Analyzing OBSCURE#BAT: Threat Actors Lure Victims into Executing Malicious Batch Scripts to Deploy Stealthy Rootkits

Blog

Threat Research

Securonix Threat Research Security Advisory

By Securonix Threat Research: Den Iuzvyk, Tim Peck

Mar 13, 2025

tldr:

The Securonix Threat Research team has been tracking a stealthy malware campaign leveraging social engineering and deceptive file downloads to trick users into executing heavily obfuscated code. This infection ultimately deploys a user-mode rootkit that manipulates system processes and registry entries to evade detection and maintain persistence.

Tracked as OBSCURE#BAT, our team recently identified a malicious campaign that relies on batch script execution to kick off a deep-nested chain of malware resulting in a user-mode rootkit that we identified as r77 rootkit which has the ability to cloak or mask any file, registry key or task beginning with a specific prefix. It has been targeting users by either masquerading as legitimate software downloads or via fake captcha social engineering scams.

The malware that gets installed leverages a user-mode rootkit to establish persistence and evade detection on compromised systems. The infection begins with highly obfuscated batch scripts, which execute a series of environment variable manipulations and PowerShell commands to deploy the next-stage payload. The malware stores obfuscated scripts in the Windows Registry and ensures execution via scheduled tasks, allowing it to run stealthily in the background. Additionally, it modifies system registry keys to register a fake driver (`ACPIx86.sys`), further embedding itself into the system.

What makes this malware particularly dangerous is its ability to hide files, registry entries, and running processes using user-mode API hooking. Any artifacts matching a specific pattern (`$nya-`) become invisible to standard Windows tools like Task Manager, Explorer, and shell commands such as “dir” to list directory contents. The malware also interacts with critical system processes allowing it to embed deeper into legitimate processes and services. Security logs indicate it is capable of deleting evidence of its activity while remaining undetectable by conventional methods.

Key observations

- Attackers are using several methods to get users to execute malicious batch scripts
- Obfuscated batch script initiates infection by executing dynamic PowerShell commands.
- Persistence established through scheduled tasks and registry injected PowerShell scripts.
- Files, registry entries and processes hidden using API hooking, preventing detection by users and standard tools.
- Attempts to register a fake driver (`ACPIx86.sys`) via the Windows Registry for further persistence.
- Signs of process injection into legitimate Windows processes.
- The infected system prevents visibility of any files, processes or registry keys matching the “`$nya-`” prefix.
- PowerShell execution delays and hidden processes suggest advanced rootkit behavior.
- Final payloads include a user-mode rootkit and another system mode rootkit which gets executed as a service
- Malware regularly monitors for clipboard and command history and saves them into hidden files for exfiltration

Initial infection

Code execution begins when a user unknowingly or knowingly executes a malicious batch file. We observed several methods used by the attackers:

Delivery method 1: Fake captcha to code execution:

The idea behind this social engineering scam is to trick the user into executing malicious code by masquerading as a legitimate Cloudflare captcha. Many of the captchas we observed appeared to be behind typosquatted domains such as hxxps://cooinbase[.]net in one example we observed.

We've all seen the dialog boxes and clicked the fire hydrants and crosswalks dozens of times. Attackers pray on these rather innocuous tasks in order to trick the user into executing malicious code instead.[a]

As seen in the figure below, when a user clicks the "Verify you are human" checkbox, code is copied via JavaScript to the user's keyboard. Instructions follow to hit Windows + R to open the run dialog box, paste in the just copied contents, and then hit OK.

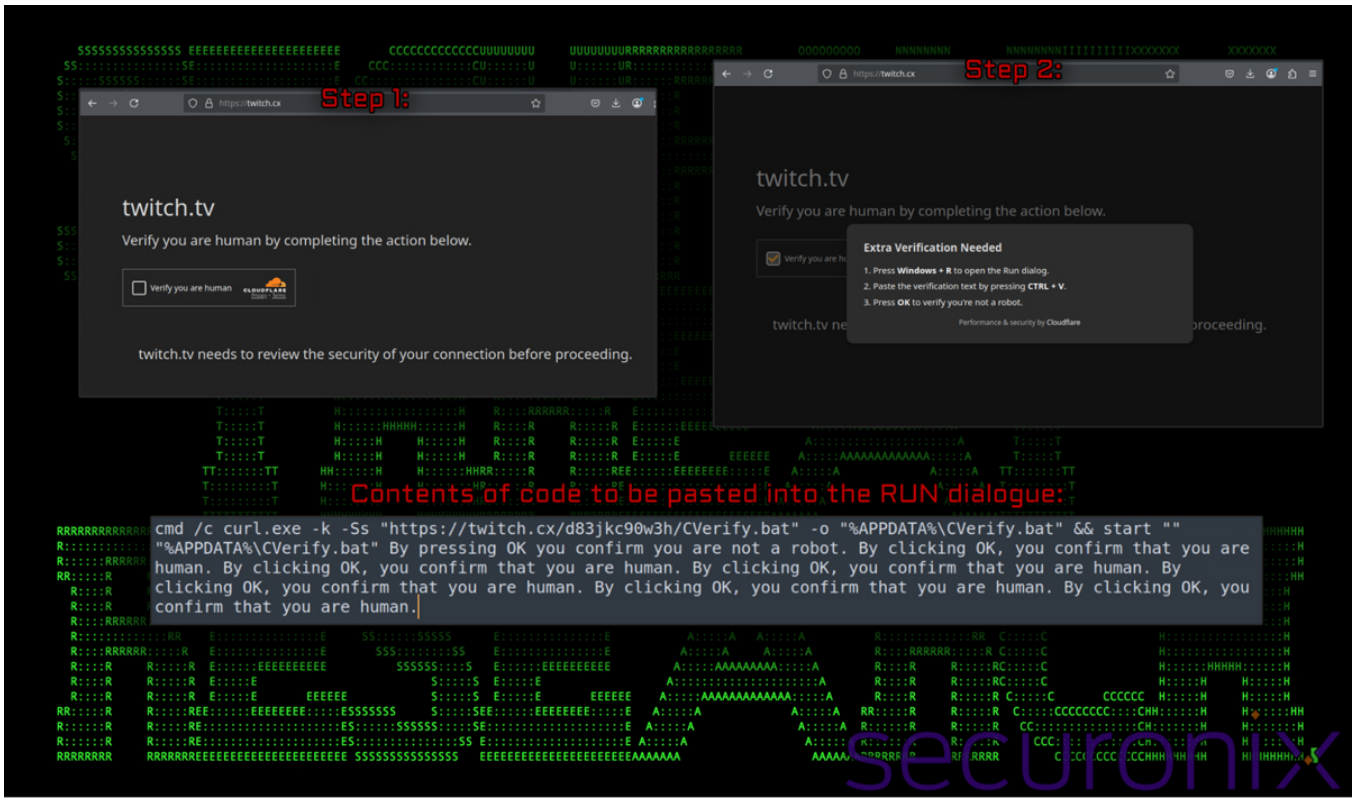

Figure 1: Initial code execution

Delivery method 2: Masquerading as legitimate tooling

The attackers also opted to spread the malware by masquerading as legitimate tools and software. We observed several file names that revealed some insight as to the tools they were masquerading as. As you're see further down in the campaign, SIP (VoIP) software, the Tor Browser, Adobe software and other network and message client software. Unfortunately we were not able to obtain any information as to the download links, though we speculate that malvertising could have been an initial starting point.

Initial code execution analysis

While we were able to gather many of the attackers .bat files, we'll focus on one in particular. This batch file was a part of a downloaded zip file "sip.zip" which contained a malicious batch file named "install.bat". The rest of the files appeared to be legitimate and non-malicious.

The install.bat file serves as the initial execution point for the malware, setting off a series of actions that establish persistence and deploy the embedded RAT malware and rootkit. When a user clicks on install.bat, it calls PowerShell to execute obfuscated commands, which in turn write additional scripts or payloads to disk, modify the registry, and schedule tasks for persistence, which we'll dive into further down.

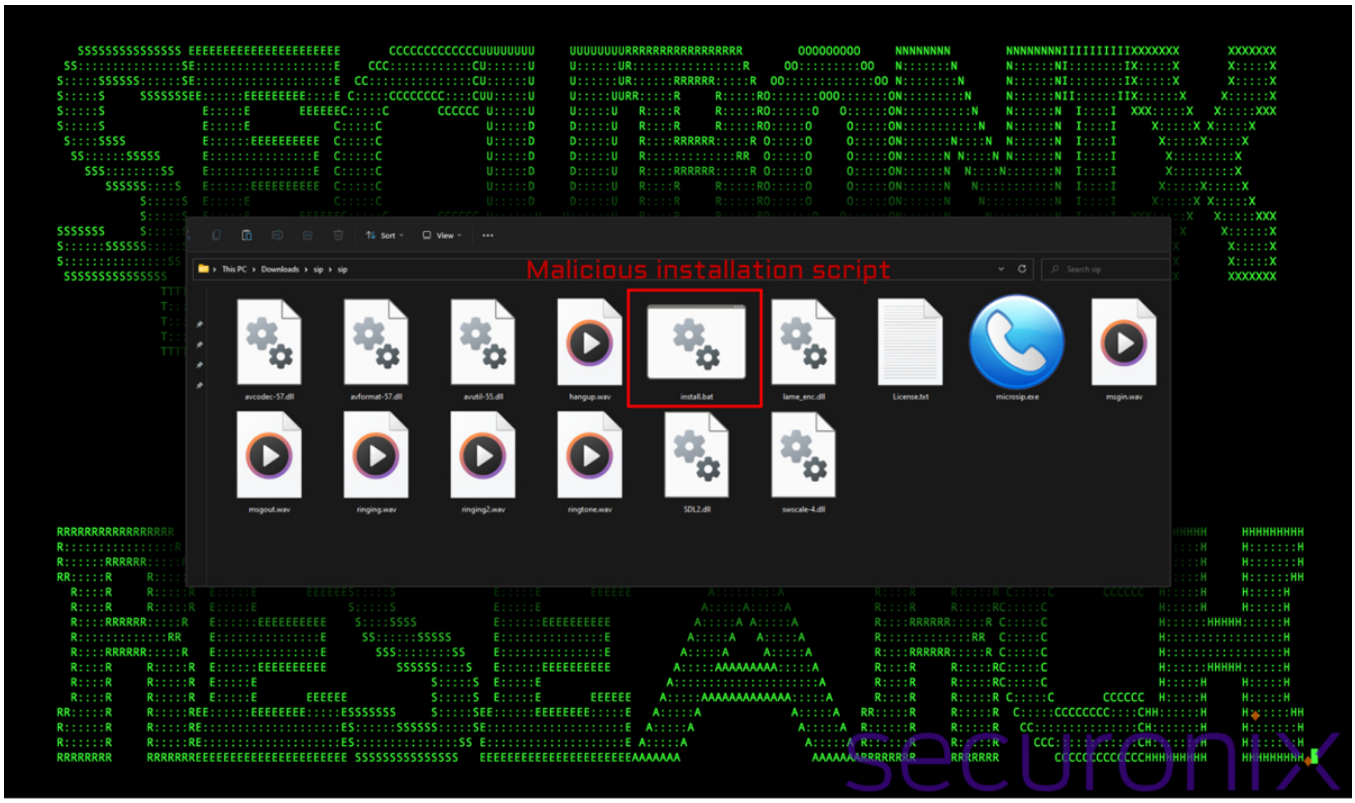

Figure 2: sip.zip file contents – install.bat

The batch file is highly obfuscated, making static analysis difficult. Because of this, detections were minimal. At the time of writing, we analyzed a few batch files identified as part of the OBSCURE#BAT campaign and they all scored very low (0–2 detections on VirusTotal).

Figure 3: VirusTotal detections for install.bat

At first glance, the batch scripts used in the campaign are massive. In this example, we'll analyze "install.bat" which stood at over 6MB in size.

After opening the batch file in a text editor, the reason for its size becomes apparent. The script contains thousands of lines of variables defined or concatenated to thwart analysis.

The script essentially consists of two main components:

1. The obfuscated batch script code which we'll dive into now
2. A very large string of random characters at the last line of the script – an embedded payload we'll discuss later.

The obfuscated batch script relies heavily on padding the code with useless variables (defined by %…%) and character substitutions. Strings of commands are built from hundreds of predefined variables which, when parsed, execute the next-stage payload.

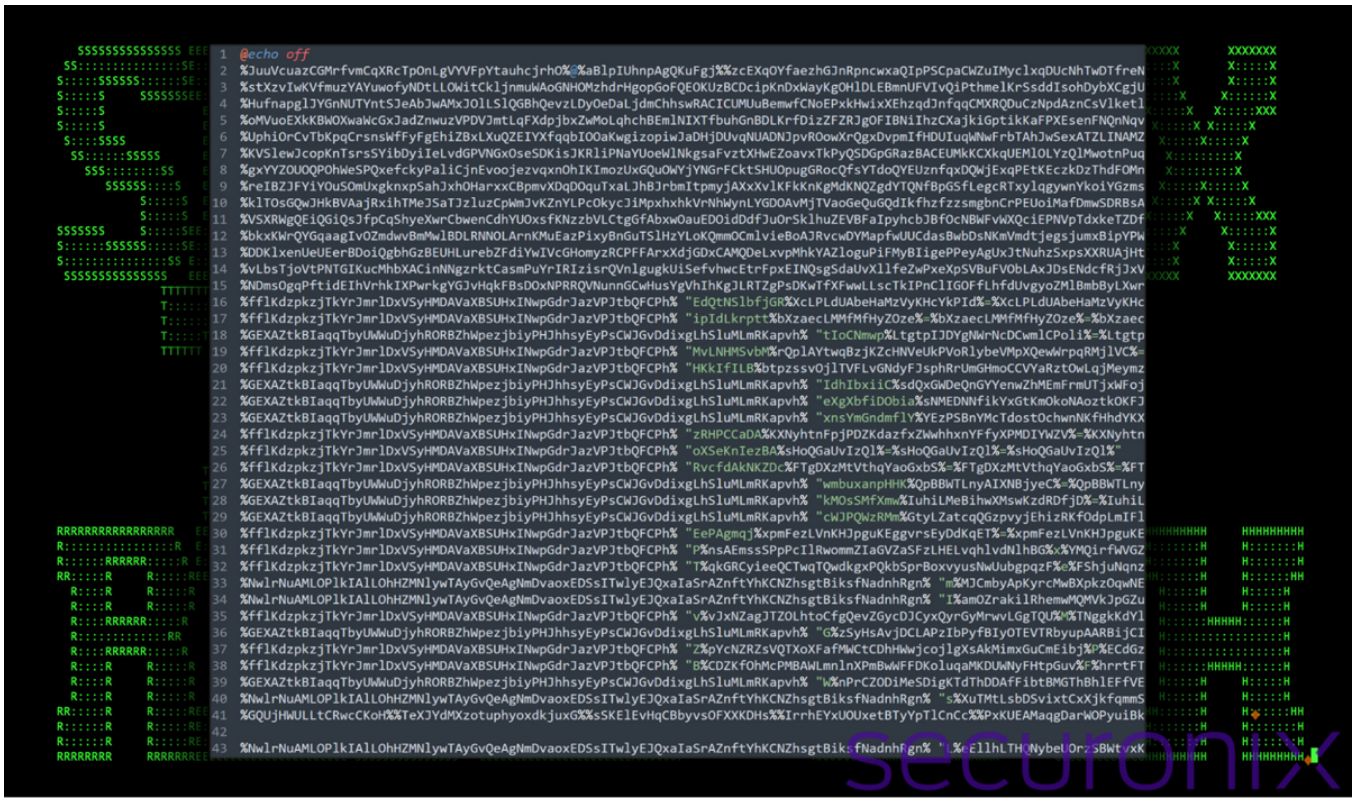

Figure 4: Obfuscated contents of install.bat

Deobfuscating the contents of install.bat took some time, but once we identified the variable sequences that built up the SET command, it started to come together. We replaced those sequences with the actual SET command.

Once done, the remaining lines consisted of concatenated variable strings. By inserting an ECHO command before them, we could print the contents instead of executing them.

Figure 5: Deobfuscation process of install.bat

Once initial execution is completed, the install.bat file deletes itself, hiding malicious artifacts while the next-stage PowerShell code is executed.

PowerShell Execution

The PowerShell executed in the next stage performs key tasks. First, it performs odd system checks using `Get-WmiObject Win32_DiskDrive` to query the system's physical disk drive information.

The command retrieves details about all connected storage devices, while `Select-Object -ExpandProperty` Model extracts only the model names. The output is then piped to `findstr /i 'WDS100T2B0A`, which performs a case-insensitive search for the SSD model WDS100T2B0A (a Western Digital Blue SSD, 1TB SATA 2.5″).

If this specific SSD is found, the script proceeds with additional checks (such as verifying if the F:\ drive is empty) – an anti-analysis technique to detect forensic environments. However, we could not determine why execution would be halted based on these checks.

Figure 6: PowerShell execution – disk drive checks

Lastly, the script either exits with error code 900 if the SSD (WDS100T2B0A) is detected and F:\ is empty, or continues normally.

PowerShell Obfuscation

The PowerShell script employs multiple obfuscation techniques to evade detection. One primary method is string obfuscation using concatenation and character replacements – evident from the extensive use of `.Replace()` functions.

For example, one line uses PowerShell invoke expressions to run:

```
'$rFUJ=
[nKSnKynKsnKtnKemnK.nKSenKcnKunKrinKtnKynK.nKCnKrnKynKpnKtnKonKgnKrnKanKpnKhynK.nKAnKesnK]nK::nKCrnKenKanKtnKe(nK)nK;'.R
eplace('nK', '');
```

After removing “nK” characters, the command becomes:

```
$rFUJ = [System.Security.Cryptography.Aes]::Create();
```

This reveals that AES encryption is used to decrypt and execute an encoded payload. The script also employs function aliasing and variable obfuscation to further mask its true purpose.

Figure 7: PowerShell obfuscation in Invoke-Expressions

1. -InformationAction Ignore: This suppresses informational messages that could reveal execution details.
2. -Verbose: Used for debugging, though its effect is muted unless VerbosePreference is set to Continue.
3. -Debug: Enables detailed debugging output, which is unusual in malware and may be used to blend in with normal script behavior.

Embedded payload – Decrypt, decode and execute

With the code cleaned up for readability, we see that the PowerShell script implements a multi-layered decryption process on an embedded payload. The huge string at the end of “install.bat” is read in by PowerShell, decoded, extracted, and then executed.

The decryption process involves three key functions as shown in the figure below: `taBy`, `mDmV`, and `EHSd`, which work together to extract and execute the final payload.

Figure 8: Cleaned up PowerShell – decryption process of embedded payloads

The `EHSd` function executes the final stage of the decoded payload. It takes two arguments, `$iMqB` and `$GkzP`, and processes `$iMqB` which stores the final decoded payload.

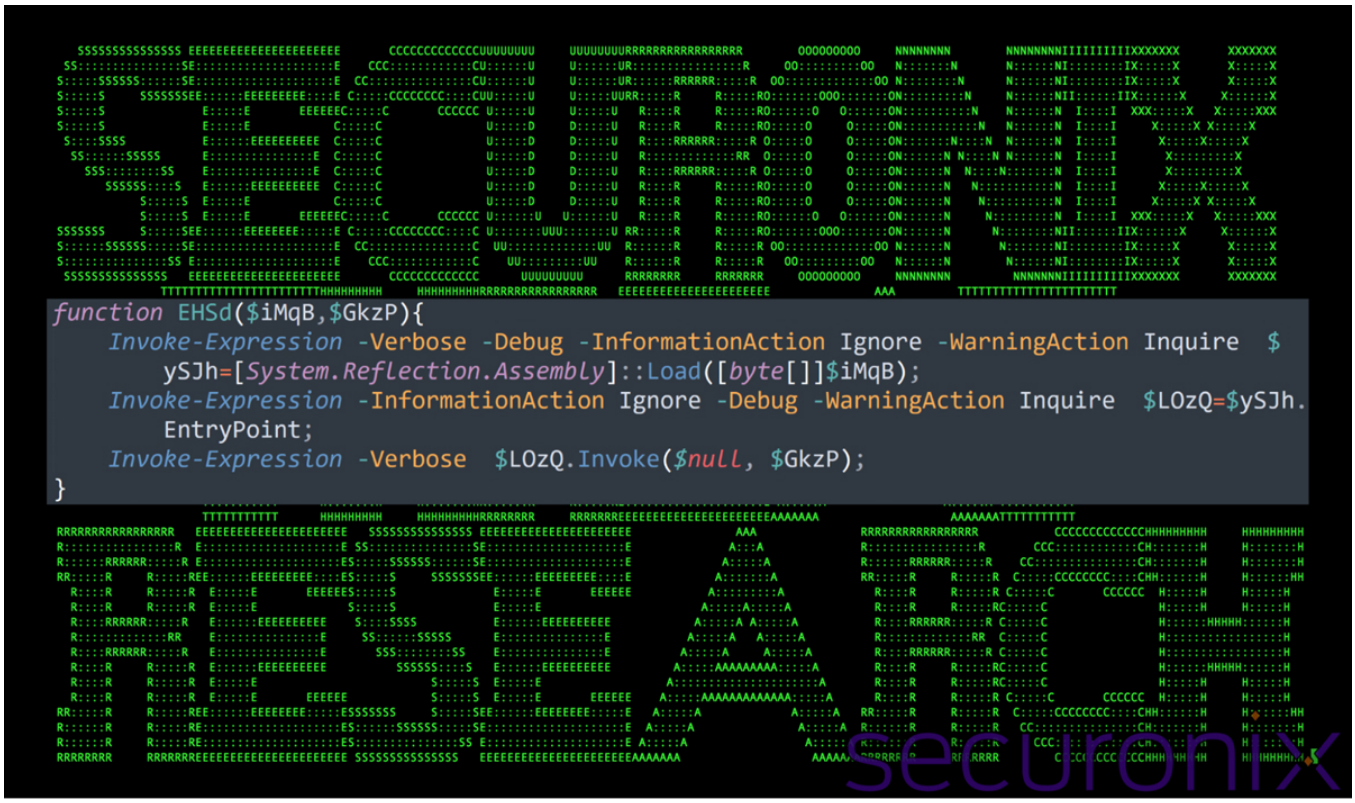

Figure 9: PowerShell execution of embedded payload

The function initializes a byte array using a series of `.Replace()` functions to remove obfuscation markers, assigns the result to `$ySJh`, and finally leverages `[System.Reflection.Assembly]` to execute the decoded payload directly in memory.

We'll dive into this payload further in the next section.

Windows registry injection

The final task of the script is injecting payloads into the Windows registry. In our case, the malware creates values under the `HKLM:\SOFTWARE\OOhhhm` key as part of the `srh` function.

Figure 10: PowerShell execution – Windows registry injection

These registry values include `XFtkNtIDKz` and `YvrtRvSWmsUlsKkNV`, which store AES and IV keys for decoding further payloads. These keys were also used to decode the initial payload that is reflectively loaded.

Additional keys were created – for example, the first key in "`HKU\S-1-5-21-...<SID>...-1001\Environment\onimaiuc`" contained a copy of the malicious PowerShell code from install.bat.

The registry payload is stored in "`HKLM\SOFTWARE\OOhhhm=\NbPqJZCqG`", which is dynamically generated.

Figure 11: Log analysis – PowerShell and encrypted payloads injected into the Windows Registry

Similar to install.bat, the PowerShell executed from the registry reads in encrypted payloads in chunks, replaces certain characters, decrypts them, and executes via reflection.

Even stealthier persistence in Windows registry

The malware employs advanced techniques to hide in the Windows registry to evade detection by tools like regedit. We logged additional keys and values created at:

- `HKLM\SOFTWARE\$nya-dll64`
- `HKLM\SOFTWARE\$nya-dll32`
- `HKLM\SOFTWARE\$nya-config\paths\C:\Windows\system32\drivers\ACPIx86.sys`
- `HKLM\SOFTWARE\$nya-config\service_names\ACPIx86`
- `HKLM\SOFTWARE\$nya-config\pid\3499344`

However, when browsing regedit, the created key `$nya-dll64` did not exist!

This is achieved via rootkit functionality that manipulates visible system data. The malware likely employs API hooking, process injection, and registry tampering to achieve stealth and persistence.

For example, renaming the file with the PowerShell command Rename-Item makes hidden files visible because the filter is hardcoded to specific patterns.

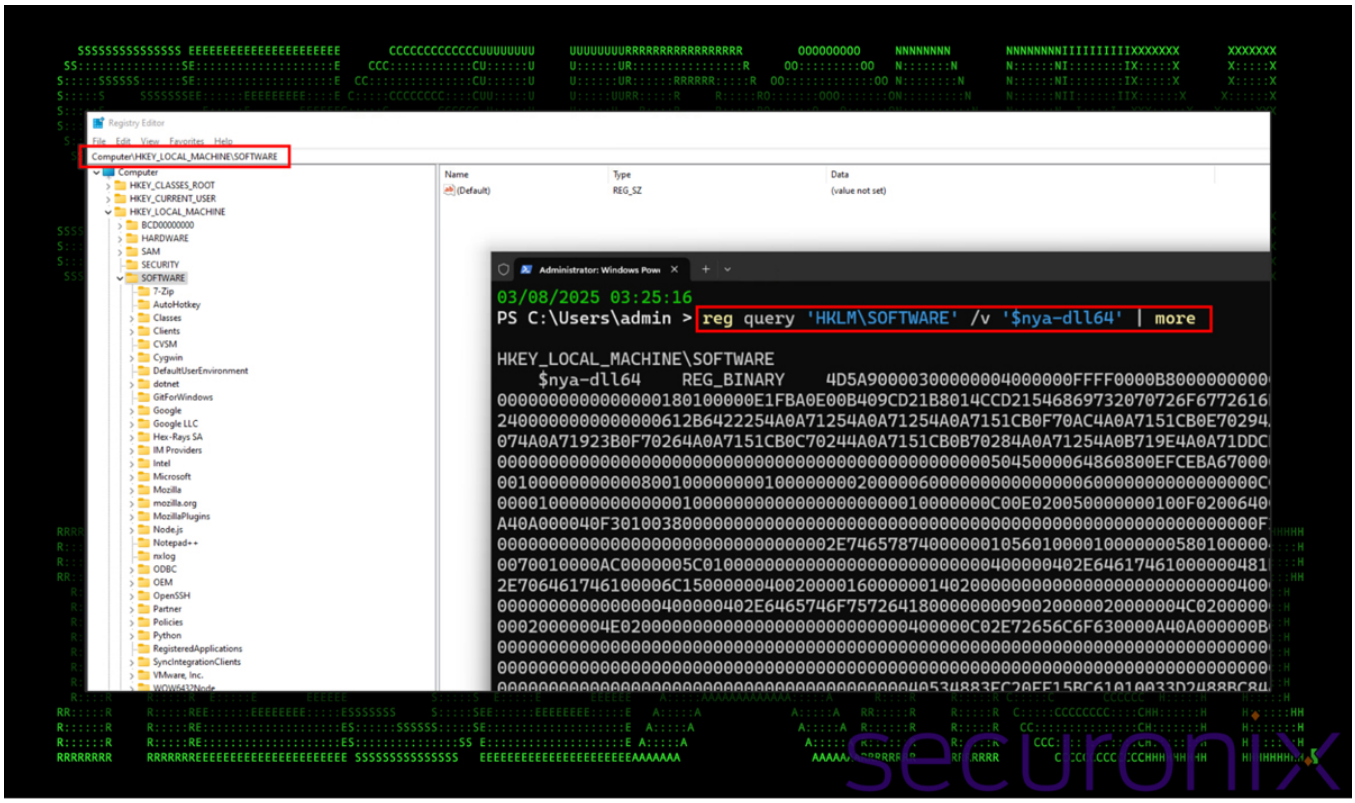

Figure 12: Viewing hidden binary payloads in the registry

Such techniques are common in advanced malware and APT infections, where persistence and evasion are critical.

Execution is handled by calling Windows environmental variables – in this case, `%onimaiuc%` is defined, and when referenced in PowerShell, its registry content is executed.

Code execution from Windows registry

Execution happens as Windows processes environment variables. The malware stores malicious PowerShell code in `HKU<SID>\Environment\onimaiuc`. Windows loads these variables when a user logs in.

When a process references `%onimaiuc%`, it expands the stored registry content as a string. The malware ensures execution by using `Invoke-Expression $env:onimaiuc` or embedding it in another execution mechanism.

This method is stealthy because:

- It avoids writing a visible script to disk.
- Environment variables in `HKU<SID>\Environment` reload automatically at logon.
- It can be executed indirectly via scripts or scheduled tasks.

We can confirm execution by searching for instances of `Invoke-Expression $env:onimaiuc` or monitoring processes that retrieve the value of `%onimaiuc%`.

Figure 13: Testing for the presence of the `%onimaiuc%` environmental variable

Persistence through Scheduled Tasks (extreme stealth edition)

The malware maintains persistence by leveraging Windows scheduled tasks. It creates an XML configuration file "`$nya-qX6Pb164`" in `C:\Windows\System32\Tasks`.

The screenshot below shows the XML contents:

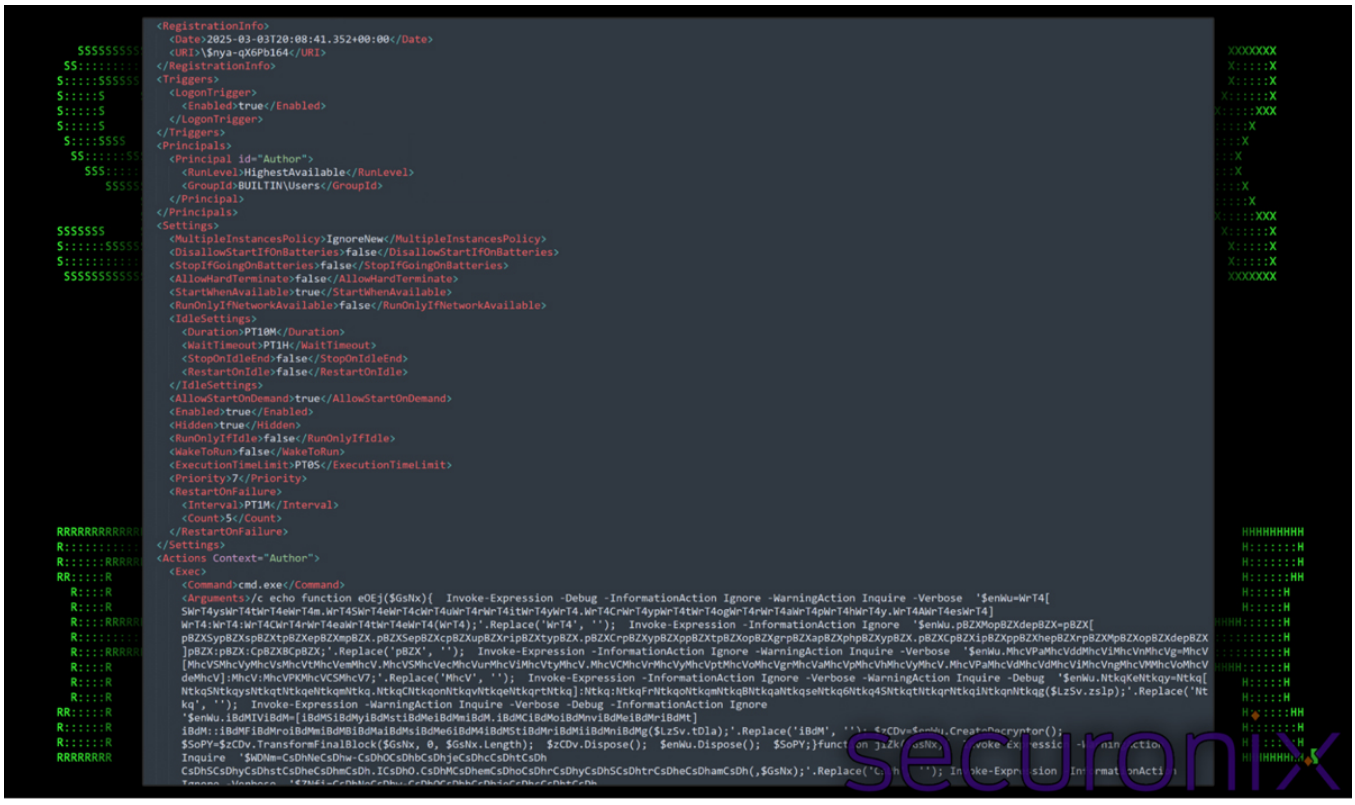

Figure 14: Screenshot of "`$nya-qX6Pb164`" scheduled task configuration

This task executes malicious PowerShell commands – the same script injected into the registry earlier. Let's break down its key components.

Key Components of the Scheduled Task

Trigger:

- <LogonTrigger>: Execute at each user logon.
- <Enabled>true</Enabled>: Task enabled upon logon.

Execution Context & Privileges:

- <RunLevel>HighestAvailable</RunLevel>: Runs with the highest available privileges.
- <GroupId>BUILTIN\Users</GroupId>: In the current user's context.

Settings:

- <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>: No new instance if already running.
- <StartWhenAvailable>true</StartWhenAvailable>: Task runs if missed due to system off.
- <AllowStartOnDemand>true</AllowStartOnDemand>: Task can be manually started.
- <Enabled>true</Enabled>: Task is enabled.
- <Hidden>true</Hidden>: Task is hidden from normal views.
- <RestartOnFailure>: Retries up to five times at one-minute intervals on failure.

Action (execution of malicious code):

At this point, the PowerShell code is executed. (Details as previously observed.)

Overall, this scheduled task ensures the malicious PowerShell code runs at every login by hiding itself and retrying upon failure. It is responsible for decrypting and executing payloads stored in the registry (`HKLM:\SOFTWARE\$nya-OXD9Q`).

The Stealthy side of Scheduled Task persistence

The unusual name "`$nya-qX6Pb164`" is reminiscent of the Windows registry key "`$nya-config`". The screenshot below demonstrates this advanced hiding technique.

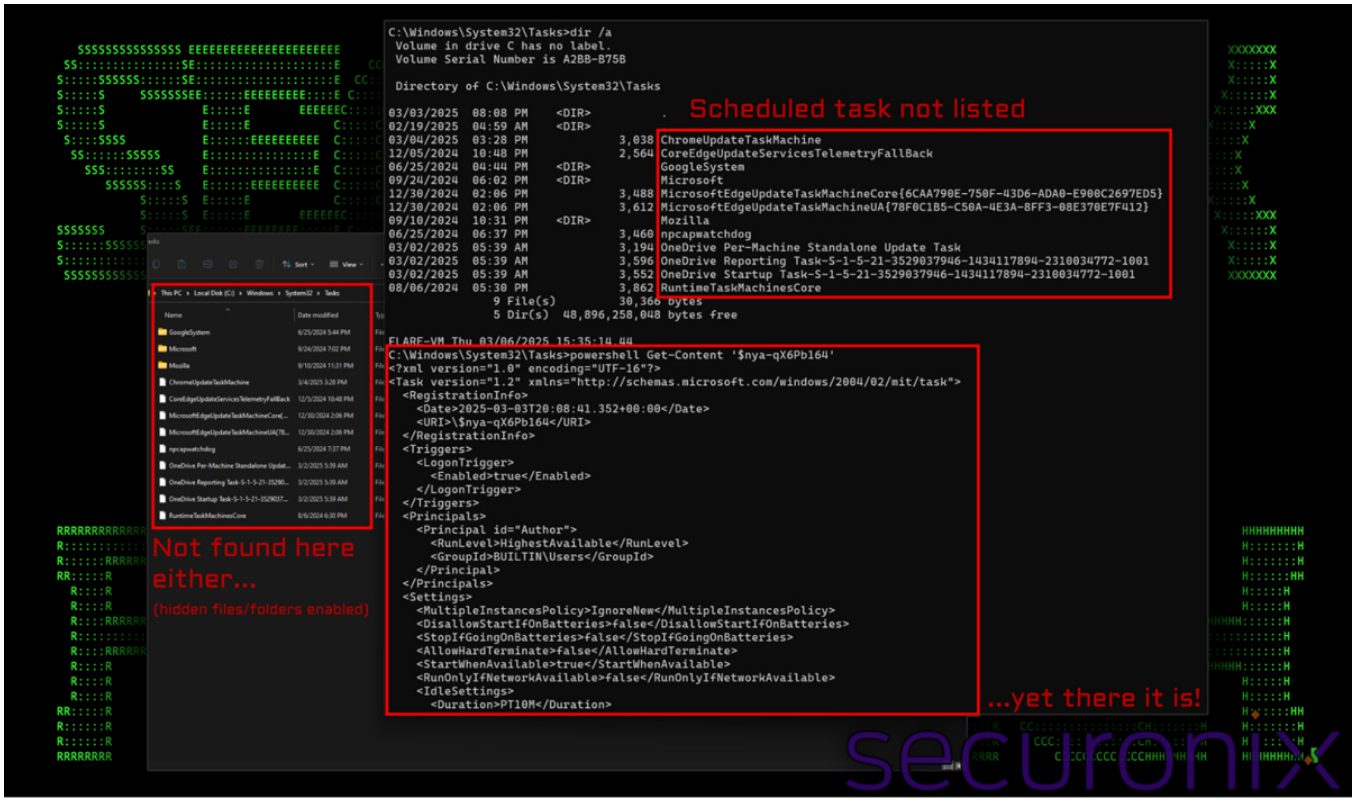

Figure 15: The scheduled tasks file "`$nya-config`" hidden from view but readable via special methods

Binary static analysis (Test.exe/.NET assemblies)

Jumping back into the core components of the attack chain: we observed a binary payload decrypted and reflectively loaded via PowerShell. Let's analyze this binary.

Upon execution, we dumped the .NET binary payload containing the assemblies loaded by PowerShell.

At a high level, the project is simply called "Test".

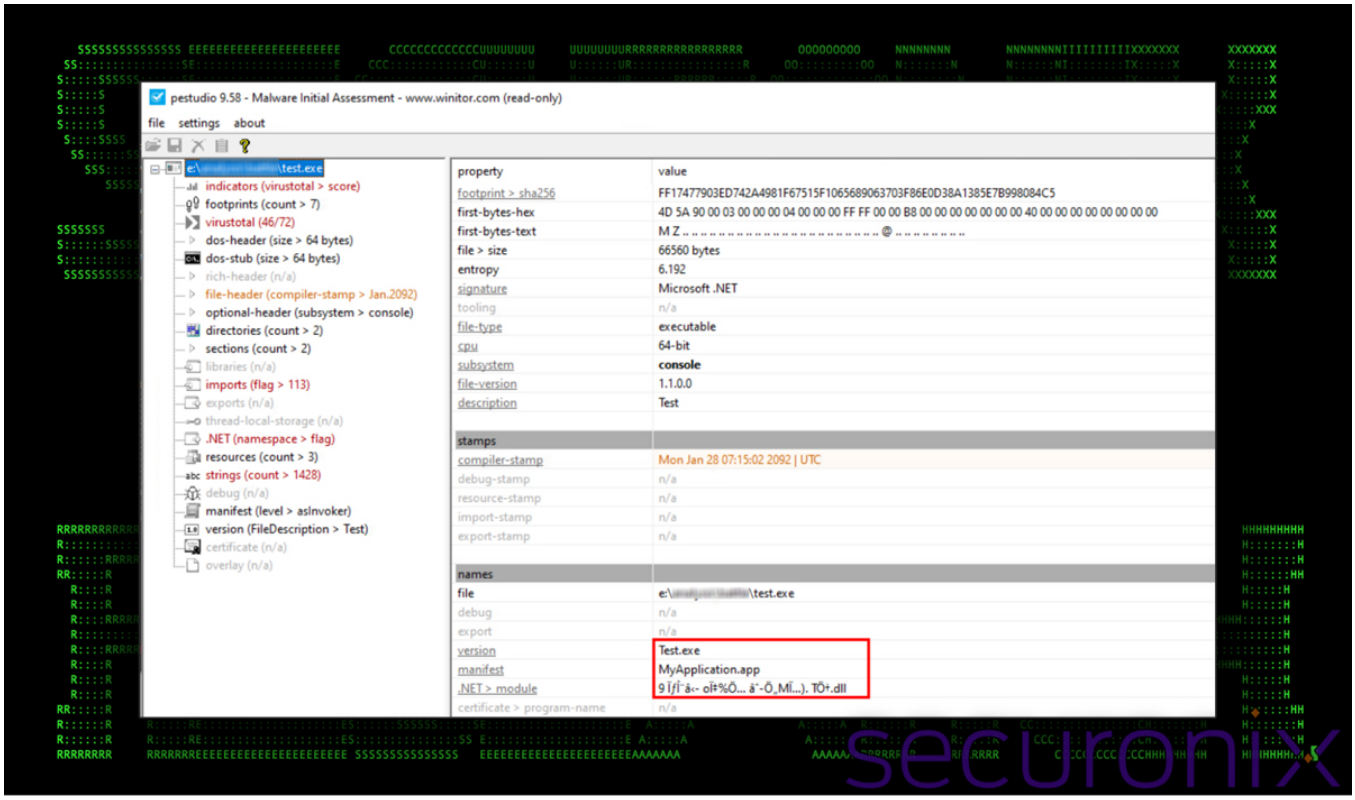

Figure 16: PE file information – "Test.exe"

Cracking open the file in ILSpy reveals heavily obfuscated source code, making reverse engineering challenging.

Figure 17: Obfuscated .NET assemblies

The highlighted function shows advanced obfuscation techniques such as control-flow obfuscation, symbol renaming, and string encryption. Key techniques include:

.NET obfuscation techniques breakdown:

1. Unicode and Symbol Mangling:

1. ぅ昊\uF83C摊鴒黫\u20B1磋ℵLR忘橘绂 \u25B9ო. ㄐ踱侪 – Function names using mixed Arabic, Chinese, and special characters.
2. Some functions start with Unicode escape sequences, hindering automated deobfuscation.

2. Array-Based Execution Flow Manipulation:

1. Multiple arrays (e.g. array, array2, array3) store variables and obfuscated logic assembled in random sequences.
2. “Fixed” statements lock arrays in memory for unsafe operations.

3. Control Flow Flattening & Dead Code Injection:

1. Multiple conditions (if, switch) obscure the true execution flow.
2. Redundant calculations mask variables or strings.

4. Pointer and Memory Manipulation:

1. Unsafe code (e.g., byte* ptr, IntPtr, fixed arrays) indicates raw memory operations.
2. Direct manipulation of memory regions for decryption or function resolution.

Automated .NET deobfuscation tools yielded mixed results, offering only partial readability.

.NET API Calls

By examining API calls (e.g., in kernel32.dll), we gain insight into potential process manipulation, code injection, and evasion techniques.

```
using System;
using System.Runtime.InteropServices;

// Token: 0x02000004 RID: 4
internal class Class2
{
    // Token: 0x06000067 RID: 103
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern bool SetThreadContext(IntPtr hThread, IntPtr lpContext);

    // Token: 0x06000068 RID: 104
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern bool GetThreadContext(IntPtr hThread, IntPtr lpContext);

    // Token: 0x06000069 RID: 105
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    // Token: 0x0600006A RID: 106
    [DllImport("kernel32", CharSet = CharSet.Ansi, SetLastError = true)]
    public static extern IntPtr LoadLibrary([MarshalAs(UnmanagedType.LPStr)] string lpFileName);

    // Token: 0x06000068 RID: 107
    [DllImport("Kernel32.dll")]
    public static extern IntPtr AddVectoredExceptionHandler(uint First, IntPtr Handler);

    // Token: 0x0600006C RID: 108 RVA: 0x00009600 File Offset: 0x00007800
    public unsafe Class2()
    {
        int num = 1;
        int num2 = num;
        int num3 = num2 * 4;
        byte[] array = new byte[1];
        object[] array2 = new object[1];
        object[] array3 = new object[] { this };
        byte* ptr = (byte*)(&struct17.struct17_0);
        int num4;
        while (num4 != 1)
        {
            byte b = (byte)(*(sbyte*)ptr);
            ptr++;
            int num5;
            if (b >= 1 && b <= 3)
            {
                if (2 >= b)
                {
                    calli(System.Void(System.Object), (Class2)array2[num5 - 1], Class2.intptr_0[*(int*)(ptr + num3)]);
                    num5--;
```


Figure 18: .NET API calls

SetThreadContext and GetThreadContext: These functions modify or retrieve a thread's execution context, potentially redirecting execution flow.

GetProcAddress and LoadLibrary: These functions resolve DLL function addresses dynamically to evade static detection.

AddVectoredExceptionHandler: Registers an exception handler; malware may use it for anti-debugging or alternate execution flow.

Overall, the API calls suggest process manipulation, API obfuscation, and anti-analysis techniques.

AMSI bypass

Debugging Test.exe revealed several anti-debugging traps, including AMSI patching to bypass antivirus detections.

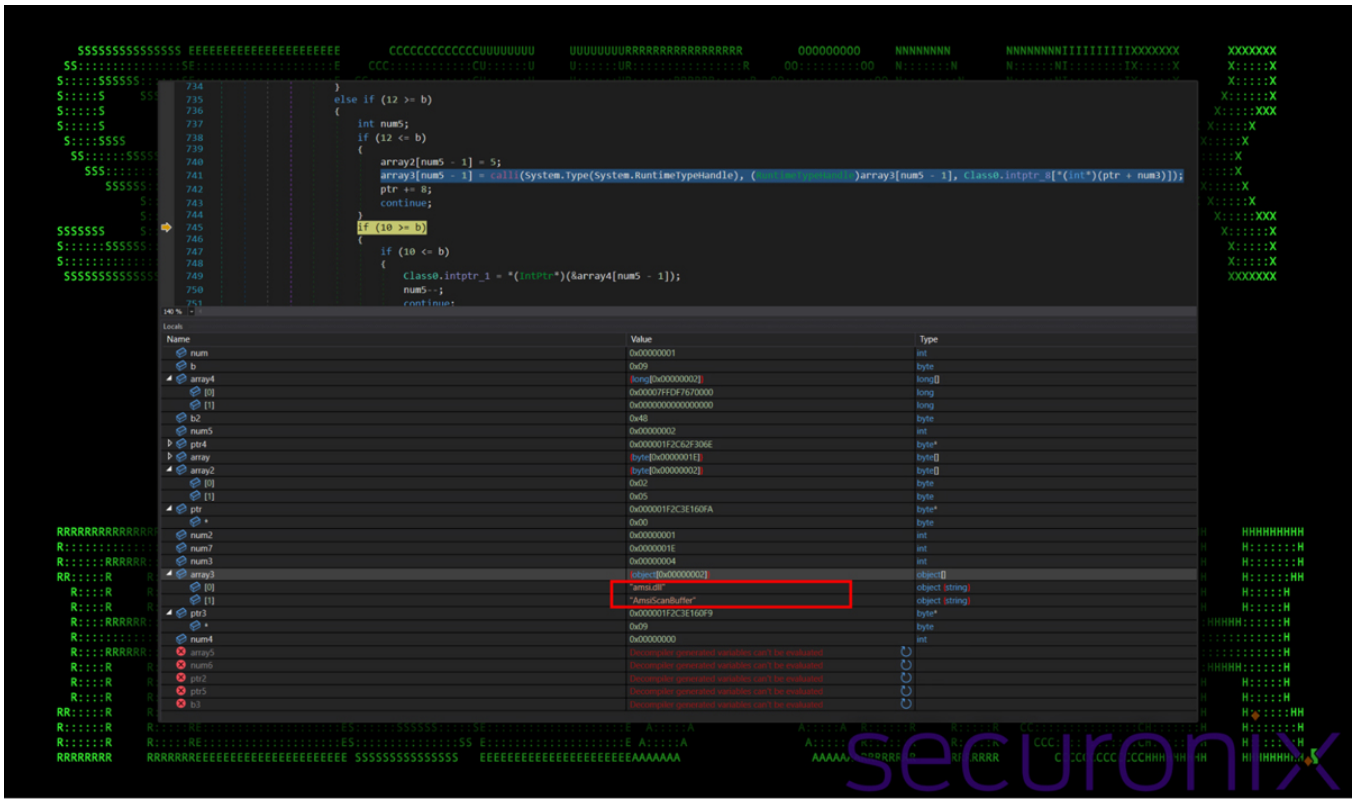

Figure 19: .NET AMSI bypass

AMSI (Antimalware Scan Interface) is designed to block malicious scripts. By patching amsi.dll (using techniques like “calli”), the malware bypasses these scans.

It achieves this by loading amsi.dll and using functions such as LoadLibrary and GetProcAddress to modify AMSI functions like AmsiScanBuffer.

This patch forces AMSI to always return a “clean” result, allowing malicious code to execute undetected.

Binary dynamic analysis (Test.exe/.NET assemblies)

Execution of the PowerShell script that reflectively loaded the binary payload (Test.exe) showed that the payload began beaconing immediately, connecting to 86.54.42[.]120 on port 4782 – a common port for open-source QuasarRAT.

Since QuasarRAT is written in CSharp, this aligns with our static analysis. Typically, such payloads are not obfuscated well; here, the threat actors took steps to make detection harder.

Final payload analysis (ACPIx86.sys)

As part of the attack chain, the .NET payload dropped a malicious file into `C:\Windows\System32\Drivers\` called `ACPIx86.sys`. This occurred concurrently with the creation of malicious registry keys (see “Windows registry injection”).

Windows services for persistence

The malware registers and launches the malicious driver (`ACPIx86.sys`) as a Windows service. It writes entries under `HKLM\SOFTWARE\$nya-config\service_names\ACPIx86` and `HKLM\SOFTWARE\$nya-config\paths\C:\Windows\system32\drivers\ACPIx86.sys` mapping the driver file to a service name.

Additionally, by modifying `HKLM\SYSTEM\CurrentControlSet\Services\ACPIx86`, the service is configured with its location, startup type, and execution parameters. It can be started via standard API calls or the command `sc.exe start ACPIx86`.

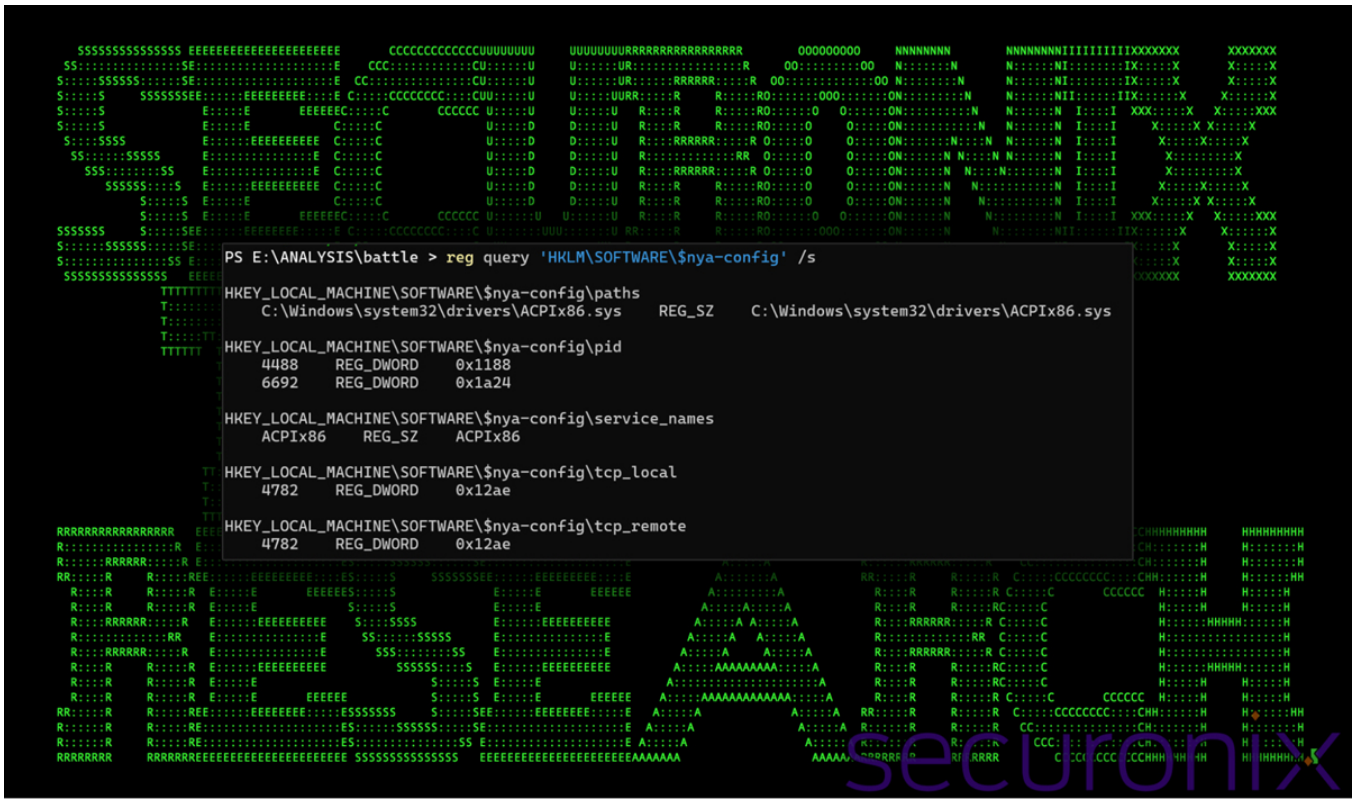

Figure 20: Registry details for the ACPIx86 service

Dynamic execution in a controlled environment yielded limited results – likely due to anti-analysis techniques.

r77 Rootkit analysis

The rootkit used in OBSCURE#BAT appears to be a user-mode rootkit designed to manipulate Windows processes, files, and registry entries to remain hidden.

We identified it as r77 rootkit – an open-source ring-3 rootkit. According to its Github page, it offers functionality including:

- Files, directories
- Processes & CPU/GPU usage
- Registry keys & values
- Services
- TCP & UDP connections
- Junctions, named pipes, scheduled tasks

By default, the r77 rootkit hides the "$77" file prefix; here it has been modified to hide the "$nya" prefix.

The binary payloads for the rootkit are stored in the registry under `HKLM\Software` as `$nya-dll32` and `$nya-dll64`. These DLLs get injected into the PowerShell process invoked by the batch script. Although elevated, the rootkit remains in user space.

A user-mode rootkit operates in ring 3, avoids kernel modifications, and uses registry and scheduled tasks for persistence. Key observations include:

1. Operates in user space (ring 3) via a hidden PowerShell process
2. Uses API hooking rather than kernel hooking
3. Relies on registry and scheduled tasks for persistence
4. Injects into system processes such as winlogon.exe, explorer.exe, or powershell.exe
5. No evidence of active driver installation in Ring 0
6. No modification of SSDT or IRP hooks detected
7. Being user-mode, it evades many traditional kernel-based security tools

We observed multiple stealth techniques to hide files, processes, scheduled tasks, and registry keys.

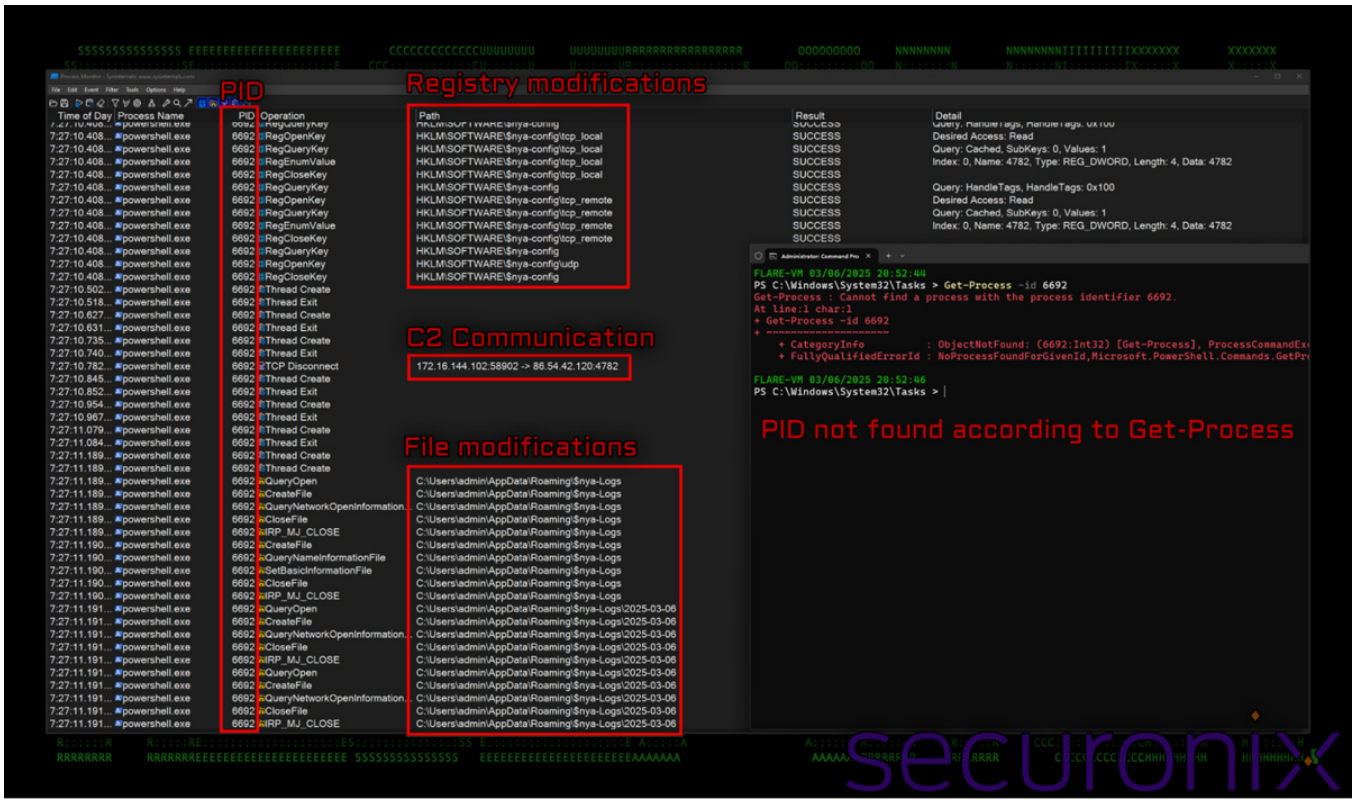

Figure 21: Rootkit hiding under process ID: 6692

Based on our analysis, the r77 rootkit exhibits the following behaviors (observed during dynamic analysis):

r77 Rootkit Capabilities:

Process hiding: The malware hides its process from tasklist, Get-Process, and Task Manager, though Sysmon may still log its activity.

- Likely achieved via API hooking (e.g., NtQuerySystemInformation).
- May modify the System Process Information table to filter its own process ID.
- The malware registers active PIDs in the registry (`HKLM\SOFTWARE\$nya-config\pid`), possibly for tracking.

File masking: Files matching `$nya-*` are hidden from Explorer, dir, and Get-Item. We verified this by renaming a file to `$xxx- prefix`.

- May hook into NtQueryDirectoryFile to filter file names.
- May manipulate NTFS metadata or redirect file queries.

Registry tampering: Certain registry keys (`HKLM\SOFTWARE\$nya-config`) are hidden until specifically queried.

- Likely via API hooking in advapi32.dll intercepting RegQueryValueEx calls.
- Registry editors and standard queries cannot enumerate these keys due to selective filtering.

Kernel interaction: The malware references a driver (`ACPIx86.sys`) in `C:\Windows\System32\Drivers`, executed as a service via registry modification.

- If the driver is loaded but hidden, kernel-mode techniques may be used.
- Could involve direct system calls to bypass user-mode detection.
- May leverage DKOM (Direct Kernel Object Manipulation) to alter process visibility.

Stealthy endpoint and user monitoring

The malware monitors user interactions such as clipboard activity and PowerShell logging. An encrypted binary file is created daily and saved in the "%APPDATA%\Roaming" directory, using the `$nya` naming scheme, which causes the folder to be hidden from Explorer and command shells.

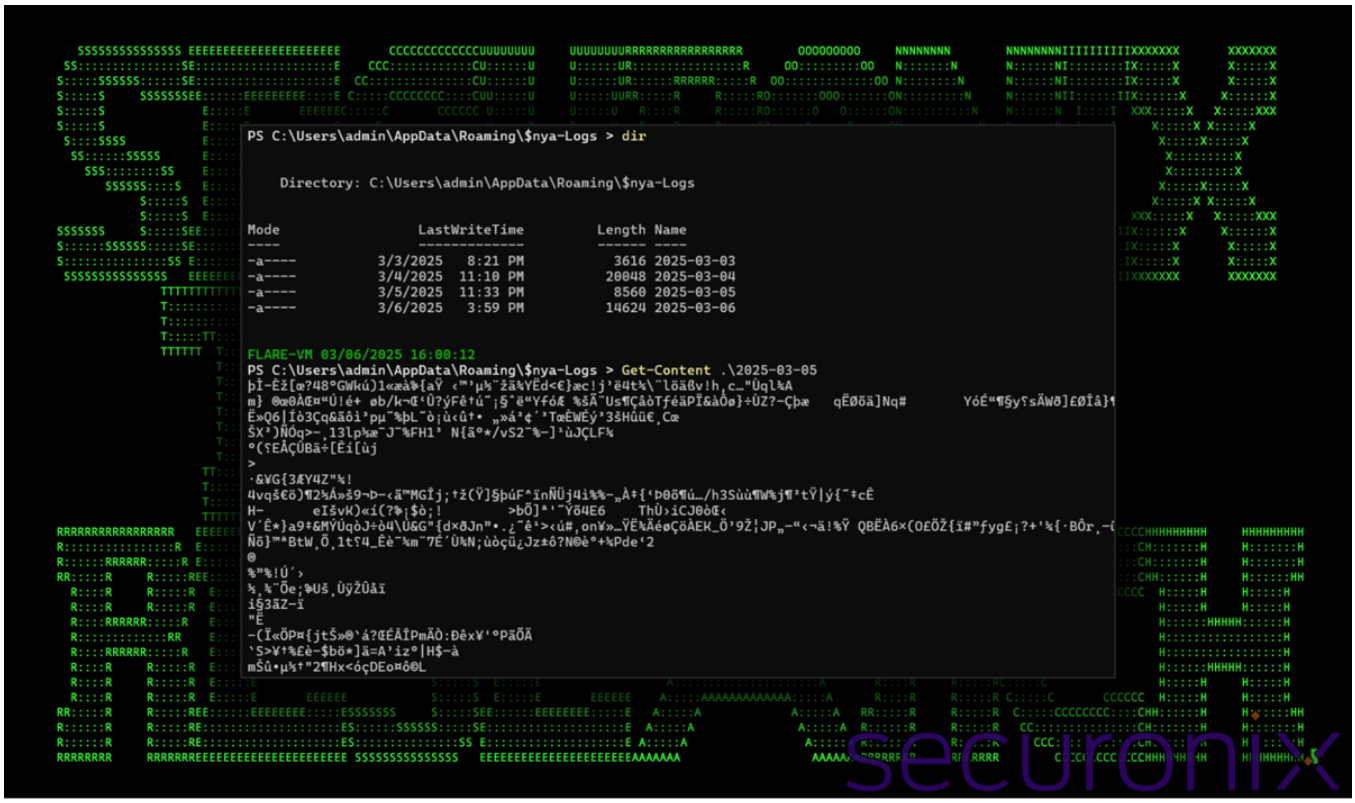

Figure 22: Files containing encrypted user data

Monitoring the r77 rootkit's process via Procmon shows regular file writes and access to these date files, which are likely exfiltrated for decryption and analysis.

Wrapping up…

OBSCURE#BAT demonstrates a highly evasive attack chain, leveraging obfuscation, stealth techniques, and API hooking to persist on compromised systems while evading detection. From the initial execution of the obfuscated batch script (install.bat) to the creation of scheduled tasks and registry-stored scripts, the malware ensures persistence even after reboots. By injecting into critical system processes like winlogon.exe, it manipulates process behavior to further complicate detection.

The most alarming aspect is the user-mode rootkit functionality, which systematically hides files, processes, and registry keys matching the pattern (`$nya-`). This requires deeper forensic and behavioral analysis to detect and neutralize.

Victimology and attribution

We believe the OBSCURE#BAT campaign primarily targets English-speaking individuals. All lure documents, file names, and websites are in English, and our analysis of the attacker's infrastructure indicates a US base, with telemetry also from Canada, Germany, and the United Kingdom.

However, we cannot definitively state which country or threat group is responsible. Updates will be provided as more information emerges.

Securonix recommendations

Maintain vigilance against social engineering, malvertising, and fake captcha scams that trick users into executing code.

- Always verify that software downloads come from legitimate websites.
- A legitimate captcha will never copy code to your clipboard and prompt execution.

Be cautious with batch (.bat) files from unknown sources, as they are a common attack vector in phishing campaigns.

Inspect batch files in a text editor before executing them.

- If infected, note that although the attacker may mask attribution, the original r77 rootkit author provides a remover or uninstaller.
- We strongly recommend deploying robust endpoint logging (e.g., Sysmon and PowerShell logging) for enhanced detection.
- Securonix customers can use the following hunting queries to scan endpoints.

MITRE ATT&CK Matrix

Tactics	Techniques
Initial Access	T1566.001: Phishing: Spearphishing Attachment
Command and Control	T1071.001: Application Layer Protocol: Web Protocols
	T1132: Data Encoding
	T1219: Remote Access Software
Defense Evasion	T1014: Rootkit
	T1027: Obfuscated Files or Information
	T1027.010: Obfuscated Files or Information: Command Obfuscation
	T1036: Masquerading
	T1112: Modify Registry
	T1140: Deobfuscate/Decode Files or Information
	T1055: Process Injection
	T1620: Reflective Code Loading
Execution	T1059.001: Command and Scripting Interpreter: PowerShell
	T1059.003: Command and Scripting Interpreter: Windows Command Shell
	T1204.002: User Execution: Malicious File
Exfiltration	T1102: Web Service
Persistence	T1053.005: Scheduled Task/Job: Scheduled Task
Resource Development	T1583.001: Acquire Infrastructure: Domains
	T1583.008: Acquire Infrastructure: Malvertising

Relevant Securonix detections

- EDR-ALL-1084-ERR
- EDR-ALL-1282-RU
- EDR-ALL-1123-RU
- WEL-ALL-1206-RU

Relevant hunting queries

(remove square brackets “[ ]” for IP addresses or URLs)

- index = activity AND rg_functionality = “Next Generation Firewall” AND destinationaddress IN (“88.222.244[.]187,150.171.28[.]10,195.211.190[.]61,147.185.221[.]24,138.197.66[.]62,185.128.227[.]28,195.211.190[.]61,100.28.201[.]155,3
- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “File created” OR deviceaction = “File created (rule: FileCreate)”) AND TargetFileName CONTAINS “$nya-“
- index = activity AND rg_functionality = “Microsoft Windows” AND baseeventid = “4698” AND CommandLine CONTAINS “$nya-“
- index = activity AND rg_functionality = “Endpoint Management Systems” AND (deviceaction = “Process Create” OR deviceaction = “Process Create (rule: ProcessCreate)” OR deviceaction = “ProcessRollup2” OR deviceaction = “Procstart” OR deviceaction = “Process” OR deviceaction = “Trace Executed Process”) AND childprocesscommandline CONTAINS “$nya-“
- index = activity AND rg_functionality = “Endpoint Management Systems” AND transactionstring5 = “SetValue” AND customstring47 CONTAINS “$nya-“

C2 and infrastructure

C2 Address
hxxps://eloquent-chebakia-e2667a.netlify[.]app
hxxps://dashing-cassata-b94dd5.netlify[.]app
hxxp://45.88.186[.]152:55553
hxxps://klck[.]ai
hxxps://kick[.]am
hxxp://klck[.]pw
hxxps://twitch.co[.]com
hxxps://twltch[.]lol
hxxp://twitch[.]cx
hxxps://twitch.co[.]com
hxxps://twitch[.]team
hxxps://twltch[.]uno
hxxps://rumble[.]tube
hxxps://pnwthrive[.]com
hxxp://cooinbase[.]net
hxxp://tiktoklive[.]studio
hxxp://secure-login-bing[.]com
hxxp://char0nbaby[.]online
hxxp://hyqyj[.]xyz
hxxp://smallmonster[.]net
*.gl.at.ply[.]gg
88.222.244[.]187
150.171.28[.]10
195.211.190[.]61
147.185.221[.]24
138.197.66[.]62
185.128.227[.]28
195.211.190[.]61
100.28.201[.]155
37.114.46[.]25

Analyzed files/hashes

File Name	SHA256
sip.zip	E33E05D3182F46F65554FDA2127D9D1D415A986B6C635485B323558A1821F56A
Eternal2.6.zip	5F7EE1C0FDB813FCF6D8A8E136940AD570BF544B1263C03D288A5EF1B90CF0F3
eternaltool-main.zip	E5386DB097DFC6BD1ACDE5302BC4B22309F151A478604713000BEDB77484881D
Darius SS-2.rar	43CC98694575DEF427DD2ADFB9FCB5E7018AEDCDA525B5E5F5877E3FD02775BE
Darius1378910 project executor-1.rar	BE725B2992385BDCBCF54C995EC1807275B761A019645F707498C958B36346AD
Test.exe (.NET assemblies)	FF17477903ED742A4981F67515F1065689063703F86E0D38A1385E7B998084C5 3367442F903D854AEE965023734F25BFB4BCA6C852D29DCB5774B9E64707FF4B

47B28D3D1AB89E207F7D634B53622960931431DCBF73FC26875659A0C20BD70D
504CC73800ED86C7627234A1D092EFA14ABCEA667AA084191E34FFF2A3EDC167
53D2B22B91F39305B436A08EF9280D4A8FA3BD038D834B1ABEADB792F8E086A1
844BE559DEBDDEC75F460FAA912490DAB6EA400FE325E59B91DF250C1E1AD4FC
FB46FF16BF658AEB5F3A19559AE6AFC10DD2AE108B8AE23457011D6DC5A4B560
F553759453259559CE7C4321898E83C9A3BDDD14758AECBD1567634CA4EA8D86
F3B652503B20261B2F83D43EFC1CC20C655B68A339805714DD95ED14F659D4BE
F180A6E4A5EC5B6EAF82C2BB31FF041A66699387483E9EB489613DBC1BACFE1B
EDDAD50D490349749C5104C3394FAE49DFD6E9070BA0000C139DD8E24E2A06BA
EA0DBC5CA8E96D8940337C5D19574498A4B398847049E62AF14F1D98346638B2
E4CCE18562FDDC70C71A8969141C56ADEB56032196F05E10524374C1EB398D7D
E2864BD791DF7E060B43598F04BE86C839E9907A1FA9C3614205B5139542D8C1
DCBC1A43E1EE9D4C4C5A426CE862B151973545111F69F5B1C036E46E801ACC82
DB82442D83C116211531F104B77ADC5C45CF531315CAFBD8F6E1F9C5DEC6C0D4
D92C28680AF30136DFD52852EDDC07E5197AFE039D84F5B2255B14AE8E15AC02
C08D8E742A34E9DC610ED5276E5CD0DCAD4F6139A03DDA07D9292D50FFD47D39
BE4FADB015D35092F3EE59938A3E68C671DE8C075F04E90FD819B61C383D4501
9B8CAE953C8F3DCB8E9E09D387D217FEA8FDF07C5E3001813A26D83AF7FCB4CC
8F2541E5C425E6353BA1170079B238632ACB21498415861C1FD27A8615A86336
879B9BA401A3B8B580980EA31050A35DD849AD3B6E00338CB81D106BBC02963F
7E658C7C9A1BE6EBD7AF0150FA6FA289D59822B4E771167E13BEDE5C9A622448
7C2A3A41217DA8A2A7D4B72BB5F0C5F45E2B7C6518526101F64B534070651DFC
794D1FF3B3FE275B49138F82B0CC597C35E1FC0A91BE3136729598D97F1086FF
72C5F9A11F126B4D1B79AC81BD03787622F2109560CEEFA762EA0C3A9E1A5E7B
5F75A50AE9F6252D1F0F135726F4A605F4148EE36C9D36C4B2D3FA6404E03B10
5322F5EEB9E789FE63A89CE7852C24593B8B2B6233D855A1646116C14BB8E88E
40F0A201E85E6CF32C48A0CFC496A55A4BB87E8C13174E6C583DE9BF7ED70590
3E88E710043B3CC9BF1AF3B373828E9EF023ABA5D697A82A9A568AE9E45CC544
346EC63BDDAD6B1889D6647ED43DCD71432F687BA8642B726AC67F08E415D77E
30A17C5C65FE87D961AEA290E97E8AA09A03C20D257E22D8AC63F5A7B67C0C6B
2DA6D8FA510E66AE79BAB6B12849B123BB9B88D23DE3B0B383E7F48F9E9CFF69
2BC694A9A6FC03043472F6FB88D3EEDA31722FACF5659AA7EDDB13C29A8FB754
1F51F00B06D5C0358B662AF01DB9690D1EB379B33B1BF7A161BA2B6FE53D6574
18D93547B1F14B452B7AD053A1A93122864D810D82A48ECC391D6D6B44FFD661
0AE3E6A8AF0D7657F820986291DAB1F071007DE4197214C976893EB78E8E200F
0A20A60EF5151F8ADAF9DCD819F970D9AFF20D8EB8F905FBA55CCC0E91C446BE
033F50893BE3BB35EC8CC358D6D7FC764D327B00158617F8DEAC08A60E5F6883
019BA14B03B42A1D3F3496659573E8BA9440340EA16166C3E294164F9BB8F3EF
D0C8C833E2DE4F7D0D92FEBC6A9845CF2A2438013A9362CEFE0878897BD322A4
682F7884B06695A44F19077EB5CD21F1823347B070C8A3773BACEBFC0439B8B8
54CB466D399CE2D3FD24B1B800E276100C3272522FF84DAB4BB1DE73E5EAECDE
FF22090E3E7D9DAC05879802BD0312D282C8A9A44B3C9A7C6AFD4B07C05624A2

FEA5DF5596BE7448E2531CC352BD5A361E128BC6B15E1AC2CA9ABE12927DFB83
F83D936E48BC89338EA9D639F39CF36C3ECAD1E59551F18E2B6D8D5AF6CF403B
EB673D5C936238EED457BFE41AD02F2081F3EF42DD5F3935A0BB11394574C60D
E060A451A4F310D4E4BC05A63B9027896B3126642182D8B176119B975689F217
C3DCE9C45B659118211B573A802ECAC94DEDE201D59B2A5DCED29D68B7A82F3B
C1FE08DEFD1651508B32FFE38892B52A519570F78E457467CCACB6FAE46B2439
BB276B2DD3726F3A712E0904EF87D41D133CD36A72ECB97DA8CEFC6BA0D33A30
BA8565D459CBCC972BBCA96122881E85F5736F4E7B56383853190C95D0334B5D
A439F188BE62856F9DC6668D11C691C031C1CB5A9574A5CAB5CFC1856C7C7676
9063336B99527F9F46B1D1F1D0DB44143B30F478CC708E217525C58CDE5FDDCD
8B10E9C4E8C475FB7357E97205A0E3C8857908DBA93846F7C771E06726DB99FA
79B898274B1AF26AC29F8BD23887244BB3766968B46D5E1012CC4485C6291CE9
70FB59AC30B0FA16FEC656CCA60BC743A32CE6222E9FBDC1896BB2AFAD3EB868
6F0BC6D96340807BAB7A76D132444DC3BA21D99A4C825BCD07363E8D1340CC85
63F6BB98A3E3256F528734F1DEDA5524D97EF3540FB2A06624C92716BA1456FD
519A389D0D183FA5CAE0390CE8CC2716247B8F50332DE8E4FC8BEE5026C8BA70
51267FBD6AA2D09FD1ECD4C9F52557AA85F3F5AF0C60223AEA55ACB562DF9AA8
4304A7028616787990476CEB92CE98842C8E049278AD9E4AFA24A1FCF1DAD782
20217810AED81399374FEC1F556B1537FA35B6499133F65E08EEDA324A72680F
31E9EA58C807633DBCC680303F1F741E2A6E58747E040E98DB133E076C6CBBED
11E0596F453CCF8F30BA8177E7DF50517E364326585D1CEFB4C59DE277B0F6AA
EF83368CAF9C9D53B4CB74B76A96E7092097E35DA214BA946A146580443D0792
B6180699E895945D6178A61FD8228576629ED0FA03DE54D78E7EE16F271F1522
2BE90FEEA3580358A7AB90D744B7C3028C8E7694863672A8A30F021B284CC94A
0368BF651D5CE7E58305DD300A428D12E6A64D456A4805E845CB1985196BB5FC
CCF58B300EC2A7D491CBD492373CCF5175056F55D4843889AD15C3F0B2DB815F
CC8E4C0C2E126938B827ACFCEC306DC9811D4AAF00934D397E3841FD6352F4D1
7BF9BE59DB4C55F5B372576B7BAAA25CFF2716D2F7E24DB1B98724A0E0927ECB
28A4EFF21C27F9E3B0E7B5383ABB407E0A3D69BFC1D094DDD4F0B5C18425C523
E1070A6B5A406CE70CF1D1655169A4EC36FB69114C3064F00D28CC01CEDFB0E8
AF9C847CC0A204E969A0FAB93AE676B06222892AFC5271186604E348852BAD37
80AF77DC9C38A3BBFB68FA66635BB3F202D72DC093305D1218BB85811CE018C9
52634530F53DFCE289317B2C4057811136FBBE873211D01E150AC32A94DD0F4A
68A0F5040DCF9B7881D1557CAD827275271027906F830F6EE90E5521A00B72E4
3B86B107B36AA1224DF2E46419F2652682C67F99222011FD63F7AB3ED43AB1D5
B2BCBC0FB471660632B6589FA96656F935F72AD5308E9B659B2E59ACAF820E02
7AA0D53BC4A08E7B61AA283C39BECCF7364AFC2174FFC958B3C5FD2D56DD9554
E1070A6B5A406CE70CF1D1655169A4EC36FB69114C3064F00D28CC01CEDFB0E8
CC8E4C0C2E126938B827ACFCEC306DC9811D4AAF00934D397E3841FD6352F4D1
E1070A6B5A406CE70CF1D1655169A4EC36FB69114C3064F00D28CC01CEDFB0E8
EE481AE34BE52DA5D9F2E8DBFAB3BDD228A7FDFA9FFF308E98B7691D3EB9D0FA
28DC3771DC4AA5D8A19BA732479F3719C276E62CFFD0FE8ADE6159A1FD3BA880

	35BFF2270DAA66D092AFAF7E6CFA3210790E1D17DD77E0AF94B361DBF632B571
install.bat	C9AA237A2A30B901D52D0074731B5AC57F70322F1FDE81F6794588C17D6BB268
Loader.bat	612FBFEBFDCC12D6EAA20F22835A1A360A747C043AE1058070D4A71EF20A59DA
downloaded_file.bat	0877653F6A24639BB02B547C94F670597C3C0CD96DF910A2AC891EAEAA9CC5F3
chiani.bat	159057BA35F3454424A4901866DE6DE286BD11715E975D4D124D33B2E83055C7
TorServer.bat	28BDD3E3C182A8E9A5082A4677DF2F2116C8973D4ECDAE023CF9FD3489B9F012
AdobeUpdate.bat	7C684A112461478A8ED1F3885628D0235BC20081C11C55FF96CC98454E096944
nouaconi.bat	2ACD42AD45F8DBB3866B537E6135672DAA48921EB00668B657243B666991C4AD
oni.bat	625E590FD62A1C8B4C85BC9F551188DE0627C5D43B234408CE57139F4EA0B7C0
uacbypassv1.bat	160165D2E7F332B29B5980C27C044DE2804552469EC70458DF6E77DFD254765E
CVerify.bat	B6B68CDAC6CDB3956DC8B7C11454E4F493FBD9157F902FFD2539545D6F7315C7 62F035A79382BF50E9959FCB272C19D5AAC64A7409DEEBF7C8E9B597F3954DB4 B2528CB39295490B53428A98FEFAFDE2D5F32C957B268F528B66756AD8AB6896 E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855 431435E3A7A8E38458CD2A2C1E97C6B3AEC993E9DFC3DE5CC665AC57C21BF528 A79E4199BA0DAC6948DEA075C26CEC05DE18D3217A25000A3C7A0FCB45BA1B02 1BB63CB3D89389F426F4CB5350E38FBD0C49CE1851F6311C5E5D246200F29DE8
cloudflare.bat	96AA71F70D16E2784488FAD332AC65287F33D059CDE4CD2858B0DAB85340BA0D
newest.bat	C1D0981485F8AFEA96D8C0CE85CB9888C96418387B8C43103B113BA283F8C59D
Mous Fix.bat	C7D9A5FFA94EECFBF5C22781750E404C6E50BA12459E3071670783598839CFDF
repair.bat	CF3F8D4D3EE1A8EFF414A221767EDE4C73424BB62D6A090AD3F65EA55FF22FA4
sus_bat.bat	D4B46254B03F3800038CC93F226BC7B5897C34FCAC0D16210F45731B57D4F86A
32ram1.2.bat	DD0396754DF3ACA8A482242EEBFA92DB7433781EBDD679507E329E34D4065C98
img.bat	04DBC65A0EC0A3D95AEEC8161816352A22CE74C19FCD002F631879E990C2D468
Eternal2.6.bat	FC6A52FA9D578565E4B6C47BE3A4F0358A01ACE3ED601C0EB88E46DD88203EBF
Loli.bat	6CD9BDC704701AB3618BE8546E471F335431929A96D10BC59F66872B144770CE
Darius1378910 Project SS.bat	9BA76C05333A17C734B4B6174E68222F689F298EF48FE5DD03D25DA7E01904F5
Bootstrapper.bat	F6E19E1C17291B9B4B2436D561D8373CF8A18841B5C4393205845BCC6BA31616
test.bat	9B7CFE2A7F46AAD42C8AEB5FCB668F2286B24FCD0241AD7C1B1B3D00856C2B18
Java_installer.bat	903391CC79136EB1EFDCA469686B96FD04FAF257D87796DDE594C500AB226150
install_apache.bat python_installer.bat	7F970EE9B2FEF5C77DB4CDB7FE536377E165E056AA056299624A224EF8E0CBA9
installer.bat	C4DF50417827B20CAFC1E724948D576CD1B90636E5D68D577856824CB9CEB328
ChatGPT+.bat	8713DD146895B8262E5096E49399DBDF4DC796D37A532912C0EBBB46DE059EAF
share.bat	EFC09D4380483145573AC4F1A2B4FE308E9BD4378BFFBC44EFD00739D2E055A7
uacbypassv1.bat	160165D2E7F332B29B5980C27C044DE2804552469EC70458DF6E77DFD254765E
uacbypassoff3925.bat	F3D48BB2DC545F0864D8B85D93AEA9C2B9A55F0FC9C7435F1DEE000802A261DA
uacon31025.bat	53B9C02EE582BF97385BEB39EE140C49F73C557CCECF3BF44C795899083A3519
AsyncClient.exe	5906FE2B69A5874697B84882DF732F77DD3160221D0746F9688E9AA9B8E0AF31
autoruns.exe	F41051697B220757F3612ECD00749B952CE7BCAADD9DC782D79EF0338E45C3B6

DefenderUpdate.exe	C7BDCEBE60356900DC4B4F8BC8B75ACC1536DF33AE7A1049BFA27192B8C62D0A
qexplorer32.exe	48B7AAE41C1F229DADD80E7635A142175CBA75D03D54F08952269720C5F2735B
rem_edge.exe	E0C27D9A377E5F18AE850D1D0EF1D69934934CABDB95172E21FE0E36807243C8
rem_edgev6.exe	BDE4436AAC1E27FE22B134CADC1E19DD954D350A5619C3593EFAC659AB1BBEFD
$nya-qX6Pb164	0ABECC48522A2AA66C798E817F0412DC71DF2875B8908255208642FE019AB9D3
nya-dll32.dll	1BAD202E452B1D1F8B365E946C446F889B2479A6198A17A3DEA1D6A4E5D12052
nya-dll64.dll	8004DF38975733770A7E2A0C71D284BC3439EB7EE74077F950AD7C0BAF2512AA

References:

[a] Change to paragraphs