

# Aria-Body Loader? Is that you?

By BlueMonkey

Published: 2021-09-29 · Archived: 2026-04-05 18:51:43 UTC

b

Hello! This is my first time writing a blog on a loader which I had gotten hold of. So, I am a new analyst in the Malware Analysis field and I am trying to do a research into cobalt strike. Recently, I ran a modified public YARA rule for cobalt strike on Virus Total and from the scan, I found two samples which I find interesting.

```
1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c
```

```
c5696e660f3cfa9232756418e40ad18729cfe32fb284bba2314dd523ba527258
```

These two samples have a relative large size (17.05 MB) as compared to other files and their size is exactly the same. Additionally, their upload timing is quite close to each other, around 3 minutes apart. I started to analyze the files and from my finding, it doesn't look anything like a cobalt strike loader. Turning to my friend who have much more experience in malware analysis, I told him my findings and he told me that from what I had described, it sounds a little bit like Aria-Body instead. So I did some read up on Aria-Body and here are what I had found...

In 2020, Check Point Research release a write up ([http://research\[.\]checkpoint\[.\]com/2020/nikon-apt-cyber-espionage-reloaded](http://research[.]checkpoint[.]com/2020/nikon-apt-cyber-espionage-reloaded)) describing how Naikon APT group is using Aria-body. In the report, they summarized the loader to have these capabilities:

1. *Establish persistence via the Startup folder or the Run registry key [some variants].*
2. *Inject itself to another process such as rundll32.exe and dllhost.exe [some variants].*
3. *Decrypt two blobs: Import Table and the loader configuration.*
4. *Utilize a DGA algorithm if required*
5. *Contact the embedded / calculated C&C address in order to retrieve the next stage payload.*
6. *Decrypt the received payload DLL (Aria-Body backdoor).*
7. *Load and execute an exported function of the DLL — calculated using djb2 hashing algorithm.*

Take note on these points as I will be mentioning some of them in this post.

## Analysis of the samples

As I had mentioned in the intro, the first thing that I noticed about the two sample is that both of them have exactly the same size. When I loaded them into PE studio, I noticed that they share the same compiler, debugger

and exports timestamp.

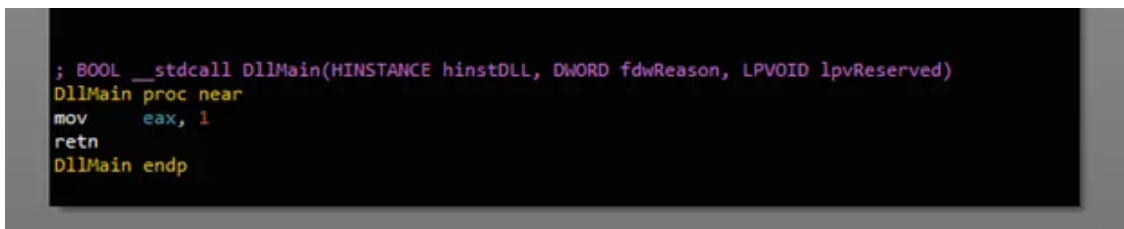
Now looking at the section's metadata, we can see that all but two of them have the same hash. The two sections with the different hash is the *.text* and the *.data* section. Although they have a different hash, we can see that the raw-size, virtual-size, raw-address and virtual-address are the same.

Looking into the import and export section, this two samples also have a same import and export.

Looking at the static properties analysis, these two files seems like twins. They could have the same origins or they might be built using a builder. But this is just my speculation at this point as there are not enough information to support the claim.

### Looking for the “action”

Now that I had done the analysis on the files properties and confirmed that it is a 64 bit DLL, it's time to throw the sample 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c into IDA for analysis. After IDA has finished loading, the first thing that it displayed is this:



```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
DllMain proc near
mov     eax, 1
retn
DllMain endp
```

DllMain

That's all *..* not helpful at all. Seems like I have to find the “action” through other means~~

Since this is a DLL, we can try to look for the “action” by checking the export functions.

	SHFileOperation	0000000180001210	1
	SHFormatDrive	<b>0000000180001260</b>	2
	SHFreeNameMappings	<b>0000000180001290</b>	3
		00000001800015F0	4
		00000001800015F0	5
	SHGetDesktopFolder	<b>00000001800012C0</b>	6
	SHGetDriveMedia	<b>00000001800012F0</b>	7
	SHGetFolderPathEx	0000000180001320	8
	SHGetIDListFromObject	<b>0000000180001350</b>	9
		00000001800015F0	10
	SHGetLocalizedName	<b>0000000180001380</b>	11
	SHGetItemFromObject	<b>00000001800013B0</b>	12
	SHGetNewLinkInfo	00000001800013E0	13
	ILCreateFromPath	0000000180001980	14
	DragAcceptFiles	<b>0000000180001410</b>	15
		00000001800015F0	16
		00000001800015F0	17
		00000001800015F0	18
		00000001800015F0	19
		00000001800015F0	20
	DragQueryPoint	<b>0000000180001470</b>	21
	DuplicateIcon	<b>00000001800014A0</b>	22
	ExtractIcon	00000001800014D0	23
	PifMgr_OpenProperties	<b>0000000180001500</b>	24
	PifMgr_GetProperties	<b>0000000180001530</b>	25
	PifMgr_SetProperties	<b>0000000180001560</b>	26
	PifMgr_CloseProperties	<b>0000000180001590</b>	27
	ILIsEqual	<b>00000001800015C0</b>	28
	DragFinish	<b>0000000180001440</b>	29
	<b>DllEntryPoint</b>	<b>00000001810F75B4</b>	<b>[main entry]</b>

### Export Functions

From this list of 30 export functions, two of them *DllEntryPoint* and *ILCreateFromPath* caught my attention. After looking through the two functions, I had determined that the *ILCreateFromPath* function contains the “actions” that we are interested in.

### Obfuscation

While scrolling through the *ILCreateFromPath* function, I noticed a pattern:

```

push    rbx
sub     rsp, 30h
lea    rcx, aPrtPP0tnpnWslly ; "]prt-.p,^p,0tnpN.,wSlyowp,b"
mov    rbx, rdx
call   sub_180001010
mov    rcx, [rbx]
lea    rdx, kernel_20
call   rax
mov    cs:qword_18110E598, rax
test   rax, rax
jz     loc_180001AD2
    
```

## Encoded String

Noticed that the value “]prt-.p,^p,0tnpN.,wSlyowp,b” in the variable *aPrtPP0tnpnWsly* was lea into *rcx* followed by calling the sub function *sub\_180001010* then followed by a *call rax*. From my analysis, the function *sub\_18001010* consist of two parts.

### 1 — Decode the string

First it will decode a string that is passed in as argument which in this case is the value in the variable *aPrtPP0tnpnWsly*.

```

loc_180001100:
movsx    ecx, byte ptr [r9+r11]
lea      r11, [r11+1]
add      ecx, 1Ch
mov      eax, 1948B0FDh
imul    ecx
inc      edi
sar      edx, 3
mov      eax, edx
shr      eax, 1Fh
add      edx, eax
imul    eax, edx, 51h ; 'Q'
sub      ecx, eax
movsxd  rax, edi
add     cl, 2Ah ; '*'
mov     [r11-1], cl
cmp     rax, r8
jnb     short loc_180001100

```

### Decode Section

This function decodes the characters by applying the concept of Substitution cipher where it takes the ASCII value of each character, add 28 follow by mod 81 and finally add 42. This is the formula for the substitution cipher that I had just describe:  $plain\_text = (cipher\_text + 28) \% 81 + 42$ . Thus, the value of variable *aPrtPP0tnpnWsly* decodes into *RegisterServiceCtrlHandlerW* which is a Win32 API.

Using this formula, I wrote a simple python script to do the decryption and ran it on all the strings found in the same data section as *aPrtPP0tnpnWsly* and here are the decrypted strings:

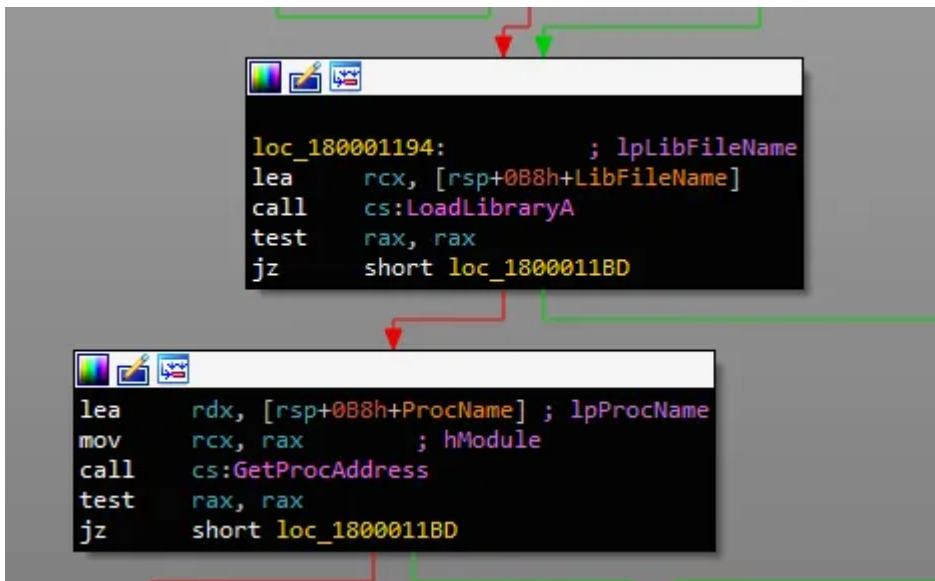
Press enter or click to view image in full size

Obfuscated	De-obfuscated
vp,ypw>=9oww	kernel32.dll
Rp.^3-.pxOt,pn.z,3b	GetSystemDirectoryW
N,pl.p[,znp--b	CreateProcessW
Rp._s,plONzy.p2.	GetThreadContext
^p._s,plONzy.p2.	SetThreadContext
at,./lwLwznp2	VirtualAllocEx
b,t.p[,znp--Xpxz,3	WriteProcessMemory
]p-/xp_s,plO	ResumeThread
blt.Qz,^tyrwpZmupn.	WaitForSingleObject
lo0!*t>=9oww	advapi32.dll
]prt-.p,^p,0tnpN.,wSlyowp,b	RegisterServiceCtrlHandlerW
^p.^p,0tnp^.l./-	SetServiceStatus
zwp>=9oww	ole32.dll
NzN,pl.pR/to	CoCreateGuid
,*n,.?9oww	rpcrt4.dll
`/to_z^.,tyrb	UuidToStringW

### Decoded String

Looking at the de-obfuscated strings, it seems like they are hiding function calls in strings and decode them during runtime so that we cannot most of its capabilities just from looking at imports table. From the list of the De-obfuscated strings, we can see that some of the capabilities of this malware includes creating thread and writing into memory.

### 2 — GetProcAddress



### GetProcAddress

Once the string is decoded, the function will then call *LoadLibraryA* and *GetProcAddress* before returning the address of the call.

## Get BlueMonkey’s stories in your inbox

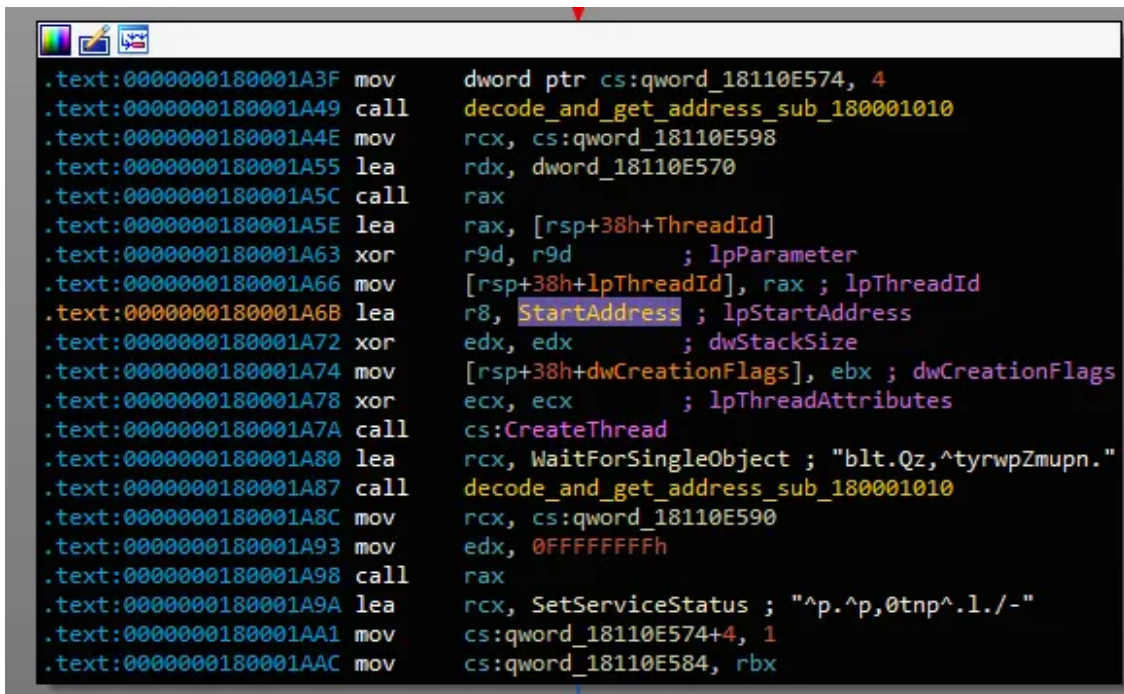
Join Medium for free to get updates from this writer.

Remember me for faster sign in

Alright, now that we know that this malware has the ability to hide its function calls what's next?

## Threading and New Processes

Following down the *ILCreateFromPath*, I saw that the malware creates a new thread to execute the function called *StartAddress*.



```
.text:00000000180001A3F mov     dword ptr cs:qword_18110E574, 4
.text:00000000180001A49 call    decode_and_get_address_sub_180001010
.text:00000000180001A4E mov     rcx, cs:qword_18110E598
.text:00000000180001A55 lea    rdx, dword_18110E570
.text:00000000180001A5C call    rax
.text:00000000180001A5E lea    rax, [rsp+38h+ThreadId]
.text:00000000180001A63 xor     r9d, r9d ; lpParameter
.text:00000000180001A66 mov     [rsp+38h+lpThreadId], rax ; lpThreadId
.text:00000000180001A6B lea    r8, StartAddress ; lpStartAddress
.text:00000000180001A72 xor     edx, edx ; dwStackSize
.text:00000000180001A74 mov     [rsp+38h+dwCreationFlags], ebx ; dwCreationFlags
.text:00000000180001A78 xor     ecx, ecx ; lpThreadAttributes
.text:00000000180001A7A call    cs:CreateThread
.text:00000000180001A80 lea    rcx, WaitForSingleObject ; "blt.Qz,^tyrwpZmupn."
.text:00000000180001A87 call    decode_and_get_address_sub_180001010
.text:00000000180001A8C mov     rcx, cs:qword_18110E590
.text:00000000180001A93 mov     edx, 0FFFFFFFFh
.text:00000000180001A98 call    rax
.text:00000000180001A9A lea    rcx, SetServiceStatus ; "^p.^p,0tnp^.1./- "
.text:00000000180001AA1 mov     cs:qword_18110E574+4, 1
.text:00000000180001AAC mov     cs:qword_18110E584, rbx
```

Create New Thread

So the only thing to do is to follow and look at what does the *StartAddress* function do.

Looking through the *StartAddress* function, the first thing that caught my eyes is a variable containing the string “%s\\dllhost.exe /Processid:{%s}” followed by *CreateProcessW*. It looks like the malware is trying to create a *dllhost* process.

```
.text:0000000180001698 lea rdx, aSDllhostExePro ; "%s\\dllhost.exe /Processid:{%s}"
.text:000000018000169F lea rcx, [rbp+6F0h+var_220] ; LPWSTR
.text:00000001800016A6 call cs:wprintfW
.text:00000001800016AC xorps xmm0, xmm0
.text:00000001800016AF lea rcx, [rbp+6F0h+var_6F0] ; void *
.text:00000001800016B3 xor eax, eax
.text:00000001800016B5 xor edx, edx ; Val
.text:00000001800016B7 mov r8d, 4D0h ; Size
.text:00000001800016BD mov [rbp+6F0h+var_720], rax
.text:00000001800016C1 movups [rsp+7F0h+var_780], xmm0
.text:00000001800016C6 mov [rsp+7F0h+var_788], rax
.text:00000001800016CB movups [rbp+6F0h+var_770], xmm0
.text:00000001800016CF movups [rbp+6F0h+var_760], xmm0
.text:00000001800016D3 movups [rbp+6F0h+var_750], xmm0
.text:00000001800016D7 movups [rbp+6F0h+var_740], xmm0
.text:00000001800016DB movups [rbp+6F0h+var_730], xmm0
.text:00000001800016DF movups [rsp+7F0h+var_798], xmm0
.text:00000001800016E4 call memset
.text:00000001800016E9 lea rcx, CreateProcessW ; "N,pl.p[,znp--b"
.text:00000001800016F0 call decode_and_get_address_sub_180001010
.text:00000001800016F5 xor edi, edi
.text:00000001800016F7 lea rcx, [rsp+7F0h+var_798]
.text:00000001800016FC mov [rsp+7F0h+var_7A8], rcx
.text:0000000180001701 lea rdx, [rbp+6F0h+var_220]
.text:0000000180001708 lea rcx, [rsp+7F0h+var_780]
.text:000000018000170D xor r9d, r9d
.text:0000000180001710 mov [rsp+7F0h+var_7B0], rcx
.text:0000000180001715 xor r8d, r8d
.text:0000000180001718 mov [rsp+7F0h+var_7B8], rdi
.text:000000018000171D xor ecx, ecx
.text:000000018000171F mov [rsp+7F0h+var_7C0], rdi
.text:0000000180001724 mov [rsp+7F0h+var_7C8], 4
.text:000000018000172C mov dword ptr [rsp+7F0h+var_7D0], edi
.text:0000000180001730 call rax
```

Create New Process

After calling *CreateProcessW*, the malware then proceeds to call *VirtualAllocEx* followed by *WriteProcessMemory*.

```
.text:0000000180001750 lea rcx, VirtualAllocEx ; "at,./lwLwznP2"
.text:0000000180001757 call decode_and_get_address_sub_180001010
.text:000000018000175C mov rcx, qword ptr [rsp+7F0h+var_798]
.text:0000000180001761 xor edx, edx
.text:0000000180001763 mov r9d, 1000h
.text:0000000180001769 mov dword ptr [rsp+7F0h+var_7D0], 40h ; '@'
.text:0000000180001771 mov r8d, 5000h
.text:0000000180001777 call rax
.text:0000000180001779 mov rbx, rax
.text:000000018000177C test rax, rax
.text:000000018000177F jz loc_180001956
```

```
.text:0000000180001785 db 66h, 66h
.text:0000000180001785 nop word ptr [rax+rax+00000000h]
```

```
.text:0000000180001790
.text:0000000180001790 loc_180001790:
.text:0000000180001790 lea rcx, WriteProcessMemory ; "b,t,p[,znp--Xpxz,3"
.text:0000000180001797 call decode_and_get_address_sub_180001010
.text:000000018000179C lea rcx, [rbp+6F0h+var_710]
.text:00000001800017A0 mov r9d, 5000h
.text:00000001800017A6 mov [rsp+7F0h+var_7D0], rcx
.text:00000001800017AB lea r8, sub_181064570
.text:00000001800017B2 mov rcx, qword ptr [rsp+7F0h+var_798]
.text:00000001800017B7 mov rdx, rbx
.text:00000001800017BA call rax
```

### Allocate And Write To Memory

From the above code, we can see that the malware used the *WriteProcessMemory* function to write the function *sub\_181064570* into *dllhost* process created earlier. Although this seems to be a common process injection, it matches the checkpoint's report, where it mentions that it injects itself to another process such as *rundll32.exe* and *dllhost.exe*.

### Decoding Embedded Data

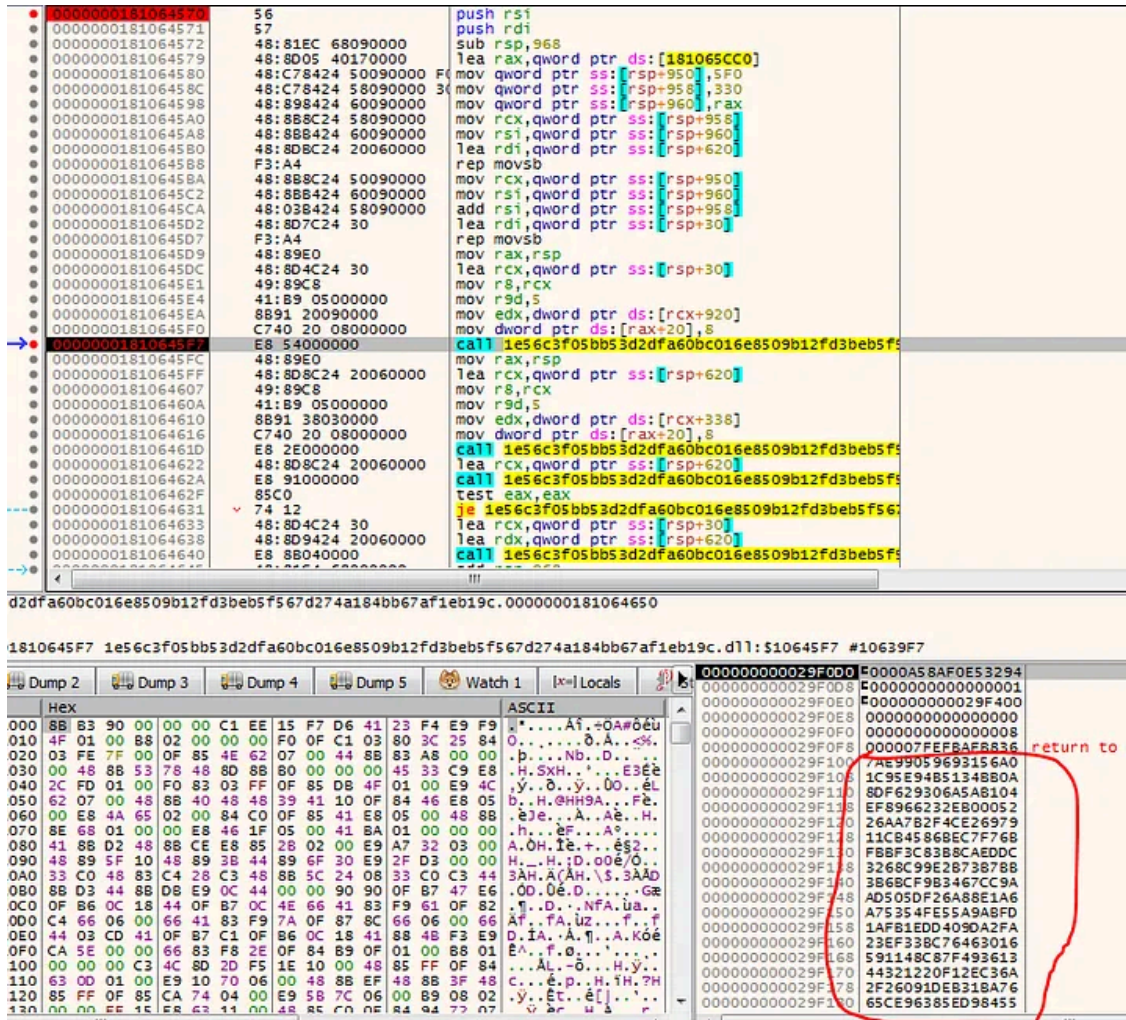
Finally, we have reached the part where we can see what this malware actually wants to do! This is how the first few lines of the function written into the process's memory looks like:

```
.text:0000000181064570
.text:0000000181064570 push rsi
.text:0000000181064571 push rdi
.text:0000000181064572 sub rsp, 968h
.text:0000000181064579 lea rax, qword_181065CC0
.text:0000000181064580 mov [rsp+978h+var_28], 5F0h
.text:000000018106458C mov [rsp+978h+var_20], 330h
.text:0000000181064598 mov [rsp+978h+var_18], rax
.text:00000001810645A0 mov rcx, [rsp+978h+var_20]
.text:00000001810645A8 mov rsi, [rsp+978h+var_18]
.text:00000001810645B0 lea rdi, [rsp+978h+var_358]
.text:00000001810645B8 rep movsb
.text:00000001810645BA mov rcx, [rsp+978h+var_28]
.text:00000001810645C2 mov rsi, [rsp+978h+var_18]
.text:00000001810645CA add rsi, [rsp+978h+var_20]
.text:00000001810645D2 lea rdi, [rsp+978h+var_948]
.text:00000001810645D7 rep movsb
.text:00000001810645D9 mov rax, rsp
.text:00000001810645DC lea rcx, [rsp+978h+var_948]
.text:00000001810645E1 mov r8, rcx
```

### Write to Memory

It looks like the malware copying two sets of data located at `qword_181065CC0` into the memory.

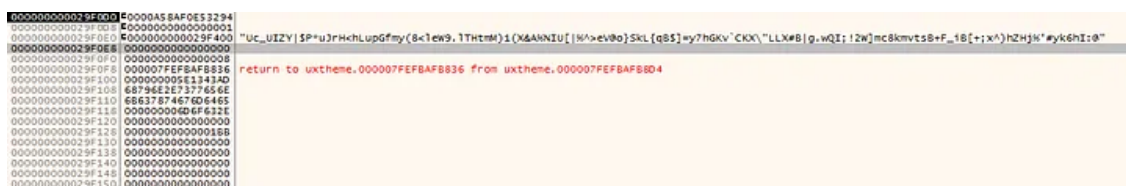
Press enter or click to view image in full size



### Encoded Blob in Memory

After copying the data into the memory, the malware calls a function which will decode the data.

Press enter or click to view image in full size



### First Decoded Blob

The first blob of data contains the URL of the C2 server “`news.nyhedmgtxck.com`” and a string of characters which doesn’t seem to be used in any part of the execution.

```
00000000773A0000 kernel32.00000000773A0000
00000000773B7070 kernel32.00000000773B7070
00000000773B67A0 kernel32.00000000773B67A0
00000000773B1260 kernel32.00000000773B1260
00000000773B5B50 kernel32.00000000773B5B50
00000000773A4F60 kernel32.00000000773A4F60
00000000773C3630 kernel32.00000000773C3630
000000007744A493 "NTDLL.RtlExitUserProcess"
00000000773C2B20 kernel32.00000000773C2B20
00000000773B65E0 kernel32.00000000773B65E0
00000000773B64A0 kernel32.00000000773B64A0
00000000773B7700 kernel32.00000000773B7700
00000000773A80A0 kernel32.00000000773A80A0
00000000773EC5B0 kernel32.00000000773EC5B0
00000000774355E0 kernel32.00000000774355E0
00000000773B14E0 kernel32.00000000773B14E0
0000000077438800 kernel32.0000000077438800
00000000773A2D50 kernel32.00000000773A2D50
00000000773B7210 kernel32.00000000773B7210
0000000077438D40 kernel32.0000000077438D40
00000000773B6580 kernel32.00000000773B6580
00000000773B9460 kernel32.00000000773B9460
```

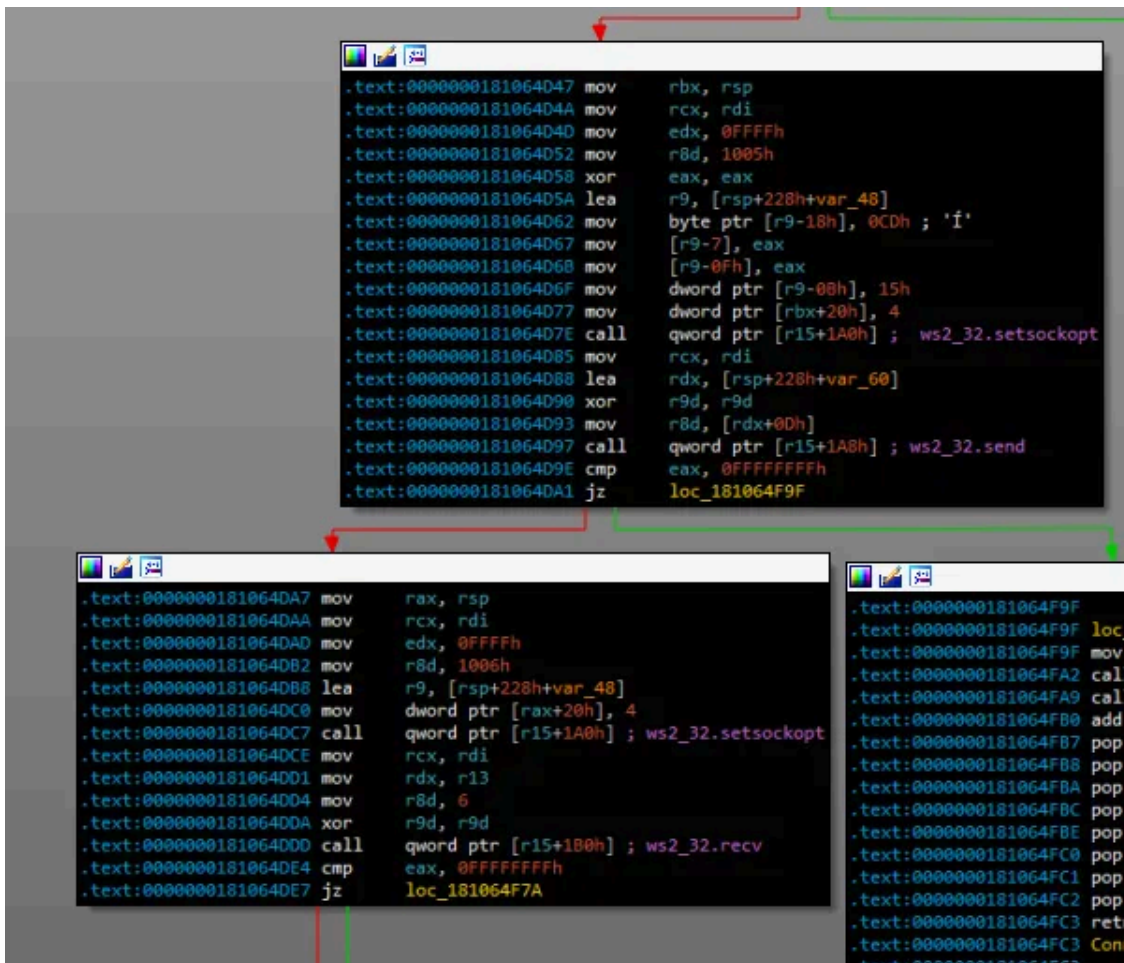
### Second Decoded Blob

And the second blob of data contains the imports table which the malware will use in the next phase of its activity. Wait a minute... does the 2 blobs of data sounds familiar?

From checkpoint's report on aria-body loader, they mentioned that one of the functionality of the loader is to decrypt two blobs of data into an Import Table and a loader configuration.

### Download and execute payload

By using the decoded import tables, the malware attempts to connects to the C2 URL to download a file.



### Download Payload

To this point, it actually matches the points mentioned in Check Point's report where Aria-body contact the embedded / calculated C&C address in order to download retrieve the next stage payload.

Too bad for us, the URL has already been sinkhole. Therefore, I am not be able download the sample for analysis ):

It's not the end yet! Although I am not able to analyze the next stage payload, I am still able to see what this loader does before passing control to the next stage payload :D

Once the payload is downloaded, the malware will first decode the payload with a XOR function. The decoded payload will then reside only in the memory. Which suggest that it could be a file-less malware.

```
.text:00000000181064E60  
.text:00000000181064E60 loc_181064E60:  
.text:00000000181064E60 mov rcx, rdi  
.text:00000000181064E63 call qword ptr [r15+188h] ; ws2_32.closesocket  
.text:00000000181064E6A mov rax, rsp  
.text:00000000181064E6D mov rcx, rsi  
.text:00000000181064E70 movzx r9d, byte ptr [r13+0]  
.text:00000000181064E75 mov rdx, rsi  
.text:00000000181064E78 movzx r13d, byte ptr [r13+1]  
.text:00000000181064E7D xor r9d, r13d  
.text:00000000181064E80 mov r8d, ebx  
.text:00000000181064E83 mov dword ptr [rax+20h], 0  
.text:00000000181064E8A call xor_decode_payload_sub_1810657D0  
.text:00000000181064E8F test ebp, ebp  
.text:00000000181064E91 jnz short loc_181064ED8
```

```
.text:00000000181064E93 mov rcx, rsi  
.text:00000000181064E96 mov rdx, r15  
.text:00000000181064E99 call Copy_data_to_mem_loc_181065810  
.text:00000000181064E9E mov rbp, rax  
.text:00000000181064EA1 test rbp, rbp  
.text:00000000181064EA4 jz short loc_181064ED8
```

Decode Payload and Copy to Memory

Next, the malware then calls a function which checks if the payload contains the magic number “MZ” and “PE”.

Press enter or click to view image in full size

```
.text:000000001810648D0  
.text:000000001810648D0 sub rsp, 58h  
.text:000000001810648D4 mov r9, rcx  
.text:000000001810648D7 mov r10d, edx  
.text:000000001810648DA movzx eax, word ptr [r9]  
.text:000000001810648DE cmp eax, 5A4Dh ; check if memory starts with MZ  
.text:000000001810648E3 jnz loc_1810649AB
```

```
.text:000000001810648E9 movsxd rax, dword ptr [r9+3Ch]  
.text:000000001810648ED cmp dword ptr [rax+r9], 4550h ; check for PE header  
.text:000000001810648F5 jnz loc_1810649AB
```

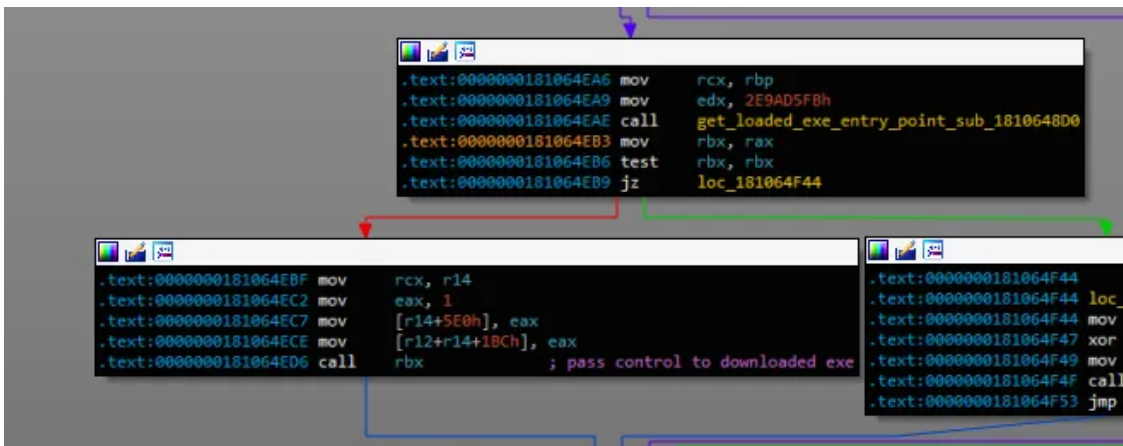
Check for PE and Section Header

Once verified, the malware will finally get the entry point to the payload by calculating the djb2 hash of the payload’s export and comparing it with 0x2E9AD5FB. Without the second stage payload, I am unable to determine what is the export name based on that hash.

```
result = 5381i64;  
for ( i = (char)*a1; *a1; i = (char)*a1 )  
{  
    if ( a3 && i >= 0x61 )  
        i -= 32;  
    result = i + 33 * (_DWORD)result;
```

DJB2 Hash

Finally, the malware then passes the execution control to the payload.



The way this malware get the entry point also matches what Check Point had described where the loader loads and execute an exported function of the DLL — calculated using djb2 hashing algorithm.

## Conclusion

Phew... Finally! We've reached the conclusion~~ v^^v

From the analysis, this malware looks like a loader which will download a payload from the C2 and execute the payload on the memory. The capabilities of this sample is very similar to the Aria-body loader that is described by Check Point where 5 out of the 7 points matching the analysis. I am unable to determine if this sample “establishes persistence via startup folder or run registry” and the “utilization of the DGA algorithm”. Putting the capabilities aside, I had look through the sample with the hash “40c49ecbe1b7bdodbb935138661b6ca4” mentioned in Check Point’s report and code wise, it looks vastly different from this sample.

Noticed that up to this point, I have only talked about the analysis of one of the samples. Well, I had done the analysis on both of the sample and in regards of the code executions, they are the same. The only difference between the two sample in regards to what is relevant to the execution and its function, is that the C2 string and the string of character in the first blob of data is different. Instead of going to “news[.]nyhedmgtxck[.]com”, the C2 of the other sample is “www[.]etnwtmrkh[.]com” both of which are sinkholed.

Therefore, based on the capabilities, am I right to say that this could be a variant of Aria-Body loader?

---

Source: <https://medium.com/insomniacs/aria-body-loader-is-that-you-53bdd630f8a1>