oR10n Labs

or10nlabs.tech/reverse-engineering-the-mustang-panda-plugx-rat-extracting-the-config/

By oR10n

2020-07-05

HomeReverse EngineeringReverse Engineering the Mustang Panda PlugX RAT – Extracting the Config

Reverse Engineering the Mustang Panda PlugX RAT – Extracting the Config

Hello everyone! This is a continuation on the series of blog posts focused on reverse engineering a new-ish variant of PlugX malware gaining traction around the Asia Pacific region.

On my previous post, we reverse engineered the loader to determine how it decrypts, load, and execute the actual RAT component of PlugX. For this post, we will continue from where we left off and focus on one thing – extracting the malware configuration.

Introduction

Extracting malware configuration is one of the sub-tasks we can focus on when reverse engineering malware. By accomplishing this, we can obtain information used by the malware during execution such as hardcoded C2 addresses, bot IDs, mutexes, file paths, and registry keys among others.

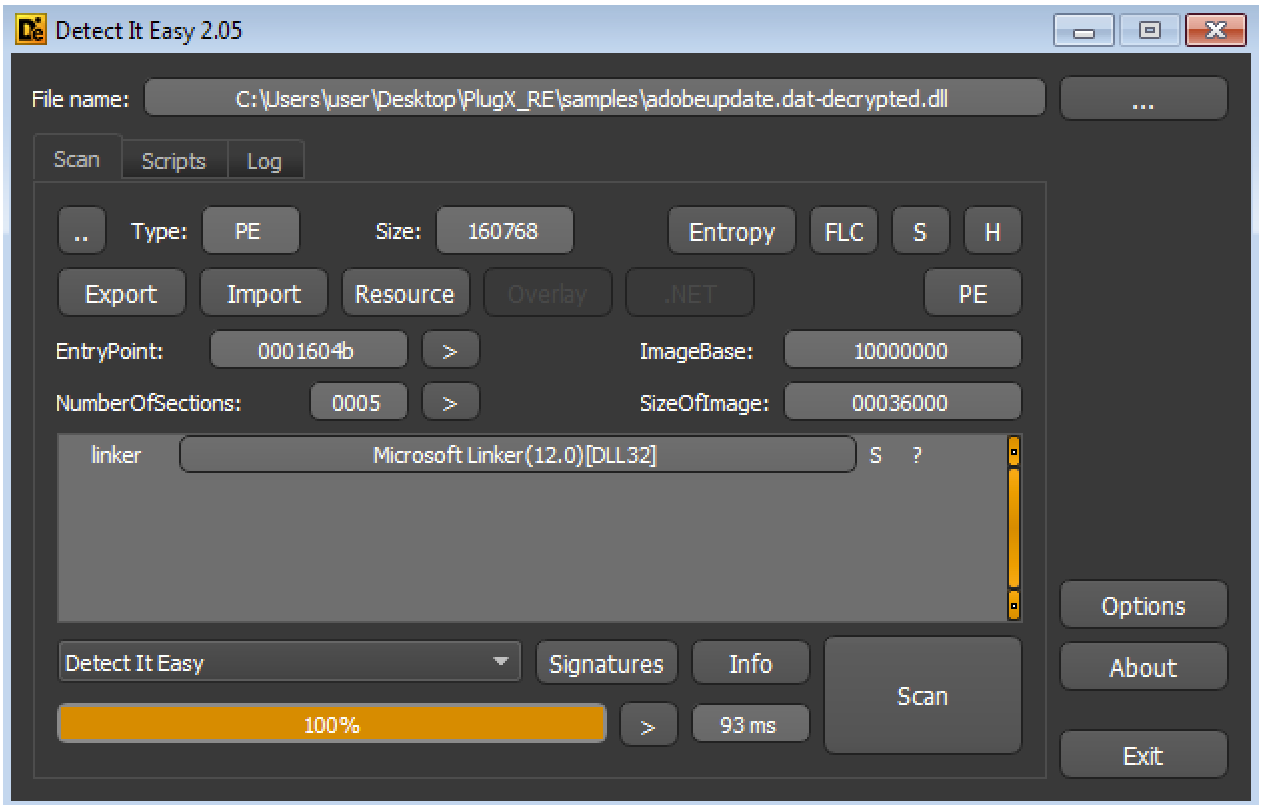
More often than not, a malware configuration is stored within the malware binary itself as an encoded or encrypted blob to prevent easy detection and analysis. We have to do some reverse engineering in order to figure out where it is located, how it is encoded or encrypted, and how to reverse the algorithm to see the plain text configuration. After successfully extracting the configuration, we can create an automation scrip to facilitate extraction to a huge number of samples. This is extremely useful for a number of use cases such as tracking C2 addresses for a specific malware family, blacklisting C2 addresses on network security devices, hunting for indicators-of-compromise within an environment, assisting incident responders during an investigation, and so on.

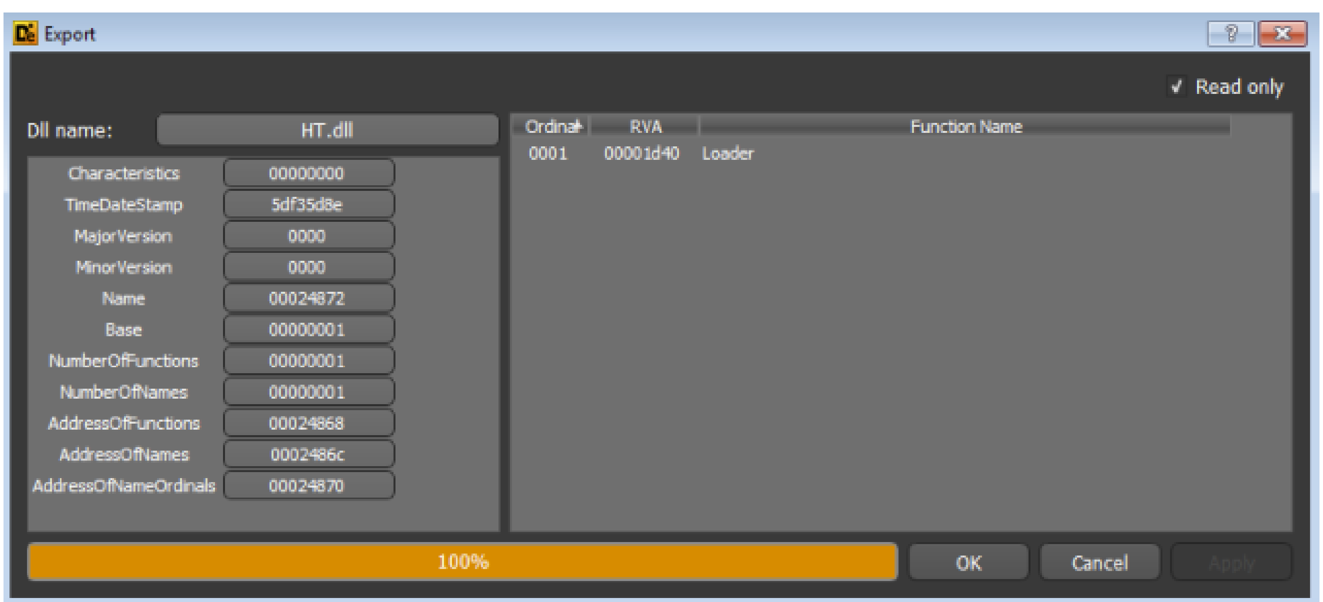
Let’s continue with the analysis.

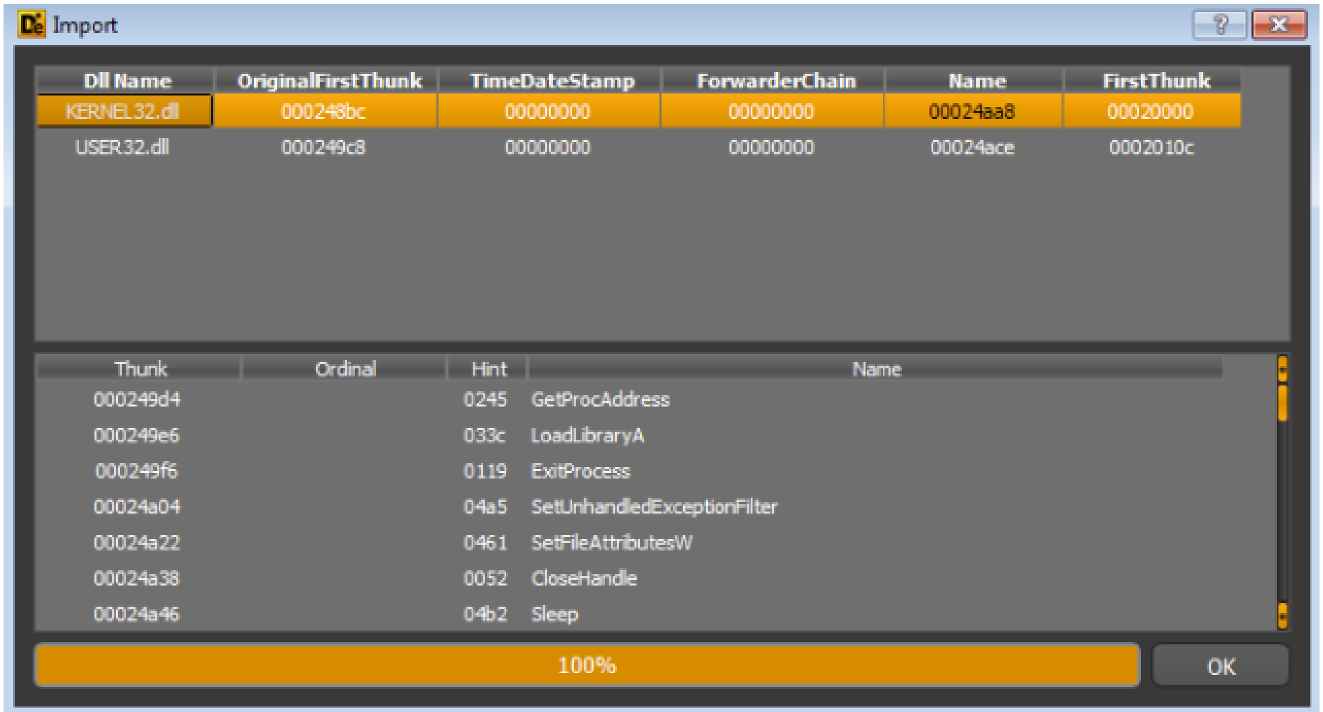
Extracting the configuration

On my previous post, we were able to figure out that the payload is decrypted by the loader using XOR with multi-byte key. We also created an automation script to decrypt the payload and save it to disk. We'll use this script to save a copy of the decrypted payload to disk and start from there.

Looking at the payload in DiE, we can see that it is a DLL file with a DLL name of **HT.dll**, 1 Export function named **Loader**, and a bunch of Import functions from **kernel32.dll** and **user32.dll**.

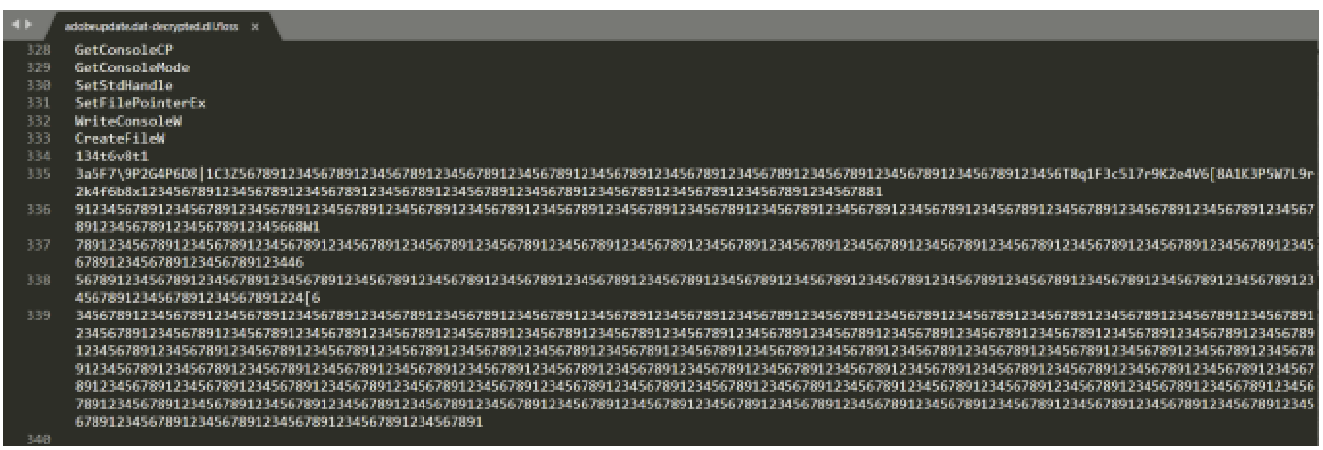






Interestingly enough, the Import functions for this sample are not that many and doesn't include typical functions you see for a RAT. Furthermore, the presence of **GetProcAddress** and **LoadLibraryA** potentially indicates that this sample dynamically resolves Win32 API functions at run time just like the loader.

Running FLOSS at the sample will give us a bunch of interesting strings to pivot from at IDA for a deeper analysis, but one interesting thing that immediately struck me was this blob with repeating pattern of "**123456789**".

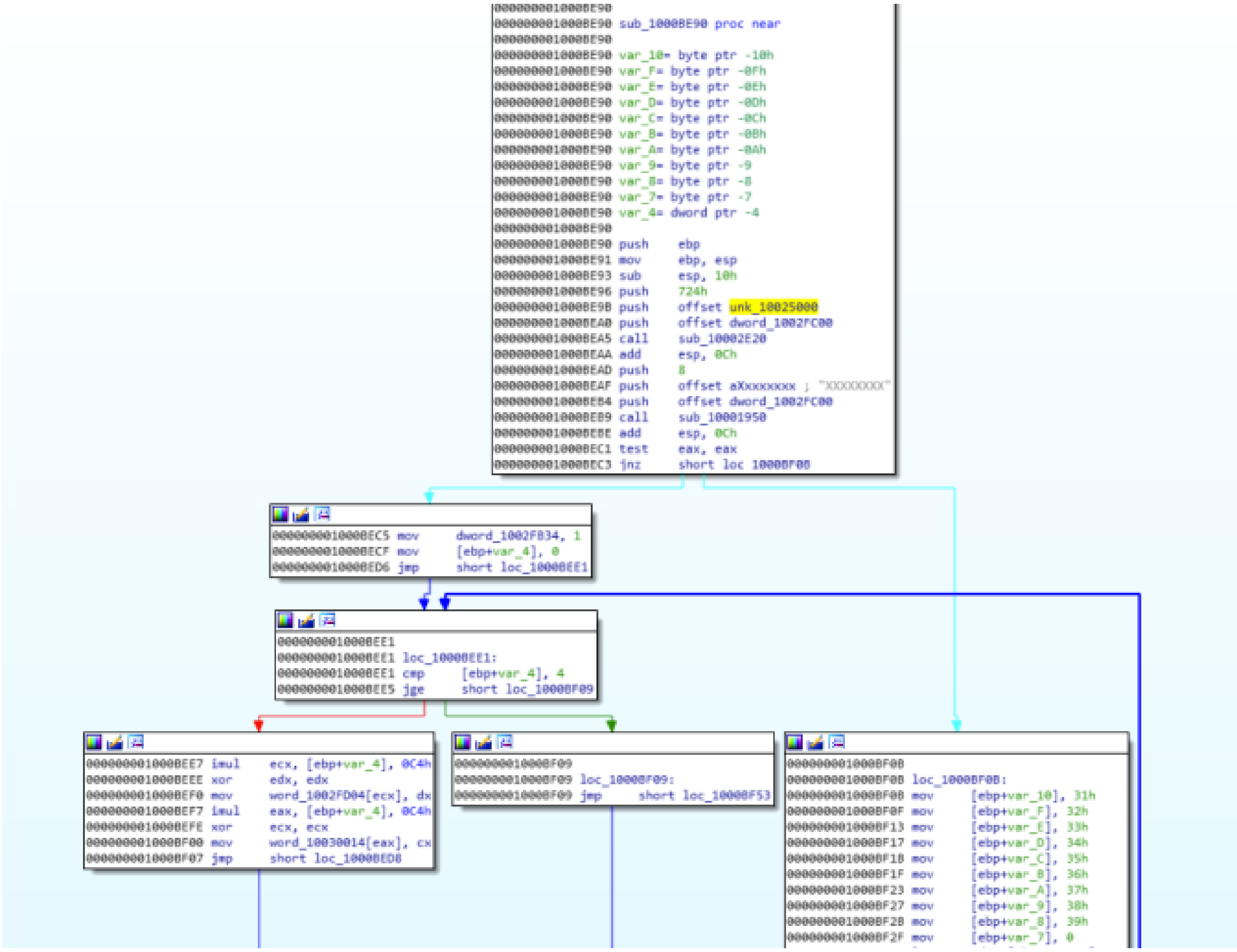


By this time, some of you may already have an idea of what this is potentially and why the repeating pattern of **123456789** is so interesting for us. But to be sure, let's take a closer look at the disassembly in IDA.

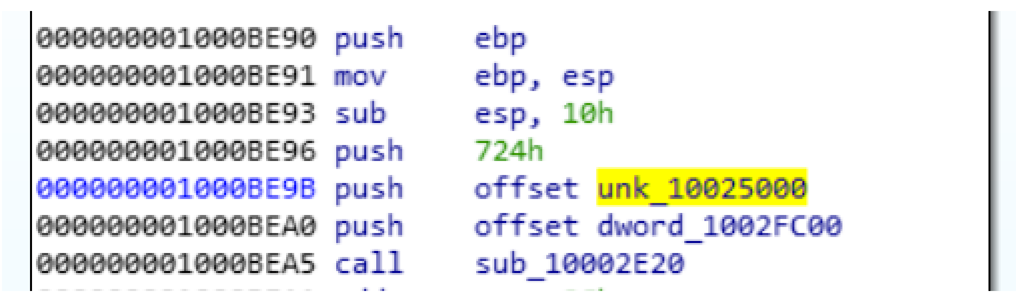
Clicking this on the **Strings window** of IDA will show us that this blob starts at offset 0 of the data section.

```
.data:10025000 ; Section 3. (virtual address 00025000)
.data:10025000 ; Virtual size                  : 0000D960 (  55648.)
.data:10025000 ; Section size in file          : 00001800 (   6144.)
.data:10025000 ; Offset to raw data for section: 00024400
.data:10025000 ; Flags C0000040: Data Readable Writable
.data:10025000 ; Alignment     : default
.data:10025000 ; ===========================================================================
.data:10025000
.data:10025000 ; Segment type: Pure data
.data:10025000 ; Segment permissions: Read/Write
.data:10025000 _data           segment para public 'DATA' use32
.data:10025000                 assume cs:_data
.data:10025000                 ;org 10025000h
.data:10025000 unk_10025000    db 0D9h ; Ù             ; DATA XREF: sub_1000BE90+B↑o
.data:10025001                 db  31h ; 1
.data:10025002                 db  33h ; 3
.data:10025003                 db  34h ; 4
.data:10025004                 db  74h ; t
.data:10025005                 db  36h ; 6
.data:10025006                 db  76h ; v
.data:10025007                 db  38h ; 8
.data:10025008                 db  74h ; t
.data:10025009                 db  31h ; 1
.data:1002500A                 db  12h
.data:1002500B                 db  33h ; 3
.data:1002500C                 db  61h ; a
.data:1002500D                 db  35h ; 5
.data:1002500E                 db  46h ; F
.data:1002500F                 db  37h ; 7
.data:10025010                 db  5Ch ; \
.data:10025011                 db  39h ; 9
.data:10025012                 db  50h ; P
.data:10025013                 db  32h ; 2
.data:10025014                 db  47h ; G
.data:10025015                 db  34h ; 4
.data:10025016                 db  50h ; P
.data:10025017                 db  36h ; 6
.data:10025018                 db  44h ; D
.data:10025019                 db  38h ; 8
.data:1002501A                 db  7Ch ; |
.data:1002501B                 db  31h ; 1
.data:1002501C                 db  43h ; C
.data:1002501D                 db  33h ; 3
.data:1002501E                 db  5Ah ; Z
.data:1002501F                 db  35h ; 5
.data:10025020                 db  36h ; 6
```

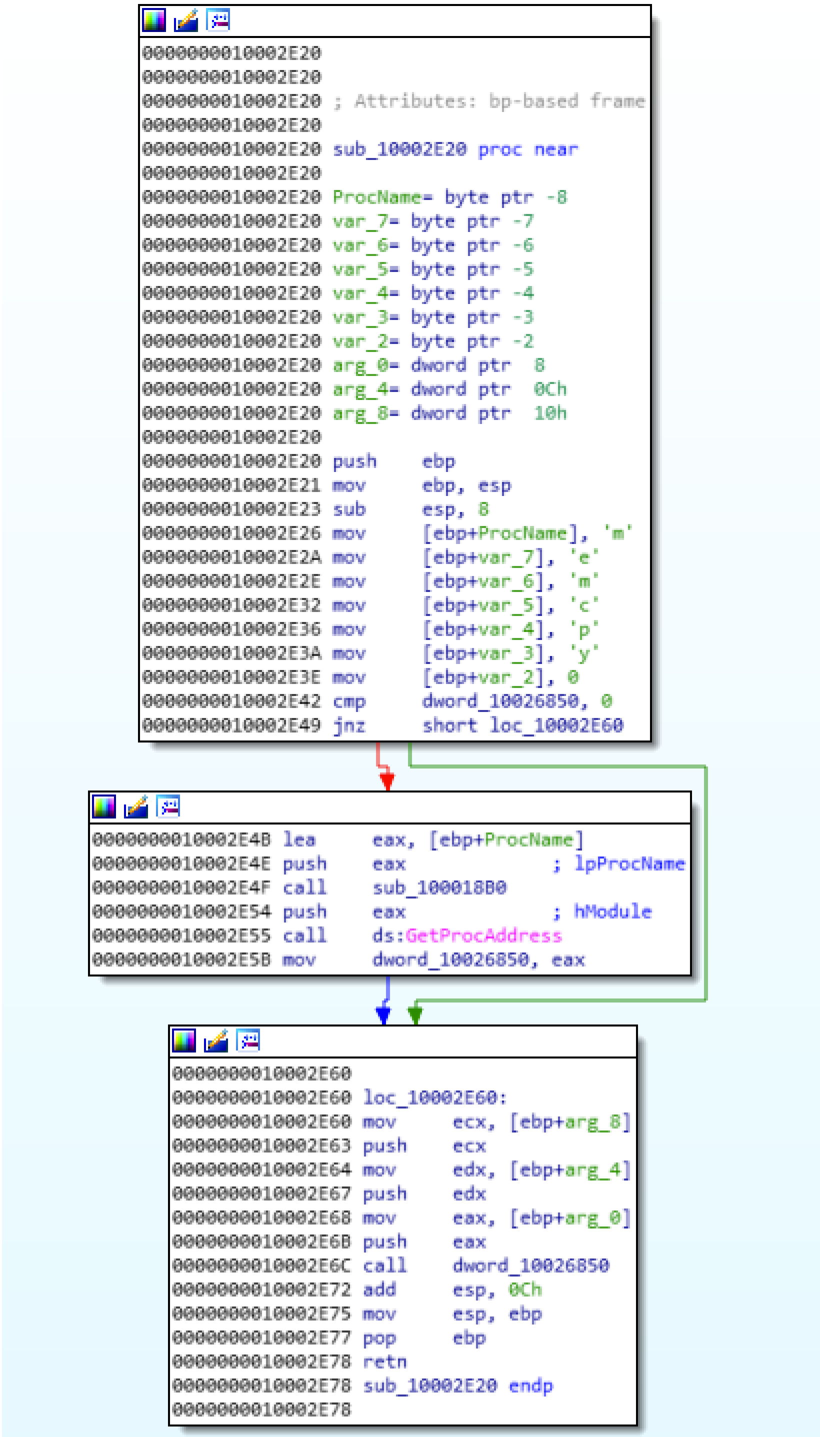
Checking the x-refs of unk_10025000, will bring us to a function at **sub_1000BE90**.

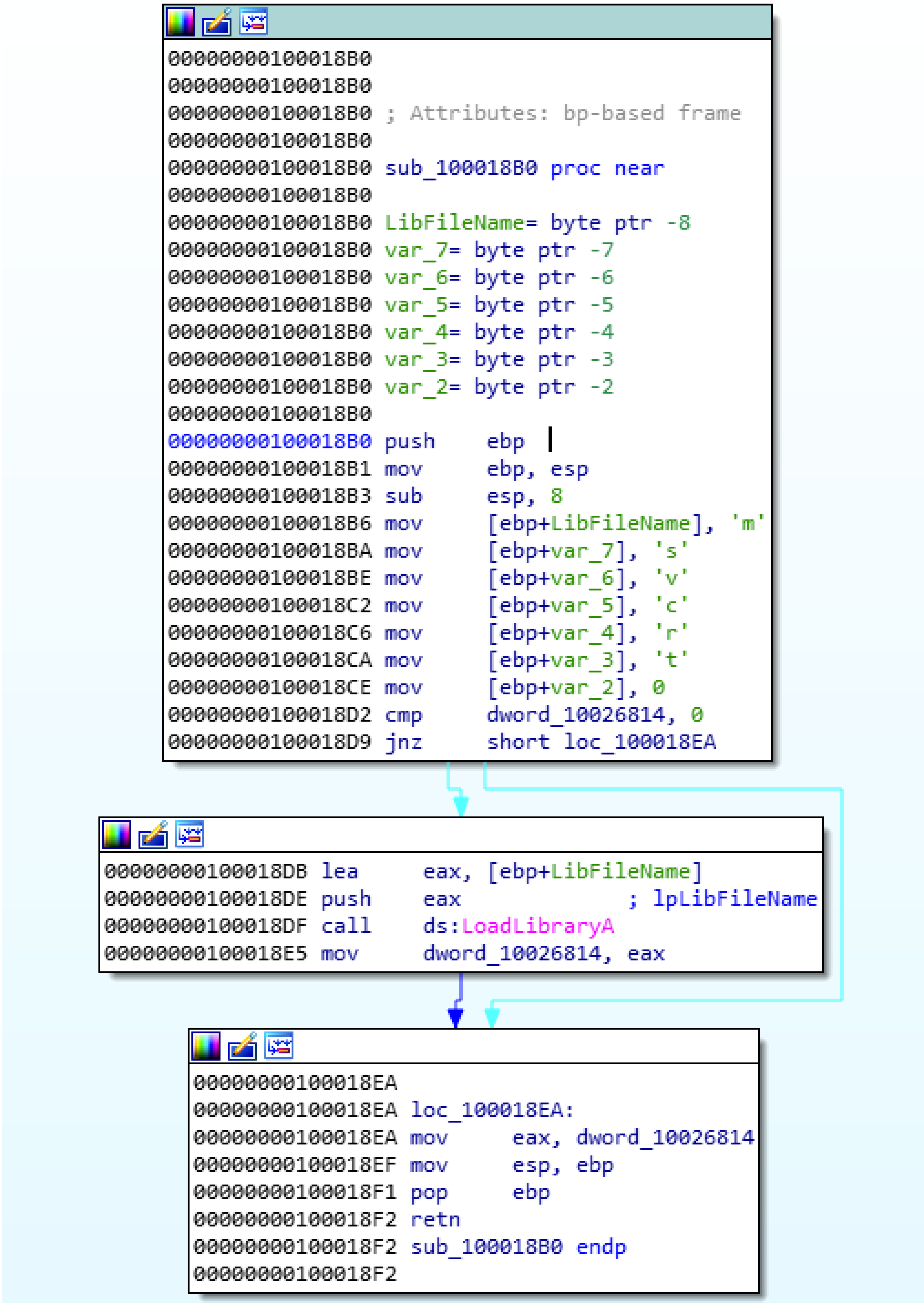


As you can see, **unk_10025000** is pushed onto the stack along with **724h** and another memory offset **dword_1002FC00** as parameters for a function located at **sub_10002E20**.



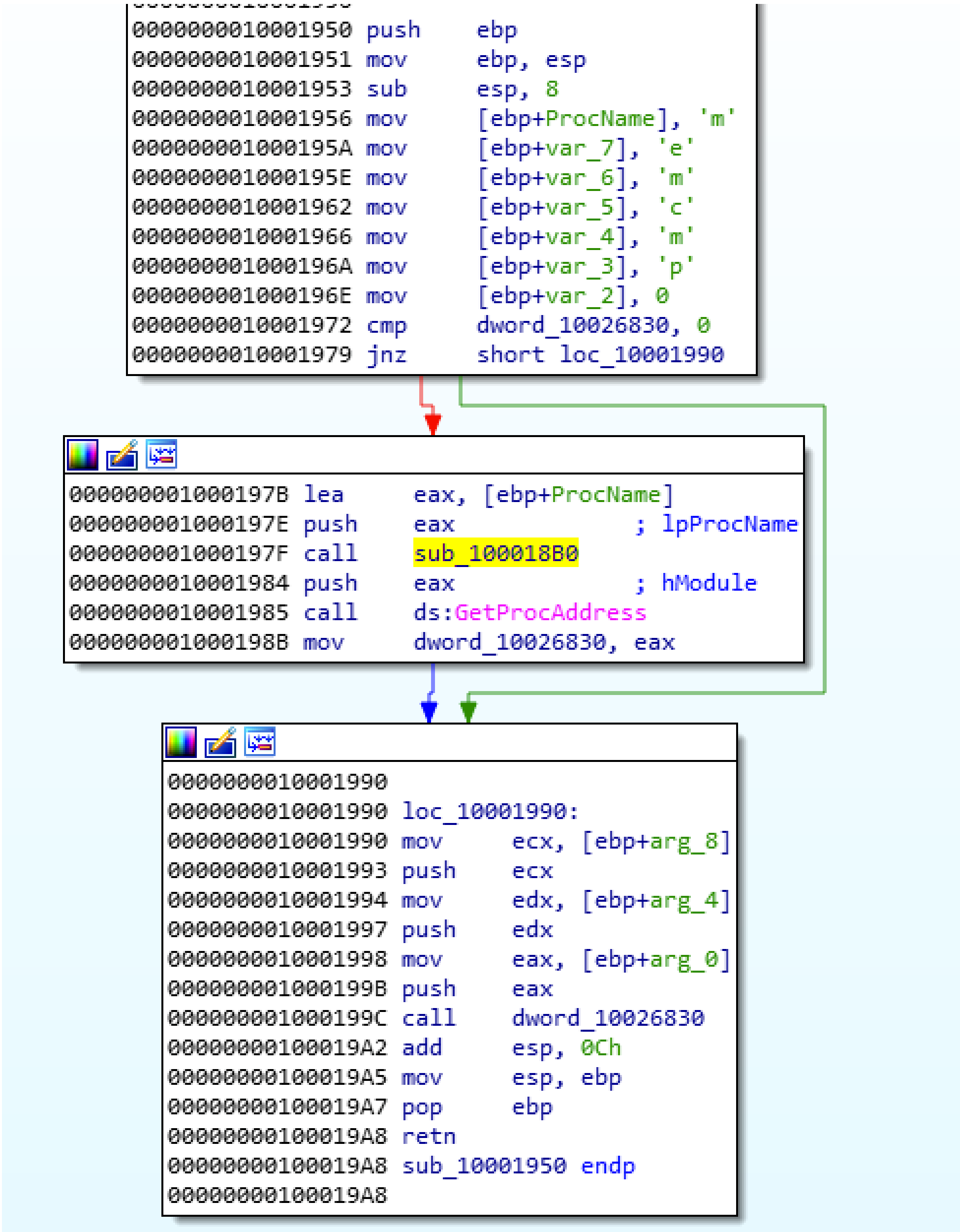
Looking closer at **sub_10002E20**, we can immediately figure out that this is a wrapper function that dynamically resolves the address of **memcpy** from **msvcrt** (**sub_100018B0**) via **GetProcAddress** and **LoadLibrary** (**sub_100018B0**) and then calls it with the parameters passed to the function.


0000000010002E20
0000000010002E20
0000000010002E20 ; Attributes: bp-based frame
0000000010002E20
0000000010002E20 sub_10002E20 proc near
0000000010002E20
0000000010002E20 ProcName= byte ptr -8
0000000010002E20 var_7= byte ptr -7
0000000010002E20 var_6= byte ptr -6
0000000010002E20 var_5= byte ptr -5
0000000010002E20 var_4= byte ptr -4
0000000010002E20 var_3= byte ptr -3
0000000010002E20 var_2= byte ptr -2
0000000010002E20 arg_0= dword ptr 8
0000000010002E20 arg_4= dword ptr 0Ch
0000000010002E20 arg_8= dword ptr 10h
0000000010002E20
0000000010002E20 push ebp
0000000010002E21 mov ebp, esp
0000000010002E23 sub esp, 8
0000000010002E26 mov [ebp+ProcName], 'm'
0000000010002E2A mov [ebp+var_7], 'e'
0000000010002E2E mov [ebp+var_6], 'm'
0000000010002E32 mov [ebp+var_5], 'c'
0000000010002E36 mov [ebp+var_4], 'p'
0000000010002E3A mov [ebp+var_3], 'y'
0000000010002E3E mov [ebp+var_2], 0
0000000010002E42 cmp dword_10026850, 0
0000000010002E49 jnz short loc_10002E60
0000000010002E4B lea eax, [ebp+ProcName]
0000000010002E4E push eax ; lpProcName
0000000010002E4F call sub_100018B0
0000000010002E54 push eax ; hModule
0000000010002E55 call ds:GetProcAddress
0000000010002E5B mov dword_10026850, eax
0000000010002E60
0000000010002E60 loc_10002E60:
0000000010002E60 mov ecx, [ebp+arg_8]
0000000010002E63 push ecx
0000000010002E64 mov edx, [ebp+arg_4]
0000000010002E67 push edx
0000000010002E68 mov eax, [ebp+arg_0]
0000000010002E6B push eax
0000000010002E6C call dword_10026850
0000000010002E72 add esp, 0Ch
0000000010002E75 mov esp, ebp
0000000010002E77 pop ebp
0000000010002E78 retn
0000000010002E78 sub_10002E20 endp
0000000010002E78



In simpler terms, this function copies the first **1828** bytes of the data section (**unk_10025000**) to another memory location (**dword_1002FC00**).

After that, **dword_1002FC00** is pushed onto the stack along with **8** and another memory offset **aXxxxxxxx** as parameters for a function located at **sub_10001950**. This function is quite similar with the previous one but dynamically resolves the address of **memcmp** from **msvcrt** and then calls it with the parameters passed to the function.



Again in simpler terms, this function compares the first 8 bytes of **dword_1002FC00** to "**XXXXXXXX**". The result of the comparison determines the execution path of the malware.

Note: The malware uses a lot of wrapper functions like this to dynamically resolve the address of various Win32 API functions from different DLLs via **GetProcAddress** and **LoadLibrary** and then execute it. This is one of the anti-detection measures implemented by this specific PlugX variant and is common through out the whole binary.

Let's focus on the execution path where it doesn't match "**XXXXXXXX**". As you can see, it pushes the string "**123456789**" onto the stack and calls a function at **sub_10002DC0**.

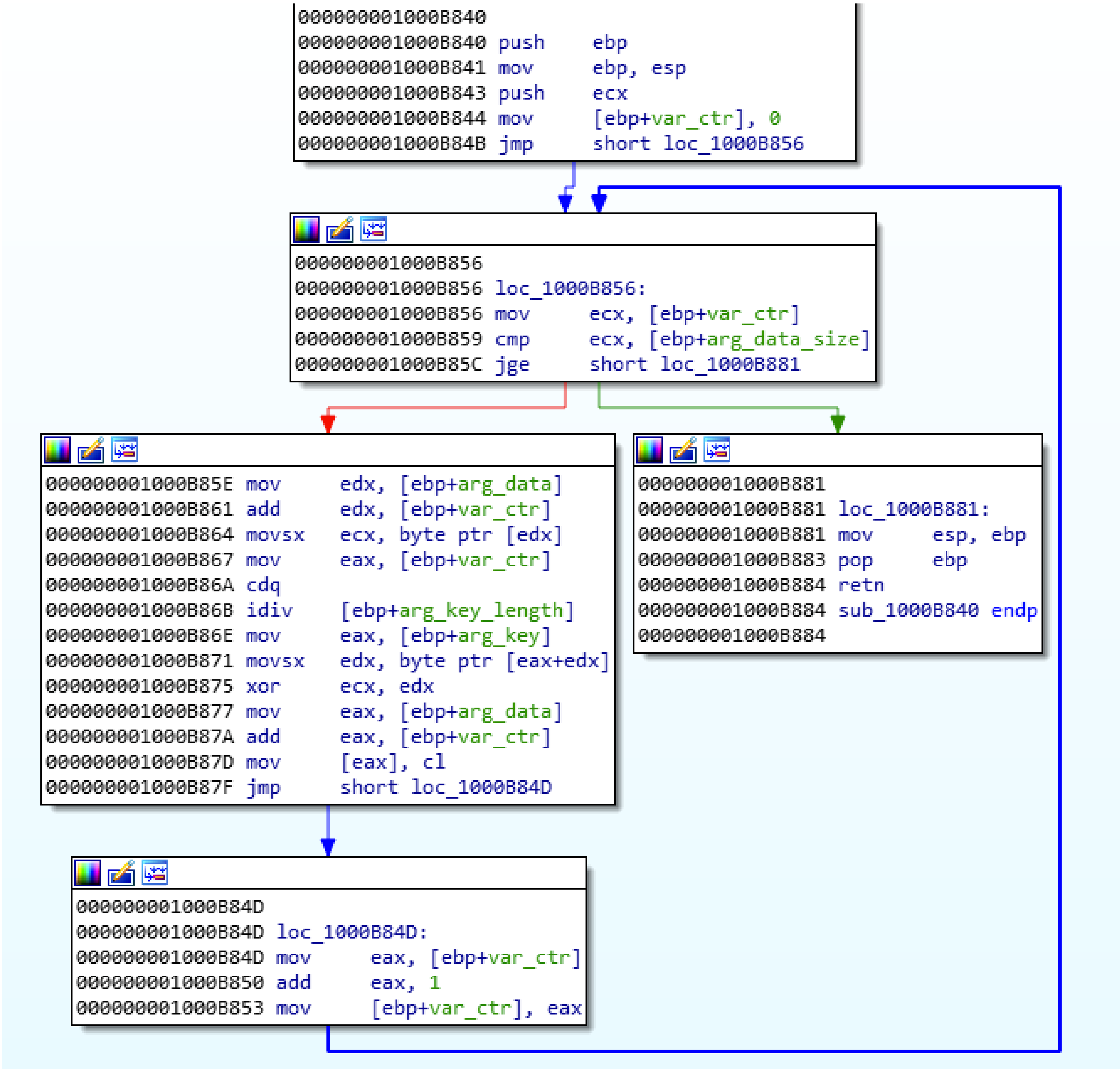
```
000000001000BF0B
000000001000BF0B loc_1000BF0B:
000000001000BF0B mov     [ebp+var_10], '1'
000000001000BF0F mov     [ebp+var_F], '2'
000000001000BF13 mov     [ebp+var_E], '3'
000000001000BF17 mov     [ebp+var_D], '4'
000000001000BF1B mov     [ebp+var_C], '5'
000000001000BF1F mov     [ebp+var_B], '6'
000000001000BF23 mov     [ebp+var_A], '7'
000000001000BF27 mov     [ebp+var_9], '8'
000000001000BF2B mov     [ebp+var_8], '9'
000000001000BF2F mov     [ebp+var_7], 0
000000001000BF33 lea     edx, [ebp+var_10]
000000001000BF36 push    edx
000000001000BF37 call    sub_10002DC0
000000001000BF3C push    eax
000000001000BF3D lea     eax, [ebp+var_10]
000000001000BF40 push    eax
000000001000BF41 push    724h
000000001000BF46 push    offset dword_1002FC00
000000001000BF4B call    sub_1000B840
000000001000BF50 add     esp, 10h
```

This is another wrapper function but this time, for **lstrlenA** to find the length of the string passed as a parameter. The resulting string length is stored in **EAX** and is pushed onto the stack along with the string “**123456789**“, **724h**, and **dword_1002FC00** which contains the blob we noted earlier before a call to **sub_1000B840** is made.

```
000000001000BF0B
000000001000BF0B loc_1000BF0B:
000000001000BF0B mov     [ebp+var_10], '1'
000000001000BF0F mov     [ebp+var_F], '2'
000000001000BF13 mov     [ebp+var_E], '3'
000000001000BF17 mov     [ebp+var_D], '4'
000000001000BF1B mov     [ebp+var_C], '5'
000000001000BF1F mov     [ebp+var_B], '6'
000000001000BF23 mov     [ebp+var_A], '7'
000000001000BF27 mov     [ebp+var_9], '8'
000000001000BF2B mov     [ebp+var_8], '9'
000000001000BF2F mov     [ebp+var_7], 0
000000001000BF33 lea     edx, [ebp+var_10]
000000001000BF36 push    edx
000000001000BF37 call    sub_10002DC0
000000001000BF3C push    eax
000000001000BF3D lea     eax, [ebp+var_10]
000000001000BF40 push    eax
000000001000BF41 push    724h
000000001000BF46 push    offset dword_1002FC00
000000001000BF4B call    sub_1000B840
000000001000BF50 add     esp, 10h
```

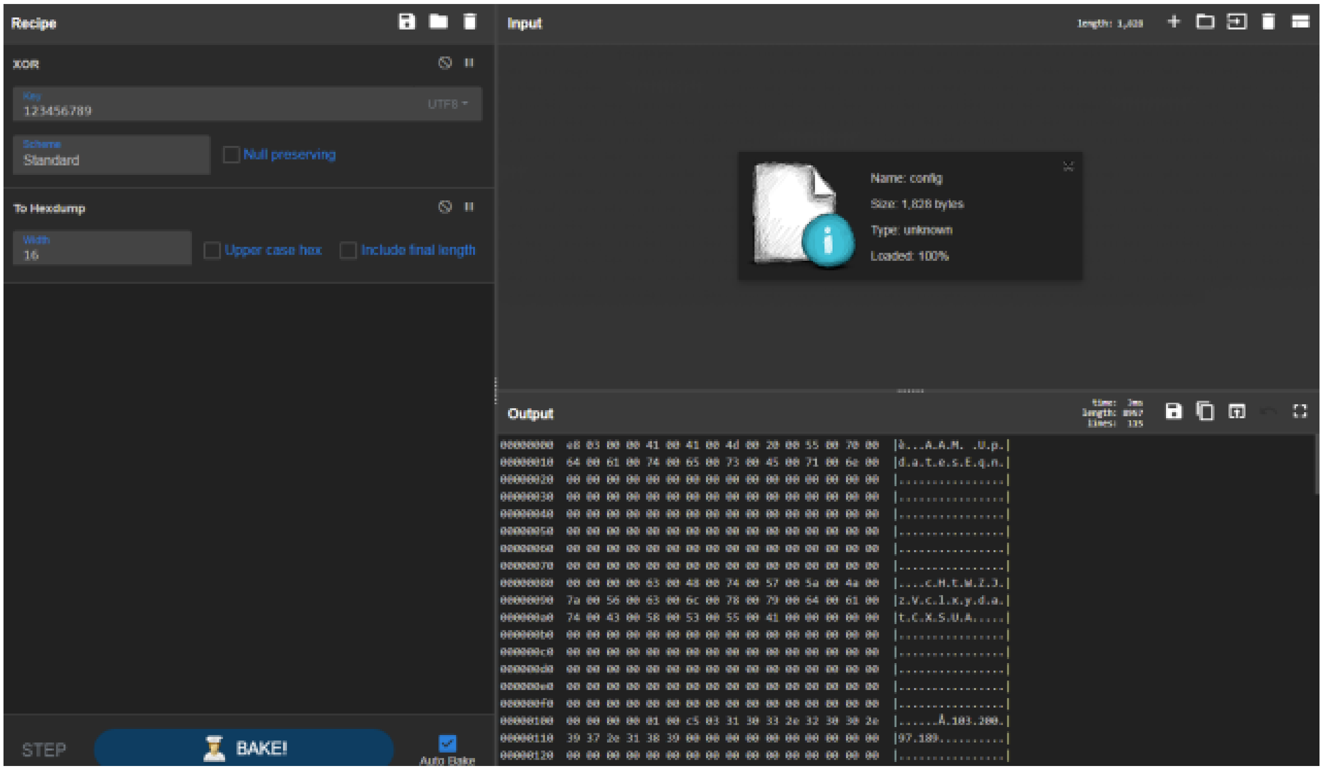
Looks familiar right? Yes, this is the same format of parameters we noted on the loader before the call to the decryption function – the key length, the key, the length of the encrypted data, and the address of the encrypted data.

Looking closer at **sub_1000B840** and due to the fact that there are repeating patterns of **123456789** on the encrypted blob, we can immediately recognize that the algorithm used to encrypt it is XOR with multi-byte key as well.



So by extracting the first **1828** bytes of the data section and performing a multi-byte XOR on it with the key **123456789**, we can decrypt the blob and see if it really contains the malware configuration.

Let’s try it with a quick Cyberchef recipe:



As you can see, there are a few interesting information in here such as:
* An Adobe themed unicode string “**AAM UpdatesEqn**”
* A random looking unicode string “**cHtWZJzVclxydatCXSUA**”
* IP addresses that are likely the C2 addresses

Each IP address entry looks like it starts with **01 00**, then followed by two bytes which are likely port numbers in hex, and then finally the IP address itself.

If we convert the port numbers in hex to decimal (little endian) and the IP addresses in hex to ascii we will get the following addresses:

```
103.200.97[.]189:965
103.200.97[.]189:110
185.239.226[.]17:965
185.239.226[.]17:110
```

Let’s try to confirm our findings by running the sample on a VM and monitoring for any interesting events using some dynamic analysis tools.

File, registry, and process events captured by Procmon:

Time of Day	Process Name	PID	Operation	Path	Result
12:25:47.8104818 PM	AdobeInstall.exe	3464	RegCreateKey	HKCR\ms-pu	SUCCESS
12:25:47.8194847 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe	SUCCESS
12:25:47.8195258 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe	SUCCESS
12:25:47.8195747 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe	SUCCESS
12:25:47.8227567 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\hex.dll	SUCCESS
12:25:47.8258334 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\adobeupdate.dat	SUCCESS
12:25:47.8258685 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\adobeupdate.dat	SUCCESS
12:25:47.8259203 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\adobeupdate.dat	SUCCESS
12:25:47.8263681 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run	ACCESS DENIED
12:25:47.8264076 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\Software\Wow6432Node	SUCCESS
12:25:47.8264242 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\SOFTWARE\Wow6432Node\Microsoft	SUCCESS
12:25:47.8264495 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows	SUCCESS
12:25:47.8264720 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion	SUCCESS
12:25:47.8264900 PM	AdobeInstall.exe	3464	RegCreateKey	HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run	ACCESS DENIED
12:25:47.8265332 PM	AdobeInstall.exe	3464	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:25:47.8265583 PM	AdobeInstall.exe	3464	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\AAM UpdatesEqn	SUCCESS
12:25:47.8272756 PM	AdobeInstall.exe	3464	WriteFile	C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe	SUCCESS
12:25:47.8330298 PM	AdobeInstall.exe	3464	Process Create	C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe	SUCCESS

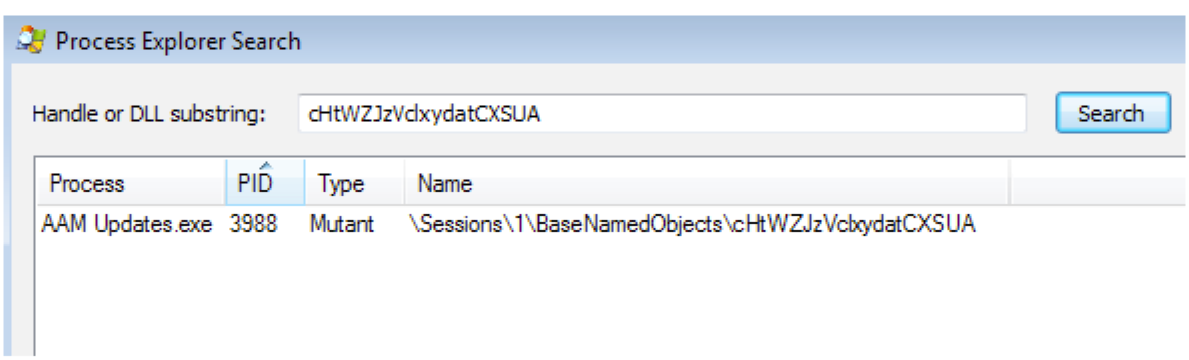
From the Procmon output, we can see that the malware:
* Created a new folder named **AAM UpdatesEqn** on the Program Data directory and copied the PlugX components to it
* Attempted to access **HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\Current Version\Run** for registry write operation but failed
* Succeeded in accessing **HKCU\SOFTWARE\Microsoft\Windows\Current Version\Run** for registry write operation
* Created a new value named **AAM UpdatesEqn**. This is a persistence mechanism which will execute **"C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe" 701** across system reboots
* Created a new process for **C:\ProgramData\AAM UpdatesEqn\AAM Updates.exe** passing the value **701** as parameter

HTTP request captured by fakenet:

```
06/28/20 12:28:50 PM [     HTTPListener80]
06/28/20 12:28:50 PM [     HTTPListener80] Storing HTTP POST headers and data to http_20200628_122850.txt.
06/28/20 12:28:52 PM [     HTTPListener80]   POST /update?wd=06304bb6 HTTP/1.1
06/28/20 12:28:52 PM [     HTTPListener80]   Accept: */*
06/28/20 12:28:52 PM [     HTTPListener80]   x-debug: 0
06/28/20 12:28:52 PM [     HTTPListener80]   x-request: 0
06/28/20 12:28:52 PM [     HTTPListener80]   x-content: 61456
06/28/20 12:28:52 PM [     HTTPListener80]   x-storage: 1
06/28/20 12:28:52 PM [     HTTPListener80]   User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;SV1;
06/28/20 12:28:52 PM [     HTTPListener80]   Host: 103.200.97.189:965
06/28/20 12:28:52 PM [     HTTPListener80]   Content-Length: 0
06/28/20 12:28:52 PM [     HTTPListener80]   Connection: Keep-Alive
06/28/20 12:28:52 PM [     HTTPListener80]   Cache-Control: no-cache
06/28/20 12:28:52 PM [     HTTPListener80]
```

From the Fakenet output, we can see that the malware communicated to one of the IP address and port combinations we noted earlier.

Searching for cHtWZJzVclxydatCXSUA handle in Procexp:



Lastly, we can see that **cHtWZJzVcIxydatCXSUA** is a mutex object used by the malware.

These observations confirm that we were successful in extracting the configuration of the malware.

Automating the config extraction

To make our lives easier, I created this quick-and-dirty python script to automatically extract the configuration information for this variant:

I've tested this script on a bunch of samples and it seems to work fine. However, there can be instances where it won't work if the config is structured differently.

That's it guys! I really hope you learned something new today and as always, thank you for reading my blog.

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