debug072:0295387B mov     edx, [ebp+var_78]
debug072:0295387E mov     [ecx+0Ch], edx
debug072:02953881 push    0D22204E4h      ; crc32("GetSystemTime")
debug072:02953886 mov     eax, [ebp+var_1C]
```

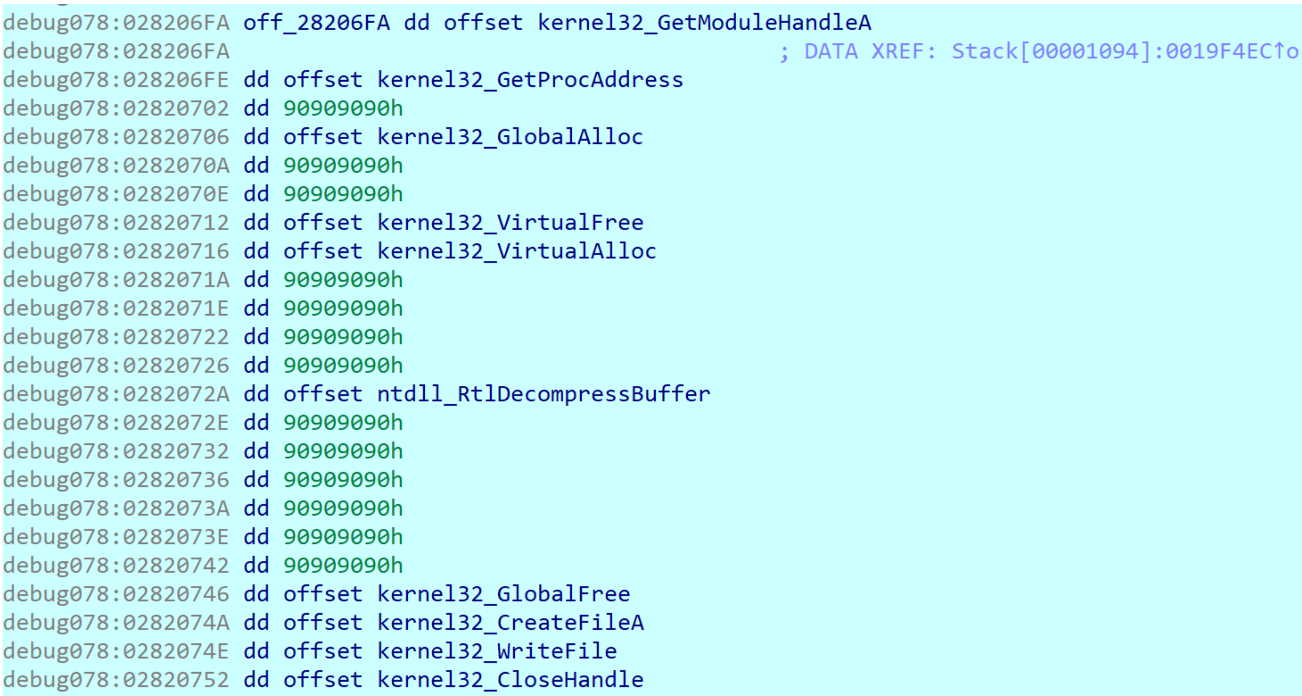


```
debug072:02953889 push    eax
debug072:0295388A call    load_func_by_crc
debug072:0295388F mov     [ebp+var_20], eax
debug072:02953892 mov     ecx, [ebp+var_54]
debug072:02953895 mov     edx, [ebp+var_20]
debug072:02953898 mov     [ecx+10h], edx
debug072:0295389B push    391AB6AFh       ; crc32("UnmapViewOfFile")
debug072:029538A0 mov     eax, [ebp+var_1C]
debug072:029538A3 push    eax
debug072:029538A4 call    load_func_by_crc
debug072:029538A9 mov     [ebp+var_38], eax
debug072:029538AC mov     ecx, [ebp+var_54]
debug072:029538AF mov     edx, [ebp+var_38]
debug072:029538B2 mov     [ecx+14h], edx
debug072:029538B5 push    0CD53F5DDh      ; crc32("VirtualFree")
debug072:029538BA mov     eax, [ebp+var_1C]
debug072:029538BD push    eax
debug072:029538BE call    load_func_by_crc
debug072:029538C3 mov     [ebp+var_5C], eax
debug072:029538C6 mov     ecx, [ebp+var_54]
debug072:029538C9 mov     edx, [ebp+var_5C]
debug072:029538CC mov     [ecx+18h], edx
debug072:029538CF push    9CE0D4Ah        ; crc32("VirtualAlloc")
debug072:029538D4 mov     eax, [ebp+var_1C]
debug072:029538D7 push    eax
debug072:029538D8 call    load_func_by_crc
debug072:029538DD mov     [ebp+var_4C], eax
debug072:029538E0 mov     ecx, [ebp+var_54]
debug072:029538E3 mov     edx, [ebp+var_4C]
debug072:029538E6 mov     [ecx+1Ch], edx
debug072:029538E9 push    10066F2Fh       ; crc32("VirtualProtect")
debug072:029538EE mov     eax, [ebp+var_1C]
debug072:029538F1 push    eax
debug072:029538F2 call    load_func_by_crc
debug072:029538F7 mov     [ebp+var_28], eax
debug072:029538FA mov     ecx, [ebp+var_54]
debug072:029538FD mov     edx, [ebp+var_28]
debug072:02953900 mov     [ecx+20h], edx
debug072:02953903 push    3FC1BD8Dh       ; crc32("LoadLibraryA")
debug072:02953908 mov     eax, [ebp+var_1C]
debug072:0295390B push    eax
debug072:0295390C call    load_func_by_crc
debug072:02953911 mov     [ebp+var_24], eax
```

Example of loading pointers to DLL functions by CRC32 of their name

8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

As a first function, OnionCrypter loads `GetModuleHandleA`. With this function it can then load `advapi32.dll` and `ntdll.dll`. In the next steps the program loads multiple functions from DLLs and stores them in the same memory space, where shell code is running. Fixed storage is created for that.



```
debug078:028206FA off_28206FA dd offset kernel32_GetModuleHandleA
debug078:028206FA                                         ; DATA XREF: Stack[00001094]:0019F4EC↑o
debug078:028206FE dd offset kernel32_GetProcAddress
debug078:02820702 dd 90909090h
debug078:02820706 dd offset kernel32_GlobalAlloc
debug078:0282070A dd 90909090h
debug078:0282070E dd 90909090h
debug078:02820712 dd offset kernel32_VirtualFree
debug078:02820716 dd offset kernel32_VirtualAlloc
debug078:0282071A dd 90909090h
debug078:0282071E dd 90909090h
debug078:02820722 dd 90909090h
debug078:02820726 dd 90909090h
debug078:0282072A dd offset ntdll_RtlDecompressBuffer
debug078:0282072E dd 90909090h
debug078:02820732 dd 90909090h
debug078:02820736 dd 90909090h
debug078:0282073A dd 90909090h
debug078:0282073E dd 90909090h
debug078:02820742 dd 90909090h
debug078:02820746 dd offset kernel32_GlobalFree
debug078:0282074A dd offset kernel32_CreateFileA
debug078:0282074E dd offset kernel32_WriteFile
debug078:02820752 dd offset kernel32_CloseHandle
```

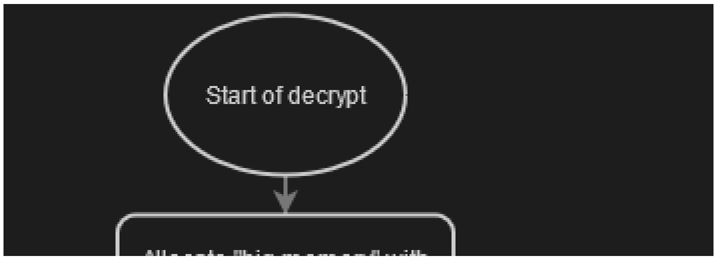
Storage of loaded functions inside shell code
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

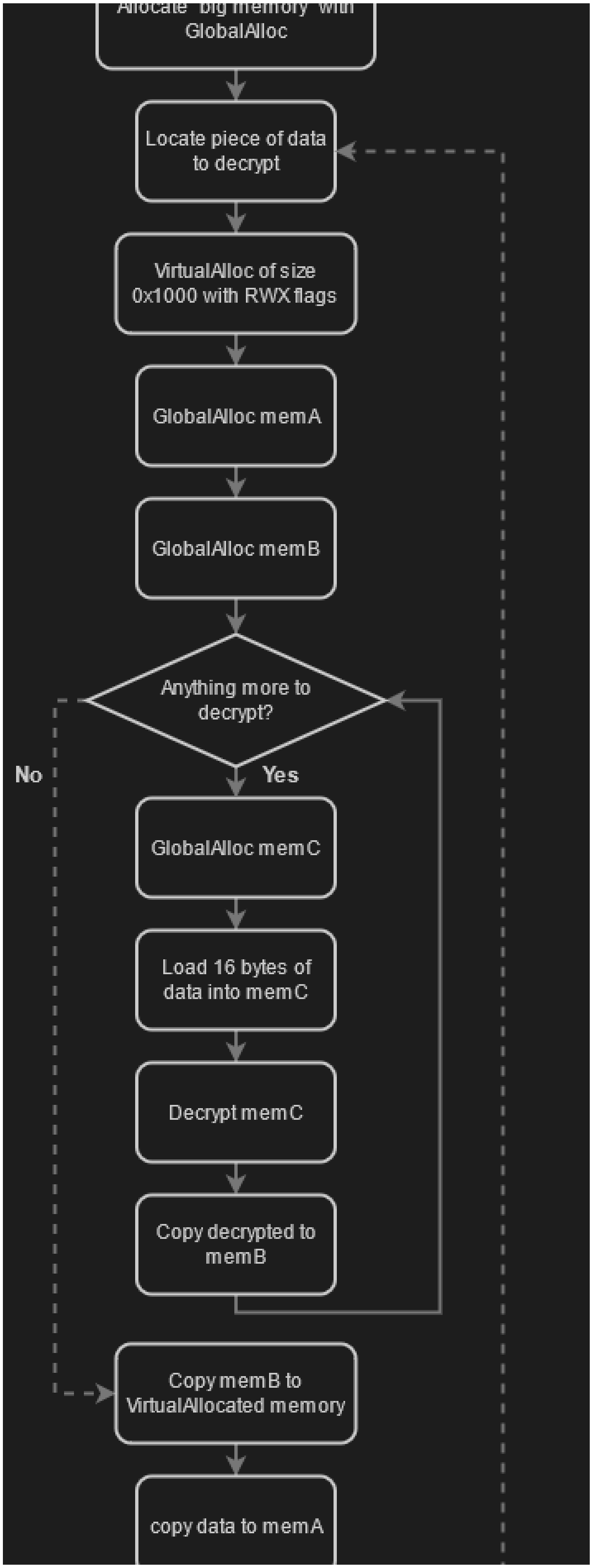
Decrypting next layer

Now shell code running on layer 2 starts decrypting layer 3. The structure of decryption is complex. At the highest level there is a big allocation of memory and a loop. Inside this loop is data decrypted in small chunks and copied into big memory, but it is not as simple as it seems.

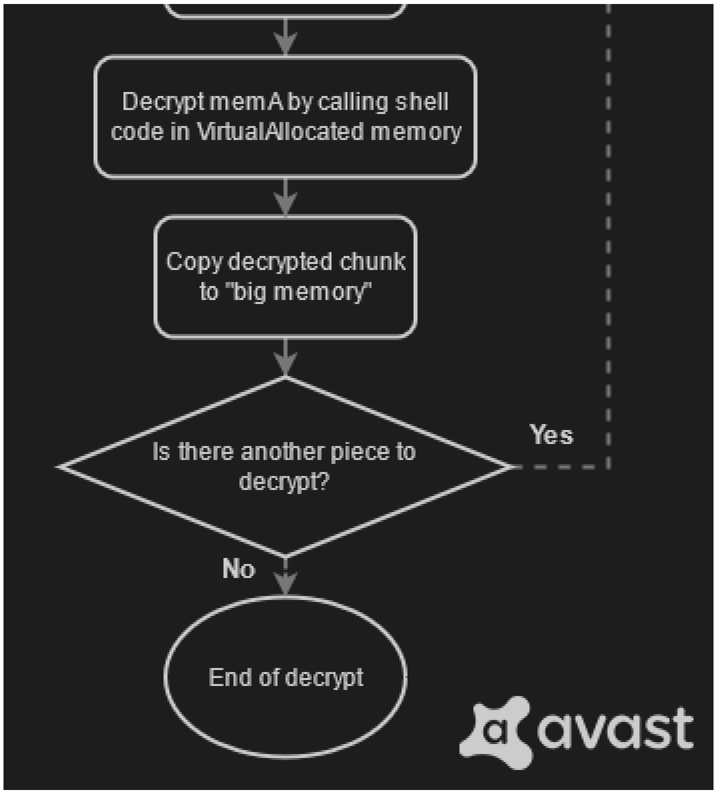
Before that data chunk gets decrypted, the program first does one `VirtualAlloc` of size 0x1000 bytes and with `RWX` flags. After that, the program starts decrypting pieces of data with size of 16 bytes and putting them together. This is accompanied by such a large number of memory allocations that hooking allocation functions is useless (and annoying).

After decrypting and joining the pieces with the size of 16 bytes, data is copied to VirtualAllocated memory. As it turned out, the data is another shellcode, which consists only of a decrypt loop. This shell code is called and decrypts some data from layer 2. Then the decrypted data is transformed again by another function and copied into memory, whose address is returned.





Main structure of decrypt next layer code

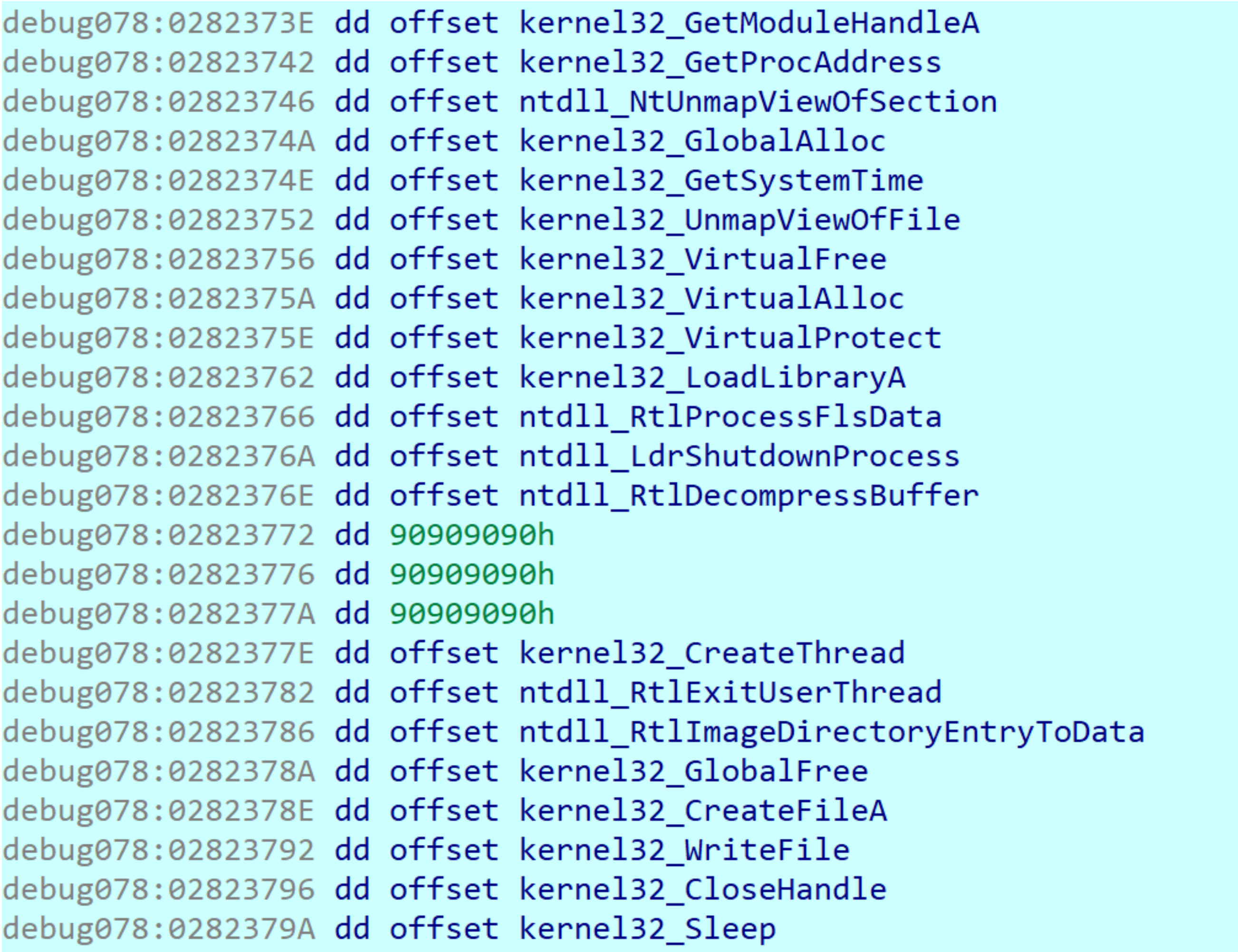


OnionCrypter has an option to compress data (or just some parts of data) with the `RtlCompressBuffer` function. This compression is used before encryption. During the decryption process chunks of data are decompressed after they are decrypted, but before they are merged with other chunks.

When all pieces are decrypted and joined, execution is passed to the place where the decrypted data is stored and the crypter starts execution of layer 3.

Layer 3

This layer is quite similar to the previous layer. At the beginning the same trick as described before is used to load some important API functions. This time the shell code loads even more functions than before.



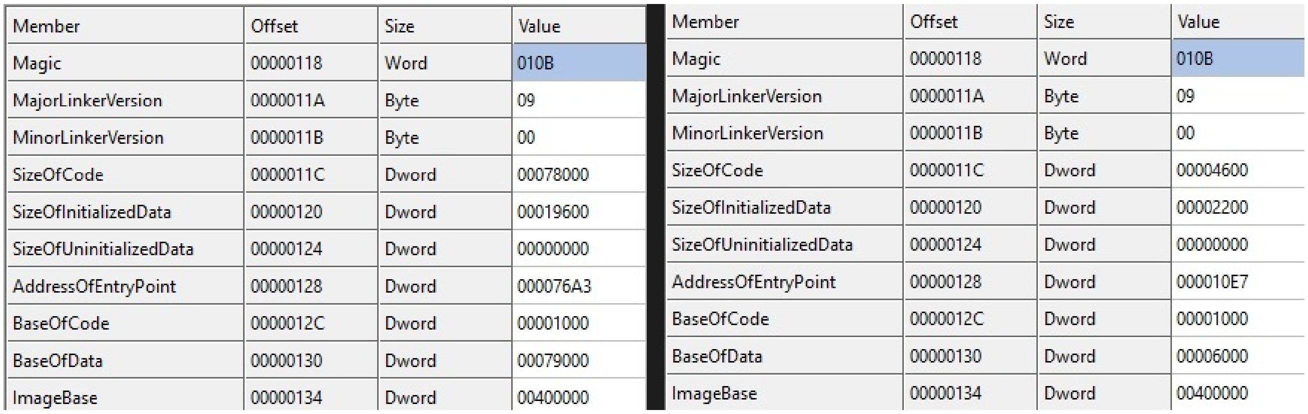
```
debug078:0282373E dd offset kernel32_GetModuleHandleA
debug078:02823742 dd offset kernel32_GetProcAddress
debug078:02823746 dd offset ntdll_NtUnmapViewOfSection
debug078:0282374A dd offset kernel32_GlobalAlloc
debug078:0282374E dd offset kernel32_GetSystemTime
debug078:02823752 dd offset kernel32_UnmapViewOfFile
debug078:02823756 dd offset kernel32_VirtualFree
debug078:0282375A dd offset kernel32_VirtualAlloc
debug078:0282375E dd offset kernel32_VirtualProtect
debug078:02823762 dd offset kernel32_LoadLibraryA
debug078:02823766 dd offset ntdll_RtlProcessFlsData
debug078:0282376A dd offset ntdll_LdrShutdownProcess
debug078:0282376E dd offset ntdll_RtlDecompressBuffer
debug078:02823772 dd 90909090h
debug078:02823776 dd 90909090h
debug078:0282377A dd 90909090h
debug078:0282377E dd offset kernel32_CreateThread
debug078:02823782 dd offset ntdll_RtlExitUserThread
debug078:02823786 dd offset ntdll_RtlImageDirectoryEntryToData
debug078:0282378A dd offset kernel32_GlobalFree
debug078:0282378E dd offset kernel32_CreateFileA
debug078:02823792 dd offset kernel32_WriteFile
debug078:02823796 dd offset kernel32_CloseHandle
debug078:0282379A dd offset kernel32_Sleep
```

Storage of loaded functions inside shell code
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

Even when these function pointers are loaded, they are not necessarily used. Some samples use `RtlDecompressBuffer` and some do not. The most probable cause of this is that OnionCrypter offers options like “additional compression” or “sleep”, which the user can choose when encrypting.

Decryption of the data is the same as in the previous layer. After decryption, OnionCrypter calls the `VirtualProtect` function in a loop and changes permissions of memory starting from the base address of the program itself to `R/W/X`. After this change, OnionCrypter copies decrypted data and overwrites itself, including the PE header and following sections. Then the program changes back memory permissions using `VirtualProtect` to ones that seem legit.

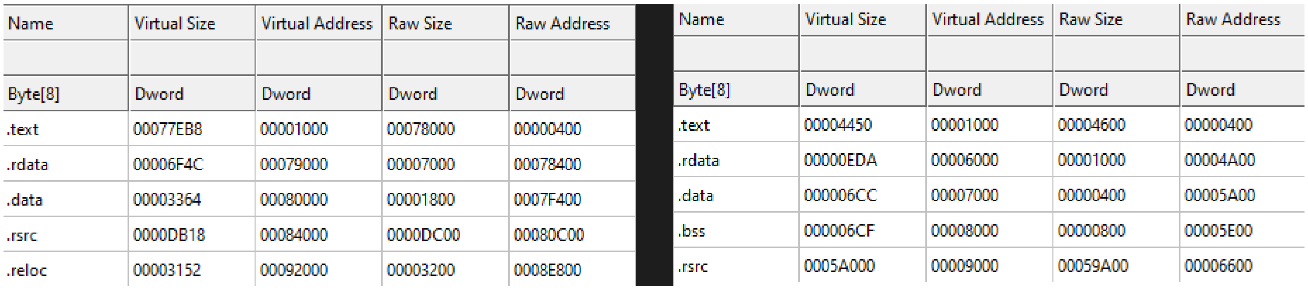
In the end, OnionCrypter finds the entry point in the new PE header and passes execution there. This is the point where the payload which is now injected into the crypter process starts running.



Member	Offset	Size	Value
Magic	00000118	Word	010B
MajorLinkerVersion	0000011A	Byte	09
MinorLinkerVersion	0000011B	Byte	00
SizeOfCode	0000011C	Dword	00078000
SizeOfInitializedData	00000120	Dword	00019600
SizeOfUninitializedData	00000124	Dword	00000000
AddressOfEntryPoint	00000128	Dword	000076A3
BaseOfCode	0000012C	Dword	00001000
BaseOfData	00000130	Dword	00079000
ImageBase	00000134	Dword	00400000

Member	Offset	Size	Value
Magic	00000118	Word	010B
MajorLinkerVersion	0000011A	Byte	09
MinorLinkerVersion	0000011B	Byte	00
SizeOfCode	0000011C	Dword	00004600
SizeOfInitializedData	00000120	Dword	00002200
SizeOfUninitializedData	00000124	Dword	00000000
AddressOfEntryPoint	00000128	Dword	000010E7
BaseOfCode	0000012C	Dword	00001000
BaseOfData	00000130	Dword	00006000
ImageBase	00000134	Dword	00400000

PE header information before and after self-injection
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE



Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	00077EB8	00001000	00078000	00000400
.rdata	00006F4C	00079000	00007000	00078400
.data	00003364	00080000	00001800	0007F400
.rsrc	0000DB18	00084000	0000DC00	00080C00
.reloc	00003152	00092000	00003200	0008E800

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	00004450	00001000	00004600	00000400
.rdata	00000EDA	00006000	00001000	00004A00
.data	000006CC	00007000	00000400	00005A00
.bss	000006CF	00008000	00000800	00005E00
.rsrc	0005A000	00009000	00059A00	00006600

Section headers before and after self-injection
8B85A4D9DF1140D25F11914EC4E429C505BD97551EDE19197D2B795C44770AFE

Conclusion

OnionCrypter is a malware family which has been around for some time. Combined with the prevalence of this crypter and the fact that samples have such a unique first layer it's logical to assume that crypter wasn't developed as a one time thing. On the contrary, according to analysis of multiple samples and their capture date, it was possible to see multiple versions of some parts of OnionCrypter.

Across all of samples these main features of the Onion crypter stay the same:

- **The three layer architecture**
- **Unique first layer with a lot of junk code**
- **Existence of the "main" function on layer 1**
- **General purpose and functionality of layer 2 and layer 3**

On the other hand these are some of the things that may vary between samples from different versions:

- **The decrypt algorithm of the second layer** – There can be found simpler and also more complicated decryption algorithms used to decrypt the layer 2, as was described in previous sections. It is improbable that authors would come up with a complicated algorithm and then change it to something simple, just to make analysis easier. That is why it is possible that this part of OnionCrypter was updated with newer versions.

- **The location of the “main” function** – In older samples the “main” function on layer 1 generally can be found very easily, because it is the `WinMain` function, which is the user-provided entry point of the application. This was changed in newer versions, because the majority of recently captured samples have quite a simple and short `WinMain` function and the “main” function can be found as one of the other functions.
- **Structure of layer 2 and layer 3** – Even though these layers can be found in all samples of OnionCrypter and always serve the same purpose they may differ in implementation. As an example there are versions, which are loading less DLL functions. Also in some older versions the loading of DLL functions is not a standalone function. Based on the analysis, the internal layers have been reworked a bit to make the layers more complex, to add new features and to make the decryption process more complicated and obfuscated.
- **Injection of the final payload** – Although the majority of samples are using the technique of self-injection described in the previous section, there were cases where the decrypted payload was injected into a new process created in a suspended state. This technique is analogous to the self injection, but is done using a combination of functions `CreateProcessInternalW`, `VirtualProtectEx`, `WriteProcessMemory` and `ResumeThread`.

This blogpost covered techniques discovered in both older and new versions of OnionCrypter. The whole process of decryption and execution of payload was described for the most complex and the most obfuscated versions, which can be considered to be the newest and the most difficult to analyze.

Indicators of Compromise (IoC)

- Hashes: https://github.com/avast/ioc/tree/master/OnionCrypter/samples.sha256
- List of the most common event names: https://github.com/avast/ioc/tree/master/OnionCrypter/event_names.txt

Appendix

- Repository: https://github.com/avast/ioc/tree/master/OnionCrypter
- IDAPython script for extraction of event names from samples: https://github.com/avast/ioc/tree/master/OnionCrypter/extras/extract_event_names.py

Tagged asanalysis, crypter, malware, obfuscation, reversing