Cracking ValleyRAT: From Builder Secrets to Kernel Rootkits

By shlomoo@checkpoint.com

Published: 2025-12-10 · Archived: 2026-10-27 02:14:54 UTC

Highlights:

- Check Point Research (**CPR**) presents a full dissection of the widely used **ValleyRAT** backdoor, also known as **Winos**/**Winos4.0**, covering its modular architecture and plugin system.
- By analyzing the publicly leaked **builder** and **development structure** (Visual Studio solutions and project files, **without** source code), we were able to accurately correlate artifacts and reverse engineer the functionality of all "**main**" plugins. The analysis reveals the advanced skills of the developers behind ValleyRAT, demonstrating deep knowledge of Windows kernel and user-mode internals, and consistent coding patterns suggesting a small, specialized team.
- The "**Driver Plugin**" contains an embedded kernel-mode **rootkit** that, in some cases, retains valid signatures and **remains loadable** on **fully updated Windows 11** systems, bypassing built-in protection features. Through detailed reverse engineering, previously unknown capabilities were uncovered, including **stealthy driver installation**, user-mode shellcode **injection via APCs**, and **forceful deletion** of AV/EDR drivers.
- The detection **statistics** for **ValleyRAT plugins** in the wild (ITW), derived from carefully crafted detection rules and verified using both internal telemetry and public services, highlight the recent surge in ValleyRAT usage, with approximately **85%** of detected samples appearing in the **last six months**, coinciding with the **public release** of the **builder**.
- The research underscores the growing accessibility of the ValleyRAT builder and development artifacts, emphasizing that future usage **cannot be easily attributed** to specific Chinese-speaking threat actors, such as Silver Fox.

Introduction

Throughout 2025, we conducted and published several reports related to our research on the Silver Fox APT. In some of them (for example, here), the threat actor delivered the well-known **ValleyRAT backdoor**, also referred to as **Winos** or **Winos4.0**, as the final stage. Since this malware family is widely used, modular, and often associated with Chinese threat actors such as Silver Fox, we decided to take a deeper look at its development, plugin system, and the capabilities exposed through individual components.

At first, we focused on collecting as much information as possible from publicly available sources. However, after the initial survey, we realized that despite the first report related to ValleyRAT being dated to early 2023, the existing material is quite limited. Most publications discuss only the specific plugins that happened to be deployed to victims or malware labs during analysis, which in many cases means only one or two plugins.

When a potential victim, including a malware lab, is infected with the ValleyRAT backdoor, the initial modules deployed are usually first-stage plugins such as the "**Online Module**" or "**Login Module**". These act as initial beacons and are responsible for retrieving and loading additional plugins from the ValleyRAT C2 server. It is entirely up to the attacker to decide whether a victim appears interesting enough to receive further components and expose more of the backdoor's capabilities. Many victims, especially malware labs, do not meet this threshold, meaning analysts only get to see the plugins that operators intentionally delivered.

Because of this limitation, we shifted our attention to searching for leaked ValleyRAT **builders** and **source code**. Normally, materials like these circulate on the dark web within small, restricted communities. Fortunately, in today's "share whatever, wherever" environment, we were able to obtain them from several publicly available GitHub repositories.

ValleyRAT is strongly associated with Chinese-speaking threat actors, so we expanded our search with Chinese keywords and phrases that seemed relevant to the malware. This approach was successful: we found not only the ValleyRAT builder but also its development structure, including Visual Studio solutions and project files. Although the actual **source code was missing**, we were still able to link individual Visual Studio projects to plugin binaries extracted from the builder.

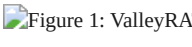
It is worth mentioning that we achieved these results only after going through many trojanized repositories, including ValleyRAT builders infected with other backdoors. Hackers hacking hackers. Eventually, we located repositories that

contained exactly the information we were looking for.

In this publication, we focus on a full dissection of the ValleyRAT modular system. We describe the builder, outline our extraction and analysis methodology, and provide detailed explanations for every plugin used by the malware. We also include an in-depth analysis of one of the most interesting components, the "**Driver Plugin**", which embeds a kernel-mode **rootkit**. Special attention is given to the rootkit and its user-mode client, along with the techniques they implement. Finally, with carefully crafted detection rules for all plugins, we present their in-the-wild detection statistics based on both our internal telemetry and public services.

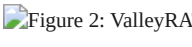
Background & Key Findings

The first interesting repository we identified is related to the ValleyRAT builder, which also functions as a C2 panel and has been publicly available since March 2025. Below is a comparison between the original Chinese repository and its translated version.


Figure 1: ValleyRAT builder/C2 panel – GitHub repository.

The claim about the "**Effective date: March 25, 2025**" mentioned in the repository structure (with the builder archive itself uploaded on March 26, 2025) correlates with the PE compilation timestamp of the builder: **Wednesday, 26.03.2025 04:10:15 UTC**. This suggests that it is likely one of the latest versions of the ValleyRAT builder.

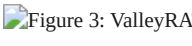
The second repository, which is slightly older (June 2024), contains development artifacts related to the ValleyRAT plugin system, including Visual Studio solutions and project structures (**without source code**). A translated version of this repository can be seen below.


Figure 2: ValleyRAT development structure – GitHub repository.

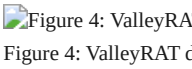
The fact that the development structure (without source code) has been publicly available for a while suggests that the **leaked ValleyRAT source code** itself has probably been circulating in the wild as well.

Both repositories are in Chinese, and as an initial step, we focused on the one containing the Visual Studio structure. We then attempted to locate matching artifacts inside the compiled builder package from the first repository.

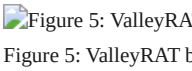
While the development structure referenced both "**main**" and "**auxiliary**" plugins, only the compiled "**main**" plugins were present inside the builder's PE resources.


Figure 3: ValleyRAT builder – plugin system in PE resources.

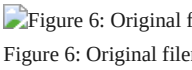
During the analysis of the development structure, one component immediately stood out: a plugin named "**Driver Plugin**", which appears to include a kernel-mode driver component.


Figure 4: ValleyRAT development structure – "Driver Plugin".

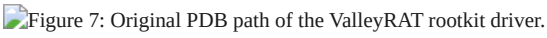
We were later able to locate this compiled plugin inside the builder's resources.


Figure 5: ValleyRAT builder – locating the "Driver Plugin" in the compiled PE resources.

The "Driver Plugin" is a DLL that acts as a user-mode client for an embedded rootkit driver. Its original filename can be recovered from the Export Directory.


Figure 6: Original filename of the "Driver Plugin" DLL.

The rootkit driver is stored within the `.data` section of the `Driver Plugin.dll`. After careful extraction that preserved its original `WIN_CERTIFICATE` structure, we identified the exact sample on VirusTotal. The driver retains the original PDB path that closely matches the Visual Studio project path from the development structure.


Figure 7: Original PDB path of the ValleyRAT rootkit driver.

The driver's compilation timestamp appears intact: **Sunday, 23.04.2023 08:10:50 UTC**. However, despite being compiled in 2023, it is signed using an expired certificate valid only between **2013–2014**. We believe this certificate was stolen and used to sign the driver.


Figure 8: ValleyRAT rootkit driver – certificate inspection.

Even though the certificate had expired, the signature still fell under the Windows Driver Signing Policy – Exceptions (the legacy driver category). For this reason, the rootkit driver could be loaded even on the latest Windows 11 systems. During our initial investigation, the certificate had not yet been revoked, allowing the driver to load successfully; later revocation prevented this.

Using carefully crafted internal detection rules covering the entire ValleyRAT modular system (all "main" plugins and the rootkit driver), we identified approximately **6,000 ValleyRAT-related samples** in the wild between November 2024 and November 2025. Notably, around **85%** of these detections occurred within the last six months of that period, which correlates with the time the ValleyRAT builder first appeared publicly. This clearly reflects the growing adoption of this modular backdoor.


Figure 9: Statistics – ValleyRAT plugin detection (ITW).

Another notable observation is that among the detected samples, we found **30 distinct variants** of the ValleyRAT builder and **12 variants** of the rootkit driver. The majority of the detected rootkits were compiled in 2025, based on PE compilation timestamps that appeared intact. Seven of the drivers were still signed with valid (non-revoked) certificates. Despite all certificates being expired long ago (validity periods ending before 2015), they fall under the driver signing policy exceptions for end-entity certificates issued before July 29th 2015 that chain to a supported cross-signed CA. We confirmed that several of these drivers were not properly detected by Microsoft Defender Antivirus, were absent from the latest version of the Microsoft Vulnerable Driver Blocklist, and could still be loaded on fully updated Windows 11 systems with all protection features enabled (including HVCI and Secure Boot). We responsibly disclosed these findings to the **Microsoft Security Intelligence** team.

As expected, an APT-level threat actor deployed this capability for a reason. ValleyRAT includes a kernel-mode module that functions as a rootkit and remains loadable even on the latest Windows versions with modern security mitigations in place.

We also discovered an interesting connection between the Chinese Sun-RAT "company" website (`https://www.sun-rat.com/`), which advertised a commercial remote-administration tool, and the ValleyRAT builder. While the main page of the website was accessible during our initial investigation, it has since disappeared, though some subpages (such as the contact and login sections) remain online. By obtaining the Sun-RAT demo product, we were able to compare it with the ValleyRAT builder.

Sun-RAT software:


Figure 10: UI view of the Sun-RAT software. ValleyRAT builder:

ValleyRAT builder:


Figure 11: UI view of the ValleyRAT builder. The similarities strongly suggest one of two possibilities:

The similarities strongly suggest one of two possibilities:

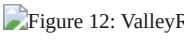
1. The ValleyRAT developers stole the source code of Sun-RAT and built their backdoor on top of it, or
2. The **leaked ValleyRAT source code** was repurposed to create a commercial product marketed as a legitimate Chinese tool.

We believe the second scenario is more likely.

In the next section, we dive into the ValleyRAT builder internals, the plugin extraction process, and the functionality of each component in the modular system.

Technical Analysis: ValleyRAT Builder

The obtained ValleyRAT builder is a 32-bit PE file, compiled on **Wednesday, 26.03.2025 04:10:15 UTC**, containing the plugins inside its resources. As previously mentioned, the builder includes only the "**main**" plugins and not the "**auxiliary**" ones.


Figure 12: ValleyRAT builder – 32-bit PE – resources.

To analyze all compiled plugins and any additional utilities embedded elsewhere in the builder (some were found in the `.data` section) or even within the plugins themselves, we needed a reliable extraction strategy. To increase confidence in the correctness of the extraction, we adopted a dual-tool methodology: DIE – Extractor (operating as a smart carver capable of pulling PE32/PE64 files even when nested inside another PE) and Resource Hacker. To validate the extraction results across DIE + Resource Hacker, we compared authentihashes and output sizes of the extracted PE files against expected PE sizes. Below is an example script demonstrating authentihash computation using LIEF:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
#!/usr/bin/env python3
"""
Compute Authenticode authentihash for all PE files in a directory using LIEF.
Usage:
python authentihash_lief.py /path/to/dir [--algo sha256] [--recurse]
Example:
python authentihash_lief.py C:\Windows\System32 --algo sha256
"""
import os
import sys
import argparse
import lief
# Map user-friendly names to LIEF enum values
ALGO_MAP = {
"sha1": lief.PE.ALGORITHMS.SHA_1,
"sha256": lief.PE.ALGORITHMS.SHA_256,
"sha384": lief.PE.ALGORITHMS.SHA_384,
"sha512": lief.PE.ALGORITHMS.SHA_512,
}
def compute_authentihash(path, algo_enum):
```

"""Return the authentihash (bytes) computed by LIEF for the given PE file."""

pe = lief.parse(path)

if pe is None:

raise RuntimeError("Failed to parse PE file")

digest = pe.authentihash(algo_enum)

return digest.hex()

def scan_dir(directory, algo_enum, recurse=False):

"""Iterate through directory and print <file>\t<authentihash>."""

for root, dirs, files in os.walk(directory):

for fn in files:

full = os.path.join(root, fn)

try:

with open(full, "rb") as f:

if f.read(2) != b"MZ":

continue

digest = compute_authentihash(full, algo_enum)

print(f"{full}\t{digest}")

except Exception as e:

print(f"[!] {full}\tERROR: {e}", file=sys.stderr)

if not recurse:

break

def main():

parser = argparse.ArgumentParser(description="Compute Authenticode authentihash for PE files using LIEF")

parser.add_argument("directory", help="Directory to scan")

parser.add_argument("--algo", default="sha256",

choices=["sha1", "sha256", "sha384", "sha512"],

help="Hash algorithm (default: sha256)")

parser.add_argument("--recurse", action="store_true", help="Recursively scan subdirectories")

args = parser.parse_args()

algo_enum = ALGO_MAP[args.algo.lower()]

scan_dir(args.directory, algo_enum, recurse=args.recurse)

if __name__ == "__main__":

main()

#!/usr/bin/env python3 """ Compute Authenticode authentihash for all PE files in a directory using LIEF. Usage: python authentihash_lief.py /path/to/dir [--algo sha256] [--recurse] Example: python authentihash_lief.py C:\Windows\System32 --

algo sha256 """ import os import sys import argparse import lief # Map user-friendly names to LIEF enum values ALGO_MAP = { "sha1": lief.PE.ALGORITHMS.SHA_1, "sha256": lief.PE.ALGORITHMS.SHA_256, "sha384": lief.PE.ALGORITHMS.SHA_384, "sha512": lief.PE.ALGORITHMS.SHA_512, } def compute_authentihash(path, algo_enum): """Return the authentihash (bytes) computed by LIEF for the given PE file.""" pe = lief.parse(path) if pe is None: raise RuntimeError("Failed to parse PE file") digest = pe.authentihash(algo_enum) return digest.hex() def scan_dir(directory, algo_enum, recurse=False): """Iterate through directory and print <file>\t<authentihash>.""" for root, dirs, files in os.walk(directory): for fn in files: full = os.path.join(root, fn) try: with open(full, "rb") as f: if f.read(2) != b"MZ": continue digest = compute_authentihash(full, algo_enum) print(f"{full}\t{digest}") except Exception as e: print(f"[!] {full}\tERROR: {e}", file=sys.stderr) if not recurse: break def main(): parser = argparse.ArgumentParser(description="Compute Authenticode authentihash for PE files using LIEF") parser.add_argument("directory", help="Directory to scan") parser.add_argument("--algo", default="sha256", choices=["sha1", "sha256", "sha384", "sha512"], help="Hash algorithm (default: sha256)") parser.add_argument("--recurse", action="store_true", help="Recursively scan subdirectories") args = parser.parse_args() algo_enum = ALGO_MAP[args.algo.lower()] scan_dir(args.directory, algo_enum, recurse=args.recurse) if __name__ == "__main__": main()

```
#!/usr/bin/env python3
"""
Compute Authenticode authentihash for all PE files in a directory using LIEF.

Usage:
    python authentihash_lief.py /path/to/dir [--algo sha256] [--recurse]

Example:
    python authentihash_lief.py C:\Windows\System32 --algo sha256
"""

import os
import sys
import argparse
import lief

# Map user-friendly names to LIEF enum values
ALGO_MAP = {
    "sha1":   lief.PE.ALGORITHMS.SHA_1,
    "sha256": lief.PE.ALGORITHMS.SHA_256,
    "sha384": lief.PE.ALGORITHMS.SHA_384,
    "sha512": lief.PE.ALGORITHMS.SHA_512,
}

def compute_authentihash(path, algo_enum):
    """Return the authentihash (bytes) computed by LIEF for the given PE file."""
    pe = lief.parse(path)
    if pe is None:
        raise RuntimeError("Failed to parse PE file")
    digest = pe.authentihash(algo_enum)
    return digest.hex()

def scan_dir(directory, algo_enum, recurse=False):
    """Iterate through directory and print <file>\t<authentihash>."""
    for root, dirs, files in os.walk(directory):
        for fn in files:
            full = os.path.join(root, fn)
            try:
                with open(full, "rb") as f:
                    if f.read(2) != b"MZ":
                        continue
                digest = compute_authentihash(full, algo_enum)
                print(f"{full}\t{digest}")
            except Exception as e:
```

```
                print(f"[!] {full}\tERROR: {e}", file=sys.stderr)
        if not recurse:
            break

def main():
    parser = argparse.ArgumentParser(description="Compute Authenticode authentihash for PE files using LIEF")
    parser.add_argument("directory", help="Directory to scan")
    parser.add_argument("--algo", default="sha256",
                        choices=["sha1", "sha256", "sha384", "sha512"],
                        help="Hash algorithm (default: sha256)")
    parser.add_argument("--recurse", action="store_true", help="Recursively scan subdirectories")
    args = parser.parse_args()

    algo_enum = ALGO_MAP[args.algo.lower()]
    scan_dir(args.directory, algo_enum, recurse=args.recurse)

if __name__ == "__main__":
    main()
```

Using this approach, we successfully extracted all plugins, helper tools, and—most importantly—the rootkit driver, which we analyze in depth later. Among the extracted helper utilities were known third-party tools such as **UPX**, **BoxedApp SDK**, and an extended logging library.

The core focus, however, is on the extracted "**main**" plugins present in both 32-bit and 64-bit variants. In total, we obtained **19 distinct main plugins**, and their counts and names (based on VS project structure and compiled PE metadata) match the layout of the original development environment.

ValleyRAT Builder: Main Plugins

To verify the functionality of all 38 plugins (19×32-bit + 19×64-bit) along with the ValleyRAT rootkit driver, we automated the reverse-engineering workflow using two AI-assisted approaches: a **live IDA MCP server** and an **offline IDA export pipeline**, similar to the process described in our publication *Generative AI as a Force Multiplier for Reverse Engineering*. All automatically generated results were manually validated, with several plugins fully reverse engineered to investigate artifacts and noteworthy code paths highlighted by the AI methods.

All plugins are capable of establishing TCP or UDP connections to a specified C2 host and exchanging plugin-specific serialized data, typically encrypted using custom XOR-based schemes. The received data generally correspond to commands that trigger specific plugin functionality. The table below summarizes all available "**main**" plugins and their primary capabilities.

CN Name	EN Translation	Main Functionality
上线模块	Online Module	Initial-stage module with embedded C2 configuration + remote shellcode execution (in-process by default; optional `tracerpt.exe` injection)
登录模块	Login Module	Full-featured main-stage agent (login/recon, system fingerprinting, command processing, reflective plugin loading)
播放监听	Playback Monitoring	Remote audio-output capture module (system playback → transmission)
查注册表	Check Registry	Remote Regedit-like module (Windows Registry management)
视频查看	Video View	Remote webcam capture and streaming module
文件管理	File Management	Full-featured remote file management

CN Name	EN Translation	Main Functionality
远程交谈	Remote Chat	Screen-locking module with integrated remote chat
远程终端	Remote Terminal	Interactive remote shell module (`cmd.exe`)
语音监听	Voice Monitoring	Remote bidirectional-audio module (microphone capture and remote playback)
差异屏幕	Difference Screen	Classic, simple RDP-like control component
代理映射	Proxy Mapping	Multiplexed reverse-proxy module (tunneling multiple TCP/UDP connections over a single control channel)
键盘记录	Keylogger	Keylogger with clipboard harvesting
系统管理	System Management	Remote system-management module (reconnaissance, command processing, process/service control, code injection)
高速屏幕	High-speed Screen	High-frame-rate remote screen capture module
后台屏幕	Background Screen	Advanced hidden RDP-like remote desktop module with web-browser automation
娱乐屏幕	Entertainment Screen	Remote screen-capture video streaming (using libx264)
压力测试	Stress Test	Feature-rich remote DDoS module (supports multiple modes of TCP/UDP/HTTP/ICMP/RawIP floods)
shellcode	shellcode	Identical to the “Online Module”, compiled as PE `.exe`
驱动插件	Driver Plugin	Advanced driver-related module (rootkit loader and user-mode client)

ValleyRAT Builder: Auxiliary Plugins

The “**auxiliary**” plugins are not included in the compiled ValleyRAT builder, meaning their functionality could not be verified through reverse engineering of actual binaries. Instead, their expected behavior can only be inferred by analyzing the logical structure of the leaked Visual Studio solutions and the functionality implied by the associated project files. Because neither source code nor compiled versions were available, the listed capabilities remain educated assumptions based on naming conventions and references within the VS project structure.

CN Name	EN Translation	Likely Functionality
telegram打包上传	telegram Package Upload	Package Telegram data and upload it
telegram自动打包	telegram Auto-Pack	Automated packaging for Telegram
UACME	UACME	UAC-bypass framework/helper
企鹅解密	Penguin Decryption	Obtain decrypted QQ/Tencent data
体积膨胀	Volume Expansion	Binary bloating/padding (?)
内网主机扫描	Intranet Host Scan	LAN host discovery and scanning

CN Name	EN Translation	Likely Functionality
写启动目录	Write Startup Directory	Set up persistence in the Startup folder
写注册表启动	Write Registry Startup	Set up persistence via Registry autorun entries
删除360急速安全账号密码	Delete 360 Rapid Security Account Passwords	Remove stored credentials from 360 Total Security
删除chrome账号密码	Delete Chrome Account Passwords	Remove Chrome stored passwords
删除ie账号密码	Delete IE Account Passwords	Remove IE stored passwords
删除qq账号密码	Delete QQ Account Passwords	Remove QQ stored passwords
删除skype账号密码	Delete Skype Account Passwords	Remove Skype stored passwords
删除sogou账号密码	Delete Sogou Account Passwords	Remove Sogou stored passwords
删除telegram账号密码	Delete Telegram Account Passwords	Remove Telegram stored passwords
删除自身	Self Delete	Self-removal/uninstaller
微信解密	WeChat Decryption	Obtain decrypted WeChat databases
提权-CreateProcessInSession0	PrivilegeElevation-CreateProcessInSession0	Privilege elevation – create a process as SYSTEM
提权-EnableDebugPrivilege	PrivilegeElevation-EnableDebugPrivilege	Enable debug privilege
提权-RtlAdjustPrivilege	PrivilegeElevation-RtlAdjustPrivilege	Adjust privileges using RtlAdjustPrivilege
提权ShellExecuteEx	PrivilegeElevation-ShellExecuteEx	Privilege elevation using ShellExecuteEx
本地组策略添加启动	Add Startup via Local Group Policy	Set up persistence via Local Group Policy
桌面录制	Desktop Recording	Record the desktop to video
测试	Test	Test plugin
解密数据	Decrypt Data	Generic data-decryption helper
计划任务	Scheduled Task	Create scheduled tasks for persistence
高级浏览器解密	Advanced Browser Decryption	Decrypt browser-stored credentials (Chrome/Edge/IE)

With a clearer view of ValleyRAT's modular design, it is apparent that most plugins implement common backdoor functionality. From a research perspective, the most interesting components are those capable of providing rare or high-impact capabilities. For that reason, the next section focuses on the ValleyRAT **"Driver Plugin**", particularly its embedded kernel-mode rootkit.

Technical Analysis: ValleyRAT Rootkit Plugin

The ValleyRAT rootkit module is embedded inside one of the "main" plugins, originally named `驱动插件` (EN: `Driver Plugin.dll`). This plugin is compiled in both 32-bit and 64-bit variants and acts as the user-mode client and installer for the

rootkit. The embedded driver itself, however, is always a 64-bit kernel-mode binary. It functions as a Windows kernel device, a file system minifilter, a registry filter, and a process/thread monitoring driver.

ValleyRAT Rootkit Plugin: User-Mode Client

The user-mode client (`Driver Plugin.dll`) serves as the controller for the kernel rootkit. It maintains an active TCP/UDP connection to the C2 server and processes inbound commands. Each command is translated into an appropriate **IOCTL request**, which is then sent to the driver to control its runtime behavior. Supported operations include:

- **Driver installation** (**Normal** or **Stealth** mode)
- **Enable or disable the driver**
- **Query driver state**
- **Add or remove hidden objects** (files, directories, registry keys, and registry values)
- **Add or remove protected processes**
- **Enumerate protected objects**
- **Force-delete arbitrary files**
- **Trigger user-mode shellcode injection** via the rootkit driver
- **Update driver configuration values**

Driver Installation and Initial Configuration:

When executing the driver installation command in **Normal Mode** (calling `DropAndInstallRootkit()` directly), the client drops the embedded driver to disk and installs it as a kernel service named `kernelquick` , creating the corresponding key: `HKLM\SYSTEM\CurrentControlSet\Services\kernelquick\` . The service is registered as a `SERVICE_KERNEL_DRIVER` with **demand start**.


Figure 13: User-mode client – DropAndInstallRootkit() function.

During installation, the client writes the initial configuration values used by the driver to hide and protect itself:

- `KernelQuick_HideFsFiles` (list of files to hide)
- `KernelQuick_ProtectedImages` (process images to protect)

Additional configuration values can be set to control stealth and filtering behavior:

- `KernelQuick_State` (on/off for overall functionality)
- `KernelQuick_StealthMode` (whether to hide the driver/service)
- `KernelQuick_HideFsDirs` (list of directories to hide)
- `KernelQuick_HideRegKeys` (registry keys to hide)
- `KernelQuick_HideRegValues` (registry values to hide)
- `KernelQuick_IgnoredImages` (process images to ignore/exclude)
- `KernelQuick_hideFS_comprise` (additional file hiding rules)

All these configuration values define the hiding rules, protection lists, and ignore lists the driver uses. Any of them can be updated later through the corresponding IOCTL operations issued by the client (triggered by a command from the C2 server).

Driver Installation: Stealth Mode (MalSeclogon-Based):

In addition to the "**Normal Mode**" of the driver installation described above, the client can also trigger "**Stealth Mode**". In that case, the `DropAndInstallRootkit()` function is supplemented by additional routines: `GetProcID_dwm()` , `CreateProcessMalseclogon()` .


Figure 14: User-mode client – driver installation (Normal vs. Stealth).

This mode primarily aims to **disrupt network connectivity during installation** and use MalSeclogon-based **impersonation** to reduce detection likelihood. To disrupt network connectivity, the client launches commands such as `cmd /c start /min ipconfig /release` and `cmd /c start /min ipconfig /renew` . The MalSeclogon technique is then used to execute these commands under an impersonated context with **PPID spoofing**:

1. `GetProcID_dwm()` locates the PID of `dwm.exe` (Desktop Window Manager) using the `FILE_INFORMATION_CLASS::FileProcessIdsUsingFileInformation`. This approach reliably returns the correct PID without causing false positives.


Figure 15: User-mode client – GetProcID_dwm() function.

2. The client temporarily modifies the `TEB → ClientId.UniqueProcess` field to spoof the **PPID** to `dwm.exe`.
3. Token objects are stolen from the `dwm.exe` process and used to invoke commands via:
    - `CreateProcessWithTokenW` (primary)
    - `CreateProcessWithLogonW` (fallback, maintains **PPID** spoofing but **without impersonation**)


Figure 16: User-mode client – CreateProcessMalseclogon() function.

4. After the commands are executed, the client restores the original `UniqueProcess` value.
5. Driver installation via `DropAndInstallRootkit()` occurs during the network disruption window.

The result is a stealthy installation sequence executed under a trusted Windows process, significantly reducing behavioral detection signals. The following process tree shows how this activity appears when Stealth Mode is used:


Figure 17: User-mode client – CreateProcessMalseclogon() – PPID spoofing + impersonation.

User-Mode Shellcode Storage and Injection:

The client also supports **user-supplied shellcode injection**, storing the operator-provided shellcode inside `HKLM\SOFTWARE\IpDates`.


Figure 18: User-mode client – shellcode storage.

When commanded, the rootkit retrieves this user-mode shellcode and performs **APC-based injection**:

- Into `dwm.exe` during initial activation
- Into **any process ID** supplied via IOCTL

ValleyRAT Rootkit Plugin: Kernel-Mode Rootkit Driver

The embedded 64-bit driver is based on the publicly available open-source project Hidden. The ValleyRAT authors significantly modified the original codebase, introducing refactoring changes, compatibility improvements for recent Windows versions, and entirely new functionality not present in the original project. Like the original *Hidden* rootkit, the ValleyRAT driver acts as a kernel device, file system minifilter, registry filter, and process/thread monitoring driver.

During initialization, it creates a device named `HiddenGate` and assigns `IrpDeviceControlHandler()` as its device-control dispatcher for IOCTL communication with the user-mode client.


Figure 19: ValleyRAT rootkit – device creation. Differential Analysis Methodology:

Differential Analysis Methodology:

Because the ValleyRAT driver is derived from a publicly accessible codebase, we focused our reverse-engineering efforts on **only the modified functionality** to maximize efficiency.

To accomplish this, we:

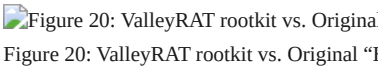
1. **Rebuilt the original Hidden rootkit** using the configuration extracted from the leaked ValleyRAT Visual Studio project.
2. Loaded the rebuilt driver into IDA, applied the PDB symbols, and created strict FLIRT signatures.
3. Generated a Diaphora database for structural diffing.
4. Loaded the ValleyRAT driver, applied the FLIRT signatures, and created its own Diaphora database.

5. Performed a differential analysis between the two drivers.

This approach allowed us to:

- Automatically match preserved functions
- Ignore superficial edits
- Highlight substantial refactoring
- Isolate genuinely new ValleyRAT functionality

Out of roughly 200 functions, only ~25 remained unmatched, representing the newly introduced ValleyRAT features. From this point on, we could concentrate exclusively on analyzing these new additions. A simplified overview of the modifications between the ValleyRAT rootkit and the original Hidden rootkit is shown below.


Figure 20: ValleyRAT rootkit vs. Original “Hidden” rootkit – XREF Tree.

Summary of Changes (ValleyRAT vs. Hidden rootkit)

Preserved functionality (from the original Hidden rootkit):

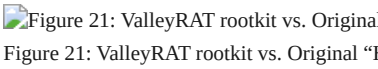
- Registry hiding (keys and values)
- File and directory hiding
- Process protection (setting limited access to processes)
- Process exclusion lists (exclude specific processes from protection features)

Removed functionality:

- Process hiding (unlinking from active process lists) – *Removed entirely due to BSOD risk triggered by modern Windows mitigations such as PageGuard*.

Configuration changes:

Registry keys used for initialization were renamed or reorganized. These values are remotely configurable via `Driver Plugin.dll`.


Figure 21: ValleyRAT rootkit vs. Original “Hidden” rootkit – configuration changes.

Added functionality:

1. `UMInjection()` – **APC-based user-mode shellcode injection**
2. `ForceDeleteFile()` – **kernel-level forced deletion of arbitrary files**
3. `SetDriverStartType_SystemStart()` – **elevated persistence by switching service start type**

Added functionality: UMInjection()

`UMInjection()` introduces kernel-mode to user-mode APC-based shellcode injection. It is invoked during driver initialization, creating a system thread that executes `UMInjectionRoutine()`.

`UMInjectionRoutine()`:

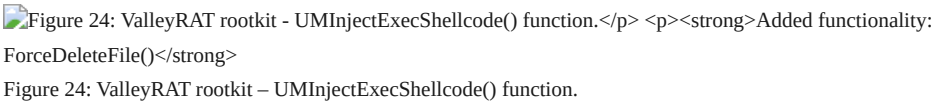
- Retrieves stored shellcode from `HKLM\SOFTWARE\IpDates`
- Locates `dwm.exe` (hardcoded target)
- Passes its PID to `UMInject()`


Figure 22: ValleyRAT rootkit – UMInjectionRoutine() function.

`UMInject()` can also be triggered directly via IOCTL `0x222144` to target any process. It locates a suitable thread, queues a **kernel-mode APC**, and triggers `UMInjectExecShellcode()`, which allocates user-mode memory, writes the shellcode, and queues a **user-mode APC** to execute it.


Figure 23: ValleyRAT rootkit – UMInject() function.

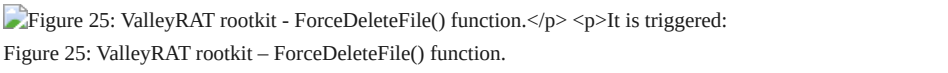
Figure 23: ValleyRAT rootkit – UMInject() function.


Figure 24: ValleyRAT rootkit - UMInjectExecShellcode() function. Added functionality: ForceDeleteFile()
Figure 24: ValleyRAT rootkit – UMInjectExecShellcode() function.

Added functionality: ForceDeleteFile()

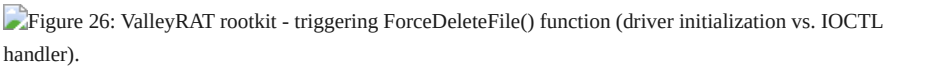
`ForceDeleteFile()` is a custom low-level re-implementation of file deletion using direct kernel IRP calls. **It:**

- Opens files via a custom `IRP_MJ_CREATE`
- Resets attributes via `IRP_MJ_SET_INFORMATION` (`FileBasicInformation`)
- Marks files for deletion via `FileDispositionInformation`
- Temporarily detaches section objects to bypass file locks, including memory-mapped executables


Figure 25: ValleyRAT rootkit - ForceDeleteFile() function. It is triggered:
Figure 25: ValleyRAT rootkit – ForceDeleteFile() function.

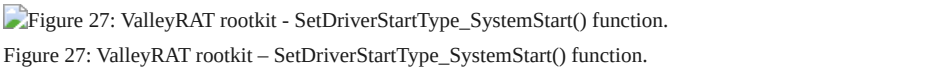
It is triggered:

- **Automatically** during driver initialization (see **Appendix A – Targeted Deletion of EDR/AV Drivers**)
- Via **IOCTL** `0x222140` from the user-mode client


Figure 26: ValleyRAT rootkit - triggering ForceDeleteFile() function (driver initialization vs. IOCTL handler).
Figure 26: ValleyRAT rootkit – triggering ForceDeleteFile() function (driver initialization vs. IOCTL handler).

Added functionality: SetDriverStartType_SystemStart()

This function updates the `kernelquick` service to use `SERVICE_SYSTEM_START`, elevating persistence from on-demand loading to loading at system startup.


Figure 27: ValleyRAT rootkit - SetDriverStartType_SystemStart() function.
Figure 27: ValleyRAT rootkit – SetDriverStartType_SystemStart() function.

In summary, the comparison between the original Hidden rootkit and the ValleyRAT-adapted variant shows that the authors did not rewrite the rootkit from scratch. Instead, they selectively refactored and modernized the existing codebase to ensure its continued viability on current Windows versions. Most changes revolve around structural refactoring, updated APIs, and compatibility adjustments that allow the rootkit to function reliably on Windows 10 and Windows 11 systems.

The newly introduced components—roughly 25 functions out of an otherwise large and legacy-heavy codebase—represent targeted functionality upgrades rather than a major redesign. These additions primarily reinforce the rootkit's integration with the wider ValleyRAT ecosystem, improve persistence and communication paths, and address stability issues caused by OS-level changes over the past decade.

Overall, the modifications reflect a pragmatic development approach: preserve the core functionality of a proven rootkit, update the parts that would break on modern systems, and extend the code just enough to support ValleyRAT's operational requirements. This strategy gives the actor a working kernel-mode stealth component while avoiding the engineering cost and detection risk associated with designing a new rootkit from scratch.

Conclusion

In this publication, we fully dissected the ValleyRAT modular system and mapped out every major component of its architecture. We analyzed all available plugins, documented their capabilities, and provided a comprehensive view of how they operate as part of a larger, well-structured backdoor ecosystem. By sharing these findings, we aim to strengthen the collective understanding of this widespread and actively abused malware family and contribute to better defensive measures across the security community.

Throughout the analysis, one theme remains consistent: the developers behind ValleyRAT possess a deep understanding of internal Windows mechanisms. Many plugins implement functionality that requires reversing complex kernel-mode and user-mode structures, undocumented behavior, and sensitive system interactions. The overall design shows a level of

consistency across different modules that strongly suggests a small, tightly coordinated development team rather than a loosely assembled collection of contributors.

A significant part of our research focuses on the **ValleyRAT kernel-mode rootkit** driver. We reverse engineered the **modified variant** used by ValleyRAT and compared it against the **original Hidden** rootkit. Despite relying on an older codebase, the actor successfully adapted it for modern Windows platforms through **refactoring** and compatibility updates. More concerning is the fact that we observed **several in-the-wild** samples **signed** with technically **valid** certificates, allowing the driver to load even on **fully updated Windows 11** systems with **all protections enabled**. This highlights a real-world security gap and demonstrates the ongoing operational capability of the threat actor's tooling.

The broader threat landscape reflects similar trends. More than **85%** of all ValleyRAT samples we observed appeared within the **last six months**, closely correlating with the period shortly after the builder was leaked. With the full build chain now publicly available, continued growth in **ValleyRAT activity** is not only expected but **likely inevitable**, especially as more actors experiment with the leaked tooling.

Finally, the public availability of both the builder and the source code **complicates attribution**. While ValleyRAT has historically been linked to Chinese-affiliated threat activity, including groups like Silver Fox, the current situation makes such **attribution unreliable**. Anyone can now compile, modify, and deploy ValleyRAT independently, blurring previous indicators and making traditional attribution approaches far less meaningful.

ValleyRAT has effectively transitioned from a previously actor-linked threat to an openly available malware framework with an active and accelerating presence in the wild. Our goal with this research is to provide defenders with the technical depth needed to **understand**, **detect**, and **counter** this evolving threat.

Protections

Check Point Threat Emulation and Harmony Endpoint provide comprehensive coverage of attack tactics, filetypes, and operating systems and protect against the attacks and threats described in this report.

References

CPR – Chasing the Silver Fox: Cat & Mouse in Kernel Shadows: https://research.checkpoint.com/2025/silver-fox-apt-vulnerable-drivers/

CPR – Generative AI as a Force Multiplier for Reverse Engineering: https://research.checkpoint.com/2025/generative-ai-for-reverse-engineering/

ValleyRAT builder: https://github.com/GkaMei/winos4.0

ValleyRAT development structure: https://github.com/Logkiss/Rat-winos4.0-gh0st/tree/master/银狐Winos

Original "Hidden" rootkit driver: https://github.com/JKornev/hidden

Sun-RAT company website: https://www.sun-rat.com/

Detect-It-Easy tool: https://github.com/horsicq/Detect-It-Easy

Resource Hacker tool: https://www.angusj.com/resourcehacker/

LIEF: https://lief.re/

Microsoft Driver Signing Policy – Exceptions: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-#exceptions

Microsoft Vulnerable Driver Blocklist: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/microsoft-recommended-driver-block-rules

MalSeclogon: https://github.com/antonioCoco/MalSeclogon

Diaphora IDA plugin: https://github.com/joxeankoret/diaphora

Appendix A – Targeted Deletion of EDR/AV Drivers

Table providing details about the EDR/AV drivers targeted by `ForceDeleteFile()` during the ValleyRAT rootkit driver initialization.

System Path	Company
C:\Windows\System32\drivers\DsArk64.sys	Qihoo 360
C:\Windows\System32\drivers\360AntiSteal64.sys	Qihoo 360
C:\Windows\System32\drivers\360FsFlt.sys	Qihoo 360
C:\Windows\System32\drivers\360netmon.sys	Qihoo 360
C:\Windows\System32\drivers\360AntiAttack64.sys	Qihoo 360
C:\Windows\System32\drivers\360AntiHijack64.sys	Qihoo 360
C:\Windows\System32\drivers\360AntiExploit64.sys	Qihoo 360
C:\Windows\System32\drivers\360AntiHacker64.sys	Qihoo 360
C:\Windows\System32\drivers\BAPIDRV64.sys	Qihoo 360
C:\Windows\System32\drivers\360reskit64.sys	Qihoo 360
C:\Windows\System32\drivers\360qpesv64.sys	Qihoo 360
C:\Windows\System32\drivers\360Sensor64.sys	Qihoo 360
C:\Windows\System32\drivers\360Box64.sys	Qihoo 360
C:\Program Files (x86)\360\360Safe\deepscan\AtS64.sys	Qihoo 360
C:\Windows\System32\drivers\sysdiag_win10.sys	Huorong Security
C:\Windows\System32\drivers\hrwfpdrv_win10.sys	Huorong Security
C:\Windows\System32\drivers\sysdiag.sys	Huorong Security
C:\Windows\System32\drivers\hrwfpdrv.sys	Huorong Security
C:\Windows\Windows\System32\drivers\hrdevmon_win10.sys	Huorong Security
C:\Windows\Windows\System32\drivers\nxeng.sys	Huorong Security
C:\Windows\System32\drivers\TAOAcceleratorEx64_ev.sys	Tencent
C:\Windows\System32\drivers\TAOAccelerator64.sys	Tencent
C:\Windows\System32\drivers\qmbsecx64.sys	Tencent
C:\Windows\System32\drivers\TFsFltX64_ev.sys	Tencent
C:\Windows\System32\drivers\TAOKernel64.sys	Tencent
C:\Windows\System32\drivers\ksapi64.sys	Kingsoft Corporation
C:\Program Files (x86)\kingsoft\kingsoft antivirus\security\kxescan\kdhacker64_ev.sys	Kingsoft Corporation
C:\Program Files (x86)\kingsoft\kingsoft antivirus\security\kxescan\kdhacker64.sys	Kingsoft Corporation
C:\Windows\System32\drivers\kavbootc64_ev.sys	Kingsoft Corporation

System Path	Company
C:\Windows\System32\drivers\KAVBootC64.sys	Kingsoft Corporation
C:\Windows\System32\drivers\kisknl.sys	Kingsoft Corporation
C:\Windows\System32\drivers\bootsafe64.sys	Kingsoft Corporation
C:\Windows\System32\drivers\ksthlp64.sys	Kingsoft Corporation
C:\Program Files (x86)\kingsoft\kingsoft antivirus\security\ksde\kisnetflt64.sys	Kingsoft Corporation
C:\Program Files (x86)\kingsoft\kingsoft antivirus\security\ksnetm\kisnetm64.sys	Kingsoft Corporation
C:\Program Files (x86)\kingsoft\kingsoft antivirus\khwinfo64.sys	Kingsoft Corporation
C:\Windows\System32\drivers\klgse.sys	Kaspersky Lab
C:\Windows\System32\drivers\klhk.sys	Kaspersky Lab
C:\Windows\System32\drivers\klflt.sys	Kaspersky Lab
C:\Windows\System32\drivers\klif.sys	Kaspersky Lab
C:\Windows\System32\drivers\klwtp.sys	Kaspersky Lab
C:\Windows\System32\drivers\klim6.sys	Kaspersky Lab
C:\Windows\System32\drivers\klkbdflt2.sys	Kaspersky Lab
C:\Windows\System32\drivers\cm_km.sys	Kaspersky Lab
C:\Windows\System32\drivers\kldisk.sys	Kaspersky Lab
C:\Windows\System32\drivers\klwfp.sys	Kaspersky Lab
C:\Windows\System32\drivers\kneps.sys	Kaspersky Lab
C:\Windows\System32\drivers\klpd.sys	Kaspersky Lab
C:\Windows\System32\drivers\klupd_klif_arkmon.sys	Kaspersky Lab
C:\Windows\System32\drivers\klupd_klif_klbg.sys	Kaspersky Lab
C:\Windows\System32\drivers\klbackupdisk.sys	Kaspersky Lab
C:\Windows\System32\drivers\klupd_klif_klark.sys	Kaspersky Lab
C:\Windows\System32\drivers\klupd_klif_mark.sys	Kaspersky Lab
C:\ProgramData\Kaspersky Lab\AVP21.3\Bases\klids.sys	Kaspersky Lab

Appendix B – IOCs

File Name (EN)	Purpose	Arch	SHA-256 Hash
Background Screen.dll	Main Plugin	32-bit	7f5bad67cec7492b023ca08e8fa3ed5db9eb186fab0472b34993fe3

File Name (EN)	Purpose	Arch	SHA-256 Hash
Background Screen.dll	Main Plugin	64-bit	a57dd44b7bc6233496657867cf053199213289f58c1c3c8d4eb565
Check Registry.dll	Main Plugin	32-bit	74d70f53748125eb4439cb790817fb1d0e9159f75c7dd5148444f5(
Check Registry.dll	Main Plugin	64-bit	0a6376107abdf30ea14f4bdaf785b2db7d18e0818bd332511dcce38
Difference Screen.dll	Main Plugin	32-bit	ed4a064ef099e0ea40faf4b1e3618f20c52833b148ae578f80f09eab
Difference Screen.dll	Main Plugin	64-bit	2c34d8fc0881d3cd4fb693fc5fe2edf405b8424174d3dbb800385fd7
Driver Plugin.dll	Main Plugin	32-bit	79daa001c67dc83bdd6189417ccf4bf83ea5da4c6211bbac91c1d7d
Driver Plugin.dll	Main Plugin	64-bit	14b85b07bfdd134e709ff973871d75d33ecca964457373b76b34a7(
Driver Plugin.dll_Driver Functions.sys_KernelQuick_Hidden.sys	Rootkit Driver	64-bit	2aa029088c04eb10b056c18fcc39395936e6f01ee9ebdeed2558e48
Entertainment Screen.dll	Main Plugin	32-bit	1bd71ea3b9409a6e86fac12039258f8ed8b59261ff2509673544e4a
Entertainment Screen.dll	Main Plugin	64-bit	13d7380344bf1f9e17e8970c01127a2fe2528d3e640b36ef478ccd4
File Management.dll	Main Plugin	32-bit	61598b986aeaeb24d7565a7bb3a113e61f88b4d4c6169d2bd7fd0b5
File Management.dll	Main Plugin	64-bit	dee2b2da6b917d2dc7d3dcbbd3c505dd4f128c07059659f9e89100(
High-speed Screen.dll	Main Plugin	32-bit	746f2d5d727511c1bd1ad936f35ac0851a520aadcf201f0d5e23dc6(
High-speed Screen.dll	Main Plugin	64-bit	9dd0e7dccc7105a30b3a71f10126be4ee5a8e770e743fc4f0bbea0e4
Keylogger.dll	Main Plugin	32-bit	96c54665cda4f04e9ff60faebcd993d0cf98988258249d9e00fe563b
Keylogger.dll	Main Plugin	64-bit	aaf8258585d086cce588a3e870eb485270ee135087eee9ef8766db9
Login Module.dll	Main Plugin	32-bit	90f24d6175e1b5fac4e2844e77554ff03dec2174f18c07c008699af5
Login Module.dll	Main Plugin	64-bit	6f79ee17dbb75d1ed7e0535a7b498c2249d538c0836d6ecee16fec4
Online Module.dll	Main Plugin	32-bit	9f456f3125d7f6ce907e13ec637b9b8c6e4a43b1c9f352d233cfebbc
Online Module.dll	Main Plugin	64-bit	93e75eada1b8f155bdb41c1af0f7d7ea390b280c6f49c8834c11af2e
Playback Monitoring.dll	Main Plugin	32-bit	054a22279de7a8c0fd75a72b39648dd2429bef07c268756087ed96

File Name (EN)	Purpose	Arch	SHA-256 Hash
Playback Monitoring.dll	Main Plugin	64-bit	860acd2b9aec21cf03e1c5ec8f79b1ef4e7b78eb9ba7a6c0a915586
Proxy Mapping.dll	Main Plugin	32-bit	5dcde82f7a2db50dddf9b42dab3e3affabedfe237d7c956a1de660a7
Proxy Mapping.dll	Main Plugin	64-bit	4d0517229ef88f2410a2a1983eaf4036872911c8cf31c3ceb38c112
Quick.exe	Builder/C2 Panel	32-bit	9e82fe6322585d613c8409fa445394e2e38f24ef85733b8dafcfa3ce
Remote Chat.dll	Main Plugin	32-bit	9ec3c31ca3bcdd4597d3e928e36fb0202a5111da7e5d169c58bd97b
Remote Chat.dll	Main Plugin	64-bit	35fbedfafa9a2267d8eab711ce0e9db66dca304a4b4379d7a965ce38
Remote Terminal.dll	Main Plugin	32-bit	85296ee0d867175da1b790f472824f6e702930676aa9b41c4f40f62
Remote Terminal.dll	Main Plugin	64-bit	e22fb0c295eefaeb4b25a0b9038a0c60cec9389b894fa22902a7122
shellcode.exe	Main Plugin	32-bit	5e4085553f083d1fd31d673f0746670dfc1f9ebb9911f2fe754e59d9
shellcode.exe	Main Plugin	64-bit	dca90d7d9e5770acbd991af69bafa80fe596430c29c78d5036a8fb08
Stress Test.dll	Main Plugin	32-bit	d17bf1c3d50bf4acba18418b0cdcc524be268848b15542e4895a74c
Stress Test.dll	Main Plugin	64-bit	55c07dd40ffcf07d569b8b762513cdbfc51e7a4c77ce66135247945
System Management.dll	Main Plugin	32-bit	7c9554c18a6b8fe87a570dd5cd5a0f041a782fc2424ab02ac675e47
System Management.dll	Main Plugin	64-bit	e60298307befa4b22eeedef02019a39c93729567fcd4a7745350fd2
Video View.dll	Main Plugin	32-bit	05e578a967168b704d8bdcba95a8d69fdda25854263e037990add0
Video View.dll	Main Plugin	64-bit	e19ae27f03c252d4e7b44c462a4edaa1ae759888bcd25cb7863c3c0
Voice Monitoring.dll	Main Plugin	32-bit	a38b91c061157011a00d29c5e3169fbf2b29c0b0cacc0153dc0cf99
Voice Monitoring.dll	Main Plugin	64-bit	b5949ce7e802740e9548ba83ccdb20470fa405fdf9866b3fc3f85a3

Source: https://research.checkpoint.com/2025/cracking-valleyrat-from-builder-secrets-to-kernel-rootkits/