

SilverFox Evolves: Trusted Software Hijacking Delivers ValleyRAT in Japan

 catonetworks.com/blog/cato-ctrl-silverfox-evolves

Tomer Pugach

July 28, 2026

Table of Contents

- [1. Executive Summary](#)
- [2. Key Findings](#)
- [3. SilverFox Expands Its Arsenal](#)
- [4. Attribution Assessment](#)
- [5. Technical Analysis Details](#)
- [6. Timestamped Files Complicate Static Detection](#)
- [7. Detection Opportunities](#)
- [8. How Cato Blocks the SilverFox Attack Chain](#)
- [9. Coordinated Disclosure and Infrastructure Abuse Reporting](#)
- [10. Research Scope](#)
- [11. Conclusion](#)
- [12. Indicators of Compromise](#)

Wondering where to begin your SASE journey?

[We've got you covered!](#)

Listen to post:

 This audio player requires that "Preferences" cookies be accepted

Executive Summary

SilverFox is expanding its toolkit. In this campaign, the group combines new vulnerable-driver abuse, newly observed abuse of legitimate applications for DLL sideloading, defense evasion, and layered recovery mechanisms to keep ValleyRAT running.

We investigated an active campaign targeting a Japanese organization in the industrial manufacturing sector. The attack begins with an invoice-themed phishing lure and uses attacker-controlled content hosted through legitimate QQ and Tencent Cloud services. The attackers then abuse *ConvertToPDF.exe* and *PDFDirect.exe* to sideload a malicious *PDFCORE8.dll*. Based on the public research we reviewed, neither application had previously been documented as a DLL-sideloading host.

The malicious DLL embeds *BootRepair.sys*, *EnPortv.sys*, and *wsftprm.sys*. Although SilverFox's use of vulnerable drivers is not new, we found no prior public association between the group and *BootRepair.sys* or *EnPortv.sys*. The malware integrates Bring Your Own Vulnerable Driver (BYOVD), DLL sideloading, NTDLL unhooking, process injection, registry-based payload storage, and two independent recovery mechanisms to impair security controls and maintain execution. The Cato SASE Platform blocked the malicious *PDFCORE8.dll* before ValleyRAT could establish remote access.

[2026 Cato CTRL™ Threat Report](#) | [Download the report](#)

Key Findings

- We identified previously undocumented SilverFox use of *BootRepair.sys* and *EnPortv.sys*.
- SilverFox abused *ConvertToPDF.exe* and *PDFDirect.exe* to sideload the malicious *PDFCORE8.dll*.
- The loader embeds three vulnerable drivers in a reusable BYOVD framework designed to terminate security processes from kernel mode.
- The malware integrates BYOVD, NTDLL unhooking, thread-context hijacking, registry storage, scheduled-task persistence and dual recovery mechanisms.
- The campaign targeted a Japanese organization in the industrial manufacturing sector and ultimately deployed ValleyRAT for remote access.

SilverFox Expands Its Arsenal

SilverFox has previously used BYOVD techniques to impair security products and facilitate ValleyRAT deployment. [Check Point Research](#) documented a SilverFox campaign using a dual-driver framework that included a previously unclassified WatchDog Antimalware driver capable of terminating security processes from kernel mode.

Our investigation identified a different driver set. The malicious *PDFCORE8.dll* embeds *BootRepair.sys*, *EnPortv.sys*, and *wsftprm.sys* and manages them through shared resource-decryption, driver-deployment, service-creation, and device-communication routines.

The process-termination capabilities of these drivers were already known independently of this campaign. [Huntress](#) previously analyzed abuse of *EnPortv.sys* in an unrelated intrusion. [PhantomKiller](#) demonstrated how *BootRepair.sys* can receive a process identifier and invoke kernel-level process termination. [CVE-2023-52271](#) documents how an affected version of *wsftprm.sys* can be abused to terminate protected processes.

Our research expands the group’s documented arsenal and reveals how several defense-evasion and execution-resilience capabilities are combined within a single framework.

Table 1. New additions to SilverFox’s documented arsenal

Cato CTRL Finding	Threat Research Significance
Previously undocumented SilverFox use of BootRepair.sys and EnPortv.sys	Adds two new driver associations to SilverFox’s known BYOVD arsenal.
Previously undocumented abuse of ConvertToPDF.exe and PDFDirect.exe	Identifies two legitimate applications used by SilverFox to sideload the malicious PDFCORE8.dll.
Integrated three-driver BYOVD framework	Shows how SilverFox manages BootRepair.sys, EnPortv.sys, and wsftprm.sys through reusable decryption, installation, service-management, and driver-communication logic.
Unified defense-evasion and recovery architecture	Combines BYOVD, NTDLL unhooking, dynamic API resolution, thread-context hijacking, internal payload supervision, and an external watchdog in one modular execution framework.

The novelty is therefore not any single technique in isolation. It is the addition of two drivers not previously associated with SilverFox, two newly documented DLL-sideloaded hosts, and the integration of kernel-level defense impairment, user-mode evasion, payload injection, and dual recovery mechanisms into one resilient attack chain targeting Japan.

Attribution Assessment

We assess with moderate-to-high confidence that this activity is linked to SilverFox. The assessment is based on the use of ValleyRAT, BYOVD-based security-process termination, DLL sideloading, the FaCai2024 marker, registry-based payload storage, related infrastructure patterns, and targeting consistent with previously reported SilverFox activity. No single artifact is treated as conclusive; the attribution is based on the combined overlap in tooling, execution flow, infrastructure, and victimology.

Technical Analysis Details

From an Invoice Lure to Trusted Execution

The campaign, initially shared by [MisaKi](#) on X, begins with an invoice-themed phishing email sent to a Japanese organization in the industrial manufacturing sector. When the recipient attempts to open the supposed invoice, they are directed to download a ZIP archive hosted through legitimate QQ file-delivery infrastructure.

The observed download originated from:

file[.]wx2[.]qq[.]com

At the time of publication, the specific malicious resource was no longer available. The threat actor abused the service to make the delivery traffic appear less suspicious and more difficult to distinguish from normal web activity.

The archive contains a downloader executable that retrieves the next-stage components from an attacker-controlled object hosted through Tencent Cloud infrastructure:

hrefbfdhfhgre-1422102728[.]cos[.]ap-hongkong[.]myqcloud[.]com

The retrieved files include:

- *PDFCORE8.dll* – the malicious second-stage DLL
- *ConvertToPDF.exe* or *PDFDirect.exe* – legitimate applications later used as DLL-sideloaded hosts

The specific malicious Tencent Cloud object was also unavailable when we completed this analysis.

As shown in Figure 1, the infection progresses from a phishing lure and cloud-hosted delivery to trusted software hijacking, kernel-level defense impairment, process injection, and ValleyRAT execution.

SilverFox Evolves: Cato CTRL Identifies New Additions to SilverFox's BYOVD Arsenal Targeting Japanese Organizations

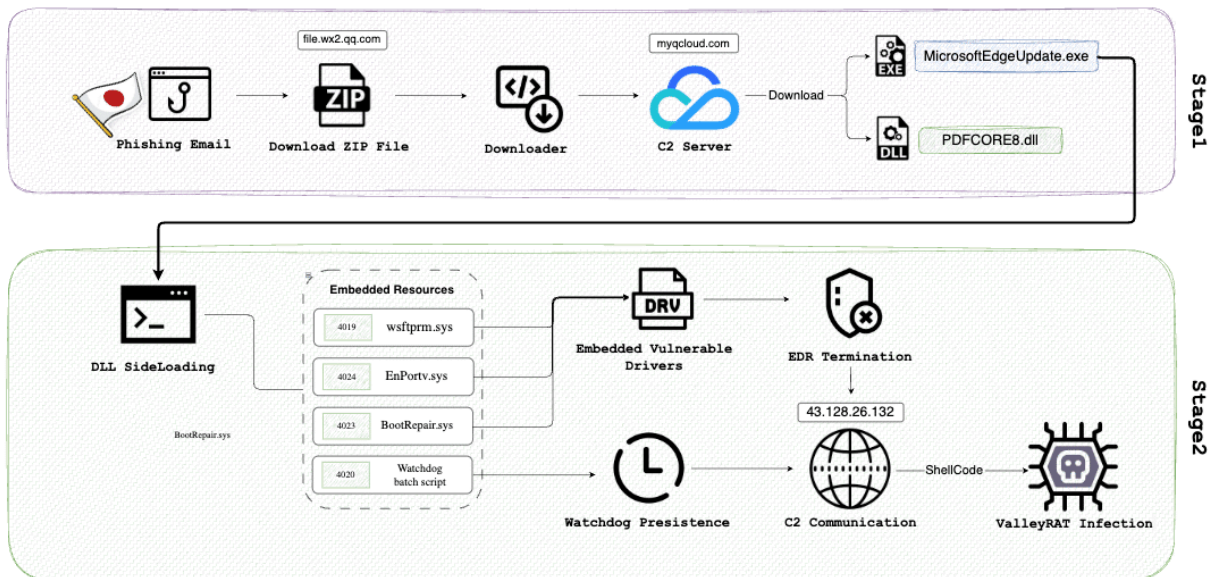


Figure 1. SilverFox infection chain from phishing delivery to ValleyRAT execution

The downloader retrieves the malicious PDFCORE8.dll together with a legitimate application that is later used as the DLL-sideloaded host. As shown in Figure 2, both components are delivered from the second-stage infrastructure and written to the compromised system.

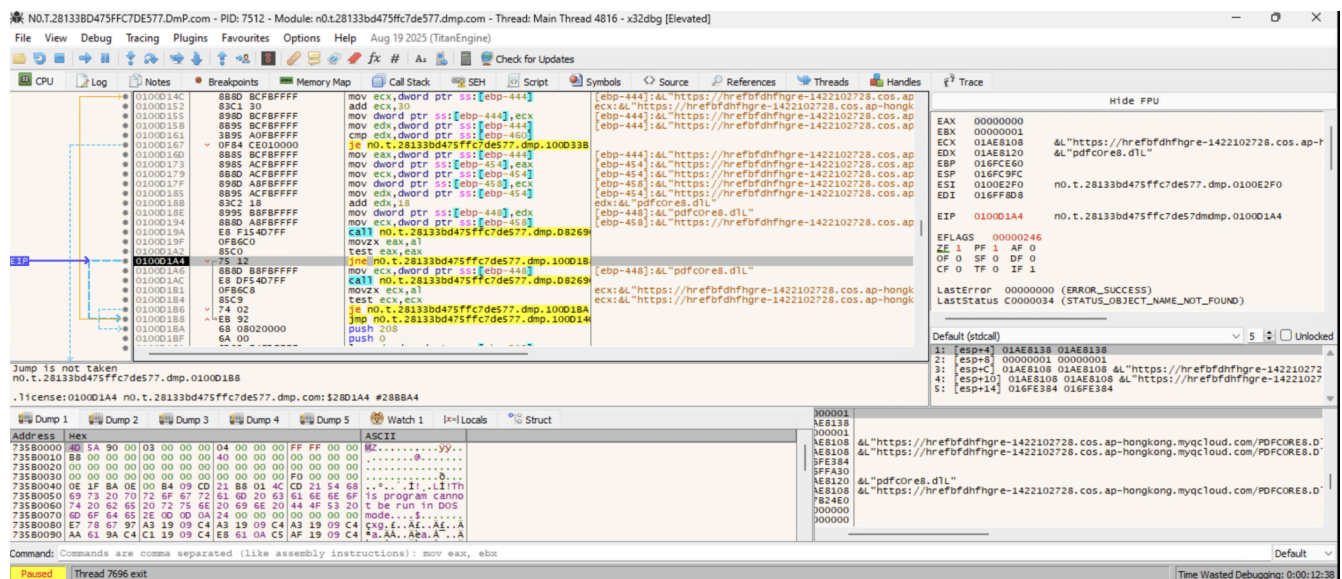


Figure 2. Components delivered for DLL side-loading

Timestamped Files Complicate Static Detection

During multiple investigations, we observed timestamp-based filenames used for ZIP archives and executable payloads. One example was:

20260608154418.zip

The filename appears to encode the date and time at which the archive was generated. Changing filenames can reduce the effectiveness of simple filename-based blocking and make separate deliveries appear unrelated, even when the underlying payload or infrastructure remains similar.

In Figure 3, we show the timestamped archive and executable together with the next-stage files retrieved from the delivery server.

Showing 7 entries in 7 events			
Time	Full Path URL	File Name	Domain Name
+ 2026/06/08 02:39:05.409	https://hrefbfdhfhgre-1422102...	PDFCORE8.Dll	hrefbfdhfhgre-1422102728.cos....
+ 2026/06/08 02:39:04.593			hrefbfdhfhgre-1422102728.cos....
+ 2026/06/08 02:39:03.861	https://hrefbfdhfhgre-1422102...	20260607154418.exe	hrefbfdhfhgre-1422102728.cos....
+ 2026/06/08 02:39:03.001			hrefbfdhfhgre-1422102728.cos....
+ 2026/06/08 02:35:39.658	https://file.wx2.qq.com/cgi-b...	20260608154418.zip	file.wx2.qq.com
+ 2026/06/08 02:35:38.018			file.wx2.qq.com
+ 2026/06/08 02:35:38.017			file.wx2.qq.com

Figure 3. Timestamped archive and next-stage payload files

The downloader also impersonates Google Chrome during network communication by using a browser-like User-Agent string. This allows the traffic to resemble routine browser activity rather than communication generated by a standalone executable.

Figure 4 shows the Chrome User-Agent string used by the downloader during communication with the delivery infrastructure.

Attack Related Events									
Search All targets									
Full Path URL	HTTP Response Code	Client	Cato App	Referrer	User Agent	Destination Country	DNS response...	File Name (SMB)	
https://hrefbdfhgre-1422102728.ccs.ap-hongkong.myqcloud.com/PDFCORE8.DII	200	chrome	tencent	-	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/148.0.0.0 Safari/537.36	Hong Kong	-	PDFCORE8.DII	
https://hrefbdfhgre-1422102728.ccs.ap-hongkong.myqcloud.com/PDFCORE8.DII	200	chrome	tencent	-	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/148.0.0.0 Safari/537.36	-	-	PDFCORE8.DII	

Figure 4. Spoofed Chrome User-Agent used by the downloader

Trusted Software Hijacking Through DLL Sideloadng

The second stage begins when the downloader writes the legitimate executable and malicious DLL to the user's temporary directory. In some observed samples, the executable was renamed:

MicrosoftEdgeUpdate.exe

The renamed file was not a Microsoft binary. It was a legitimate signed application placed under a name intended to resemble a familiar Microsoft update component and reduce suspicion.

Figure 5 shows the renamed executable as it appeared on the affected system.

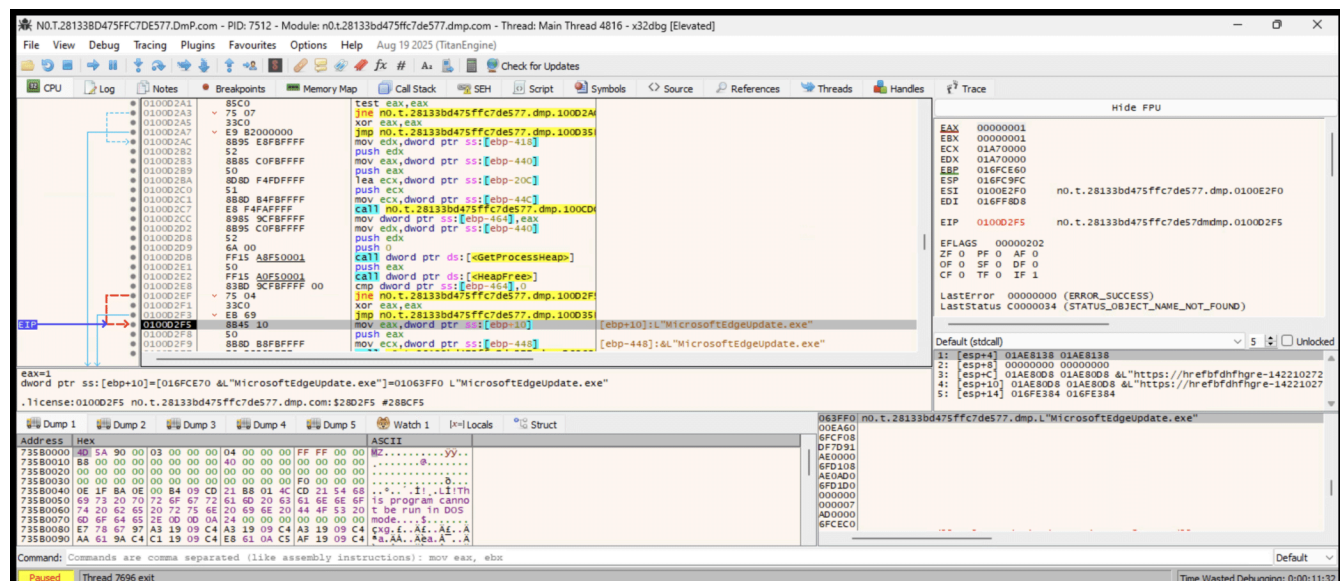


Figure 5. Legitimate application renamed to resemble a Microsoft update component

During our investigation, we identified two separate legitimate applications used through the same DLL-sideloadng pattern:

- *ConvertToPDF.exe*

- *PDFDirect.exe*

Both applications are associated with Zeon Corporation. The threat actor placed a malicious implementation of *PDFCORE8.dll* in the same directory as the trusted executable. When either application started, Windows resolved the DLL dependency from the local working directory and loaded the attacker-controlled file.

Authenticode verification showed that the legitimate executables remained digitally signed and were not modified. The malicious behavior resulted from the way the applications located and loaded *PDFCORE8.dll*, not from alteration of the signed executable itself.

Based on the public research we reviewed, neither *ConvertToPDF.exe* or *PDFDirect.exe* had previously been documented as a DLL-sideloaded host.

This distinction is important: the applications themselves are legitimate and our findings describe their abuse by the threat actor. We are not suggesting that Zeon participated in or knowingly enabled the campaign.

In Figure 6, we show the valid digital signature associated with the abused *ConvertToPDF.exe* sample.

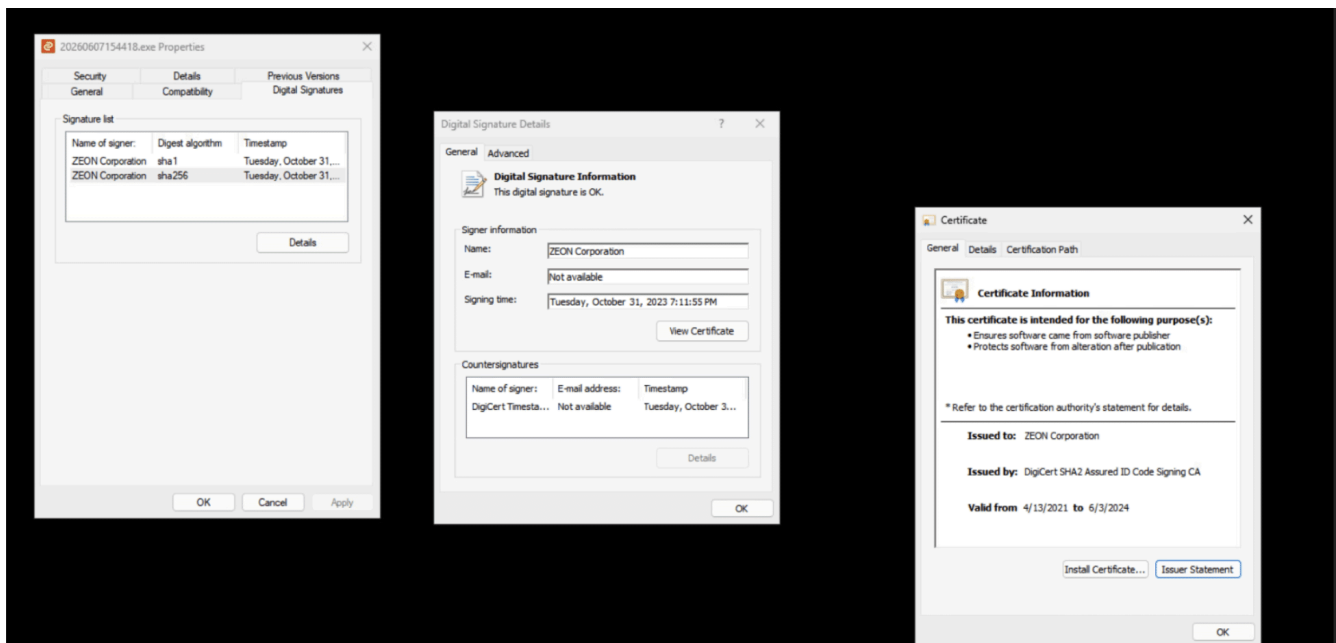


Figure 6. Valid digital signature on the abused *ConvertToPDF.exe* application

Figure 7 shows the DLL imports of the legitimate application, including *PDFCORE8.dll*, which is abused for DLL sideloading.

CFF Explorer VIII - [20260607154418.exe]

File Settings ?

20260607154418.exe

File: 20260607154418.exe

- Dos Header
- Nt Headers
- File Header
- Optional Header
- Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- Debug Directory
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor
- UPX Utility

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
002B6760	N/A	002B30DC	002B30E0	002B30E4	002B30E8	002B30EC
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ole32.dll	36	002B5090	00000000	00000000	002B79A2	00222B2C
OLEAUT32.dll	20	002B4B34	00000000	00000000	002B79AC	002225CC
oledlg.dll	1	002B5124	00000000	00000000	002B79C8	00222BC0
msi.dll	1	002B5088	00000000	00000000	002B7AC2	00222B24
WININET.dll	12	002B4FC0	00000000	00000000	002B7ACA	00222A58
Secur32.dll	1	002B4C10	00000000	00000000	002B7B14	002226A8
PDFCore8.dll	4	002B4B88	00000000	00000000	002B7B60	00222620
RPCRT4.dll	3	002B4B9C	00000000	00000000	002B7BA8	00222634
gdiplus.dll	24	002B5024	00000000	00000000	002B7DDA	00222AC0
OLEACC.dll	3	002B4B24	00000000	00000000	002B7E34	002225BC
IMM32.dll	3	002B47C0	00000000	00000000	002B7E78	00222258
WINMM.dll	1	002B4FF4	00000000	00000000	002B7E90	00222A8C

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
002B7B4A	002B7B4A	0561	RestorePlugInFrame
002B7B3C	002B7B3C	0000	CoreLibFin
002B7B2E	002B7B2E	0511	GetCoreHFT
002B7B20	002B7B20	0002	CoreLibInit

Figure 7. *PDFCORE8.dll* listed among the legitimate application's imported DLLs

Inside PDFCORE8.dll: A Self-Contained Execution Framework

Once loaded, *PDFCORE8.dll* acts as the main second-stage execution framework. Rather than retrieving every component from external infrastructure, the DLL stores several encrypted resources inside its Portable Executable resource section.

This design reduces the number of additional network requests required after the initial compromise. It also delays exposure of the embedded components until the malware needs them, limiting what static analysis tools can observe before runtime.

Our analysis identified four encrypted resources:

Resource ID	Component	Purpose
4019	wsftprm.sys	Vulnerable kernel driver
4020	Watchdog batch script	Loader supervision and recovery

Resource ID	Component	Purpose
4023	BootRepair.sys	Vulnerable kernel driver
4024	EnPortv.sys	Vulnerable kernel driver

None of these resources are stored in plaintext. The framework retrieves each component from the PE resource section, decrypts it immediately before use, writes it to disk only when necessary, executes or loads it, and later removes it when the corresponding stage is complete.

Figure 8 shows the encrypted resources embedded within *PDFCORE8.dll*.

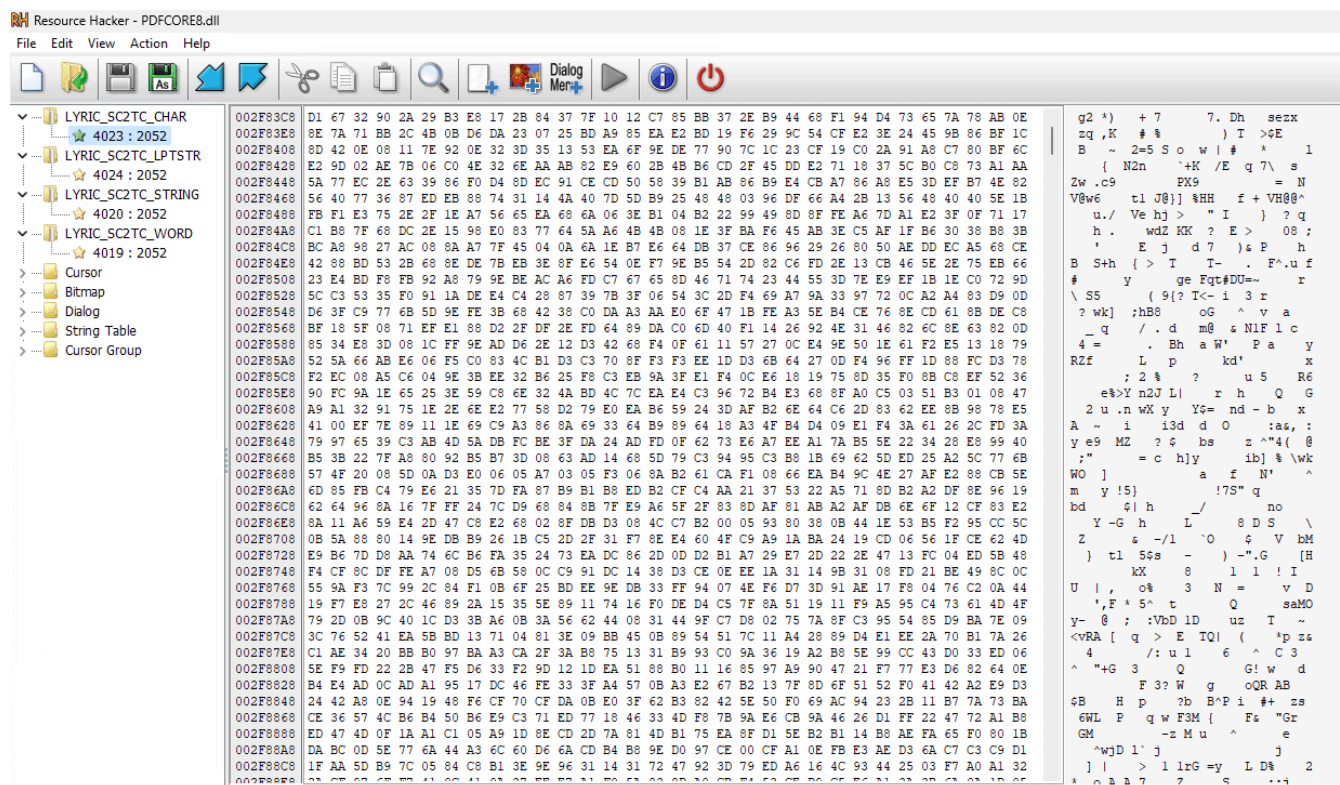


Figure 8. Encrypted drivers and watchdog resources embedded in *PDFCORE8.dll*

The embedded resources include all three vulnerable drivers supported by the framework. Figure 9 shows *BootRepair.sys* and *EnPortv.sys* within *PDFCORE8.dll*, while Figure 10 shows the embedded *wsftprm.sys* resource.

```

    2e 00 65 ...
1019459a 00      ??      00h
1019459b 00      ??      00h

    u_\\.\\BootRepair_1019459c      XREF[1]:      FUN_100145a0:100145c2(*)
1019459c 5c 00 5c      unicode      u"\\\\.\\BootRepair"
    00 2e 00
    5c 00 42 ...
101945ba 00      ??      00h
101945bb 00      ??      00h

    u_\\.\\EnPortv_101945bc      XREF[1]:      FUN_10014560:10014582(*)
101945bc 5c 00 5c      unicode      u"\\\\.\\EnPortv"
    00 2e 00
    5c 00 45 ...

```

Figure 9. BootRepair.sys and EnPortv.sys embedded in PDFCORE8.dll

```

    2e 00 65 ...
1019453a 00      ??      00h
1019453b 00      ??      00h

    u_\\.\\Warsaw_PM_1019453c      XREF[1]:      FUN_100145e0:10014602(*)
1019453c 5c 00 5c      unicode      u"\\\\.\\Warsaw_PM"
    00 2e 00
    5c 00 57 ...

```

Figure 10. wsftprm.sys embedded in PDFCORE8.dll

On-Demand RC4 Resource Decryption

The embedded resources are protected with RC4 and remain encrypted until the execution framework requires them.

We recovered the following hardcoded 128-bit key from the loader:

BB7BBB62FD9C76D5DDF37F17DDC3E7FF

The malware does not rely on Windows cryptographic APIs to perform decryption. Instead, it includes its own RC4 implementation, including routines corresponding to the Key Scheduling Algorithm and Pseudo-Random Generation Algorithm.

The principal routines identified during reverse engineering were:

Routine	Responsibility
FUN_10013E60()	Resource extraction
FUN_10014900()	RC4 Key Scheduling Algorithm

Routine

Responsibility

FUN_100149C0() RC4 Pseudo-Random Generation Algorithm

The loader decrypts each resource on demand rather than decrypting all embedded components during initialization. This allows drivers and scripts to remain concealed until the malware reaches the relevant stage of execution.

The recovered implementation follows the standard RC4 algorithm and does not appear to contain custom cryptographic modifications. In this campaign, RC4 serves primarily as an obfuscation layer that delays static exposure of the embedded components.

In Figure 11, we show the routine used to retrieve and decrypt an embedded resource immediately before use.

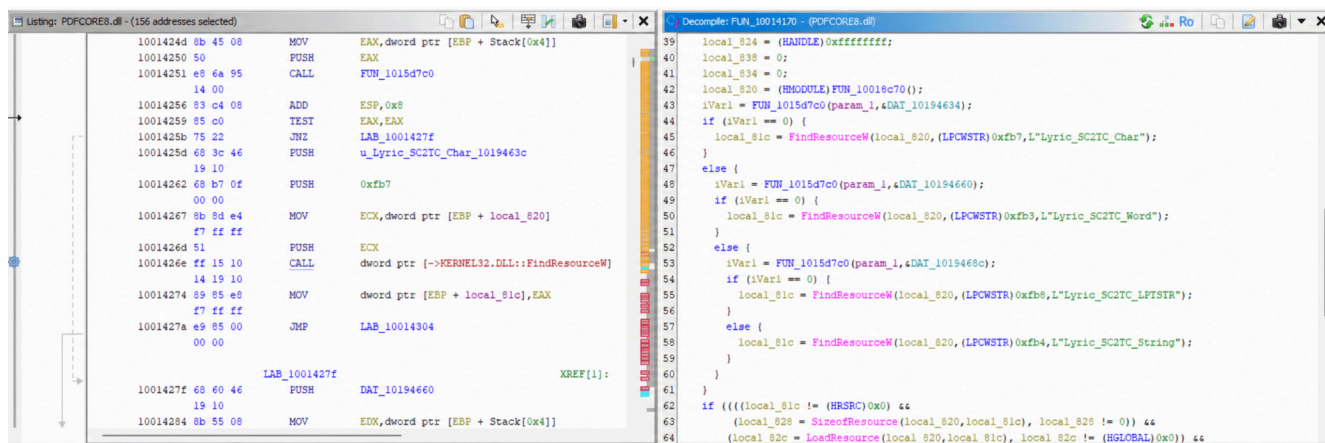


Figure 11. On-demand RC4 decryption of embedded resources

Figure 12 shows our validation of the recovered RC4 implementation and the resulting plaintext resource.

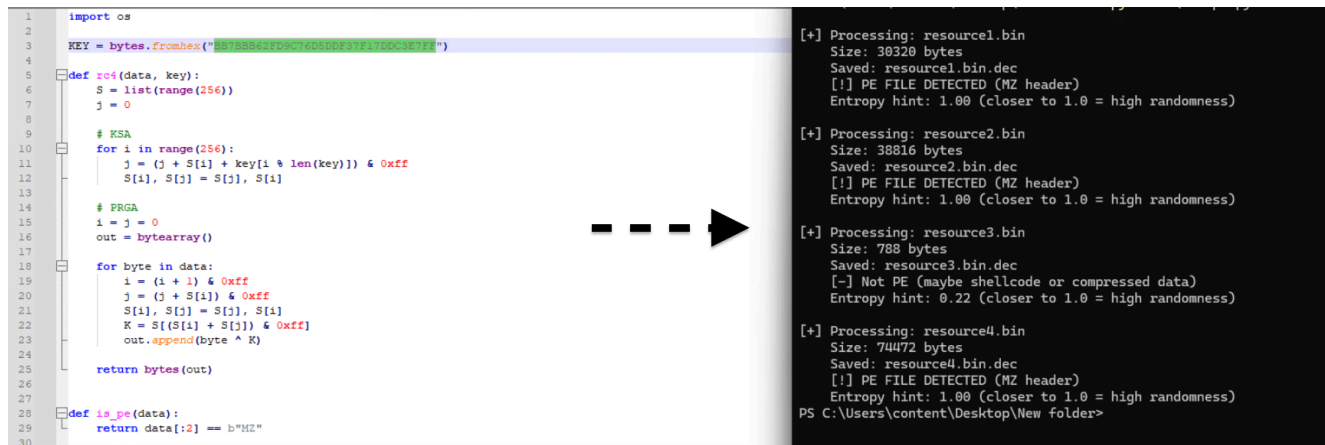


Figure 12. Validation of the recovered RC4 resource-decryption routine

With the resources decrypted, the framework can deploy one of three signed but vulnerable drivers and establish kernel-level communication. The next section examines how SilverFox uses *BootRepair.sys*, *EnPortv.sys*, and *wsftprm.sys* to terminate security processes and expand its BYOVD capabilities.

SilverFox's Modular BYOVD Framework

One of the most significant findings in this campaign is SilverFox's modular BYOVD framework. Rather than depending on a single signed but vulnerable driver, the malware supports three options: *BootRepair.sys*, *EnPortv.sys*, and *wsftprm.sys*.

After selecting a driver, the framework decrypts it using the RC4 routine described in the previous section, writes it to disk, installs it as a kernel-mode service through the Windows Service Control Manager (SCM), and opens the driver's device interface using *CreateFile()*. It then sends privileged requests through *DeviceIoControl()*.

The malware does not implement a completely separate workflow for every driver. Instead, it reuses common routines for:

- Resource extraction and decryption
- Writing the driver to disk
- Creating and starting the kernel service
- Opening the driver's device object
- Sending driver-specific IOCTL requests

The primary elements that change between implementations are the driver image, device name, and IOCTL value.

This reusable architecture gives SilverFox multiple paths to kernel-level process termination. If one driver is unavailable, incompatible, or blocked, another supported driver may provide the same operational capability.

BootRepair.sys: A New SilverFox Driver Association

The malware embeds *BootRepair.sys*, a legitimately signed but vulnerable driver previously demonstrated in public offensive-security research. The driver exposes functionality that can be abused to terminate processes from kernel mode.

Our analysis shows that SilverFox can deploy *BootRepair.sys* through the shared BYOVD framework before communicating with its device interface.

The driver's abuse potential was previously documented by the [PhantomKiller project](#). However, based on the public reporting we reviewed, this is the first documented association between *BootRepair.sys* and SilverFox activity.

EnPortv.sys: Kernel-Assisted Process Termination

The malware also embeds *EnPortv.sys*, a legitimately signed driver originally distributed with the EnCase forensic software suite. [Huntress previously documented](#) abuse of the driver in an unrelated BYOVD intrusion.

In the sample we analyzed, the malware opens the following device object:

`\\EnPortv`

Communication occurs through *DeviceIoControl()* using the observed IOCTL:

`0x223078`

Public analysis indicates that this request ultimately reaches *ZwTerminateProcess()*, allowing a caller to terminate processes from kernel mode, including protected endpoint-security processes.

Based on the public research we reviewed, this is also the first documented association between *EnPortv.sys* and SilverFox activity.

wsftprm.sys: A Known Driver in an Expanded Framework

The third embedded driver is *wsftprm.sys*, which exposes the following device object:

`\\Warsaw_PM`

The malware communicates with the driver using:

`0x22201C`

This IOCTL ultimately invokes *ZwTerminateProcess()*, enabling process termination from kernel mode.

The affected driver behavior is documented under [CVE-2023-52271](#). Unlike *BootRepair.sys* and *EnPortv.sys*, *wsftprm.sys* has previously appeared in public reporting on SilverFox-linked activity.

Why Three Drivers Matter

The inclusion of three drivers indicates that SilverFox is not relying on one fixed kernel-level exploitation path. By supporting multiple drivers through reusable logic, the malware broadens its operational options and may improve its ability to impair endpoint protection across different environments.

The design also makes the driver layer replaceable. The surrounding framework can retain the same broad workflow while changing the selected driver, device name, and IOCTL request.

The BYOVD implementation is only one part of a broader modular architecture. The same framework also incorporates NTDLL unhooking, runtime string construction, dynamic API resolution, process injection, internal payload monitoring, and an external watchdog.

Turning Kernel Access Against Security Products

After establishing communication with a vulnerable driver, the framework searches the compromised system for running security products.

Our analysis identified a hardcoded list of process names associated with endpoint detection and response, antivirus, and security-monitoring products. The user-mode component enumerates running processes and resolves the process identifier associated with each matching name.

Rather than attempting to terminate those processes directly from user mode, the framework sends the target PID to the loaded vulnerable driver. The driver then invokes `ZwTerminateProcess()` from kernel mode to terminate the selected process. This separation assigns two different responsibilities:

- The user-mode component identifies target security processes and coordinates driver communication.
- The vulnerable driver performs the kernel-mode process-termination.

This approach can bypass restrictions that may prevent ordinary user-mode processes from terminating protected security products.

Figure 13 shows the hardcoded list of security-related process names identified in the malware.

10192dbc	6e 00 74	unicode	u"ntrtsan"	XREF[1]:	101f60c4(*)
	00 72 00				
	74 00 73 ...				
10192dce	00	??	00h		
10192dcf	00	??	00h		
		u_UiWatchDog_10192dd0		XREF[1]:	101f60c8(*)
10192dd0	55 00 69	unicode	u"UiWatchDog"		
	00 57 00				
	61 00 74 ...				
10192de6	00	??	00h		
10192de7	00	??	00h		
		u_TrendMicroSvc_10192de8		XREF[1]:	101f60cc(*)
10192de8	54 00 72	unicode	u"TrendMicroSvc"		
	00 65 00				
	6e 00 64 ...				
		u_SentinelHelperService_10192e04		XREF[1]:	101f60d4(*)
10192e04	53 00 65	unicode	u"SentinelHelperService"		
	00 6e 00				
	74 00 69 ...				
		u_SentinelBrowserNativeHost_10192e30		XREF[1]:	101f60d8(*)
10192e30	53 00 65	unicode	u"SentinelBrowserNativeHost"		
	...				

Figure 13. Security processes targeted for kernel-level termination

This kernel-mode process termination supports defense evasion by reducing the visibility and effectiveness of endpoint-security controls during later stages of execution.

Restoring NTDLL to Reduce User-Mode Visibility

In addition to terminating security processes through vulnerable drivers, the malware uses a separate user-mode defense-evasion technique.

Before invoking native Windows APIs associated with later execution stages, the malware restores the in-memory *.text* section of *ntdll.dll* using a clean copy mapped from the Windows system directory.

Endpoint-security products may place inline hooks inside the in-memory executable code of system libraries such as *ntdll.dll* to monitor native API activity. By replacing the potentially modified *.text* section with the clean version from disk, the malware removes those user-mode inline hooks before calling native Windows APIs.

Figure 14 shows the malware loading a clean copy of *ntdll.dll* from disk as the first step in restoring the library's executable code.

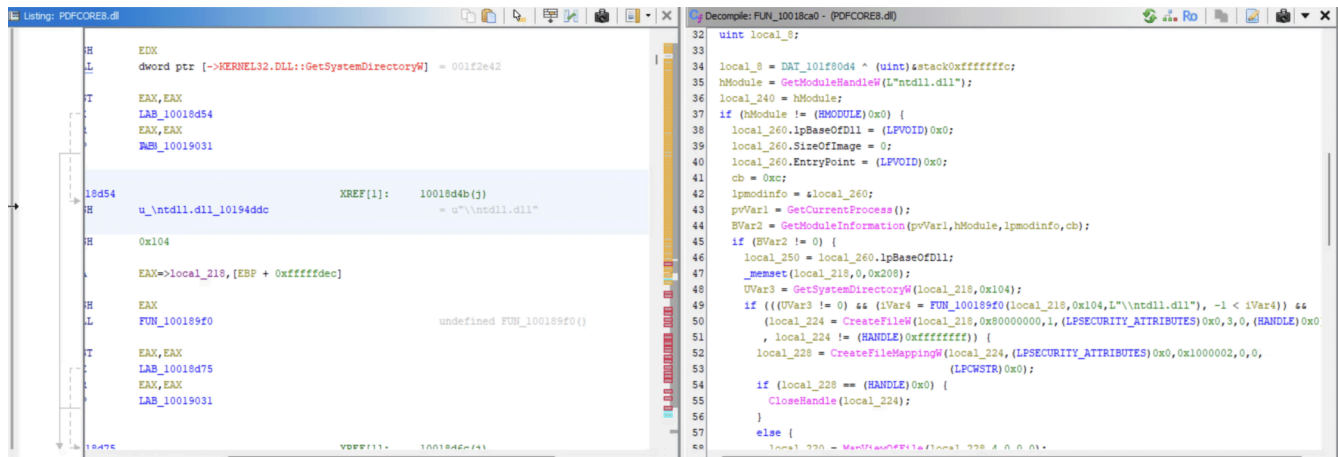


Figure 14. Clean *ntdll.dll* mapped from disk

Figure 15 shows the PE-parsing logic used to locate the executable .text section before replacing the potentially hooked in-memory code

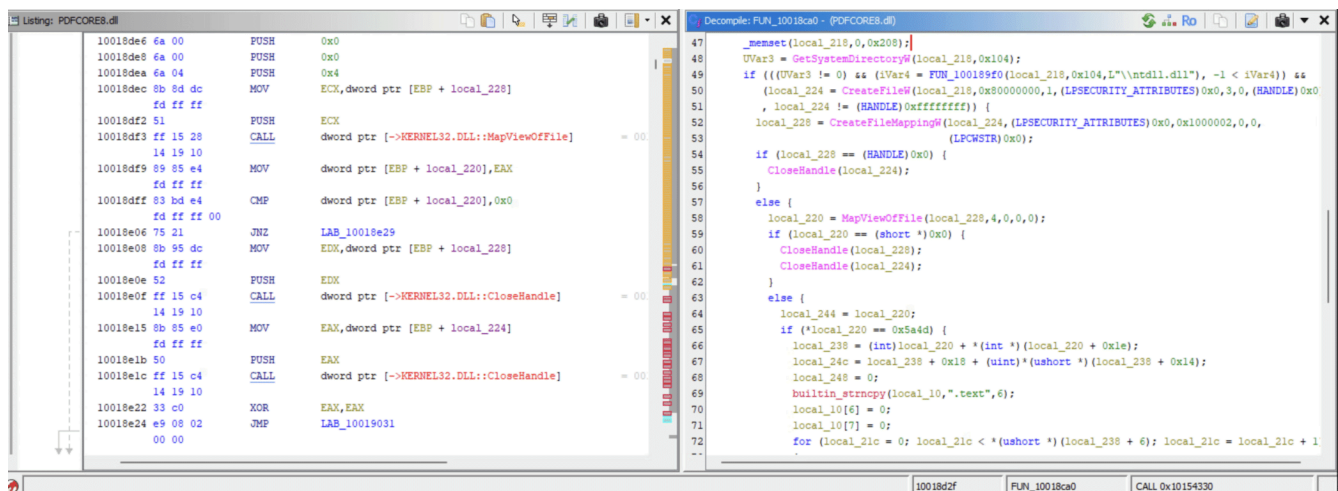


Figure 15. Identification of the *ntdll.dll* text section during unhooking

This technique does not eliminate all endpoint visibility. Kernel telemetry, behavioral analytics, network inspection, and other monitoring layers may still detect the activity. Its purpose is specifically to reduce visibility from user-mode inline hooks.

Reconstructing Strings at Runtime

The malware also avoids storing several sensitive library names as complete plaintext strings.

During initialization, the malware constructs names such as *kernel32.dll* dynamically by combining individual characters and short fragments immediately before use. Analysis of *FUN_10009DF0()* identified repeated calls to *IstrcpyW()* and *IstrcatW()* that progressively assemble the complete library name.

This technique reduces the number of meaningful indicators exposed through conventional static string extraction. The full library names exist only briefly in memory when the malware needs them.

Although runtime string construction is not sophisticated on its own, it supports the framework's broader effort to conceal its dependencies and capabilities during static analysis.

Figure 16 shows the runtime reconstruction of *kernel32.dll*.

```

10009dfe 33 c5      XOR     EAX,EAX
10009e00 89 45 fc   MOV     dword ptr [EBP + local_8],EAX
10009e03 68 ff 00   PUSH    000000ff
10009e08 6a 00     PUSH    0x0
10009e0a 8d 85 d8   LEA     EAX,>local_12c,[EBP + 0xfffffd8]
10009e10 50       PUSH    EAX
10009e11 e8 1a a5   CALL    _memset                                void * _memset(void * _Det,
14 00
10009e16 83 c4 0c   ADD     ESP,0xc
10009e19 68 00 08   PUSH    0x800
10009e1e 6a 00     PUSH    0x0
10009e20 8d 8d d8   LEA     ECX,>local_92c,[EBP + 0xfffffd8]
10009e26 51       PUSH    ECX
10009e27 e8 04 a5   CALL    _memset                                void * _memset(void * _Det,
14 00
10009e2c 83 c4 0c   ADD     ESP,0xc
10009e2f 68 ac 20   PUSH    DAT_101920ac                        = 4Bh  K
19 10
10009e34 8d 95 d8   LEA     EDI,>local_92c,[EBP + 0xfffffd8]
10009e3a 52       PUSH    EDI
10009e3b ff 15 6c   CALL    dword ptr [>KERNEL32.DLL:1:lstrocpyW] = 001f2bb6
14 19 10
10009e41 68 b0 20   PUSH    DAT_101920b0                        = 45h  E
19 10
10009e46 8d 85 d8   LEA     EAX,>local_92c,[EBP + 0xfffffd8]
10009e4c 50       PUSH    EAX
10009e4d ff 15 68   CALL    dword ptr [>KERNEL32.DLL:1:lstrocatW] = 001f2bc2
14 19 10
10009e53 68 b4 20   PUSH    DAT_101920b4                        = 52h  R
19 10
10009e58 8d 8d d8   LEA     ECX,>local_92c,[EBP + 0xfffffd8]
10009e5e 51       PUSH    ECX
10009e5f ff 15 68   CALL    dword ptr [>KERNEL32.DLL:1:lstrocatW] = 001f2bc2
14 19 10

16  undefined1 uStack_128;
17  CHAR local_2c [36];
18  uint local_8;
19
20  local_8 = DAT_101f80d4 * (uint)stack0xfffff8c;
21  _memset(local_12c,0,0xffff);
22  _memset(local_92c,0,0x800);
23  lstrocpyW(local_92c,L"K");
24  lstrocatW(local_92c,L"E");
25  lstrocatW(local_92c,L"R");
26  lstrocatW(local_92c,L"U");
27  lstrocatW(local_92c,L"E");
28  lstrocatW(local_92c,L"L");
29  lstrocatW(local_92c,L"3");
30  lstrocatW(local_92c,L"2");
31  lstrocatW(local_92c,L".");
32  lstrocatW(local_92c,L"d");
33  lstrocatW(local_92c,L"l");
34  lstrocatW(local_92c,L"l");
35  builtin_memcpy(local_2c,"GetSystemDirectoryA",0x14);
36  pCVar6 = local_2c;
37  pWVar1 = LoadLibraryW(local_92c);
38  pFVar2 = GetProcAddress(pWVar1,pCVar6);
39  (*pFVar2)(local_12c,0xffff);
40  uStack_128 = 0;
41  iVar3 = FUN_1000a500(local_12c,param_3);
42  if (iVar3 == 0) {
43      uVar4 = 0;
44  }
45  else {
46      builtin_memcpy(local_2c + 0x14,"VirtualAllocEx",0xf);
47      pCVar6 = local_2c + 0x14;
48      pWVar1 = LoadLibraryW(local_92c);
49      pFVar2 = GetProcAddress(pWVar1,pCVar6);
50      iVar3 = (*pFVar2)(*param_3,0,param_2,0x3000,0x40);
51      if ((iVar3 == 0) || (iVar5 = FUN_1000a1a0(*param_3,param_1,param_2), iVar5 == 0)) {
52          uVar4 = 0;
53      }
54      else {
55          uVar4 = FUN_10009b50(param_3,iVar3);
56      }
57  }

```

Figure 16. Runtime reconstruction of the *kernel32.dll* library name

Resolving Windows APIs Only When Needed

After constructing the required library names, the malware dynamically resolves the Windows APIs needed for each execution stage.

Rather than exposing sensitive functions through the Import Address Table, the malware uses *LoadLibraryW()* and *GetProcAddress()* to resolve APIs immediately before use.

The dynamically resolved APIs support activities including process creation, remote memory allocation, process injection, driver-service installation, and communication with vulnerable drivers.

Figure 17 shows the runtime API-resolution pattern using *LoadLibraryW()* and *GetProcAddress()*.

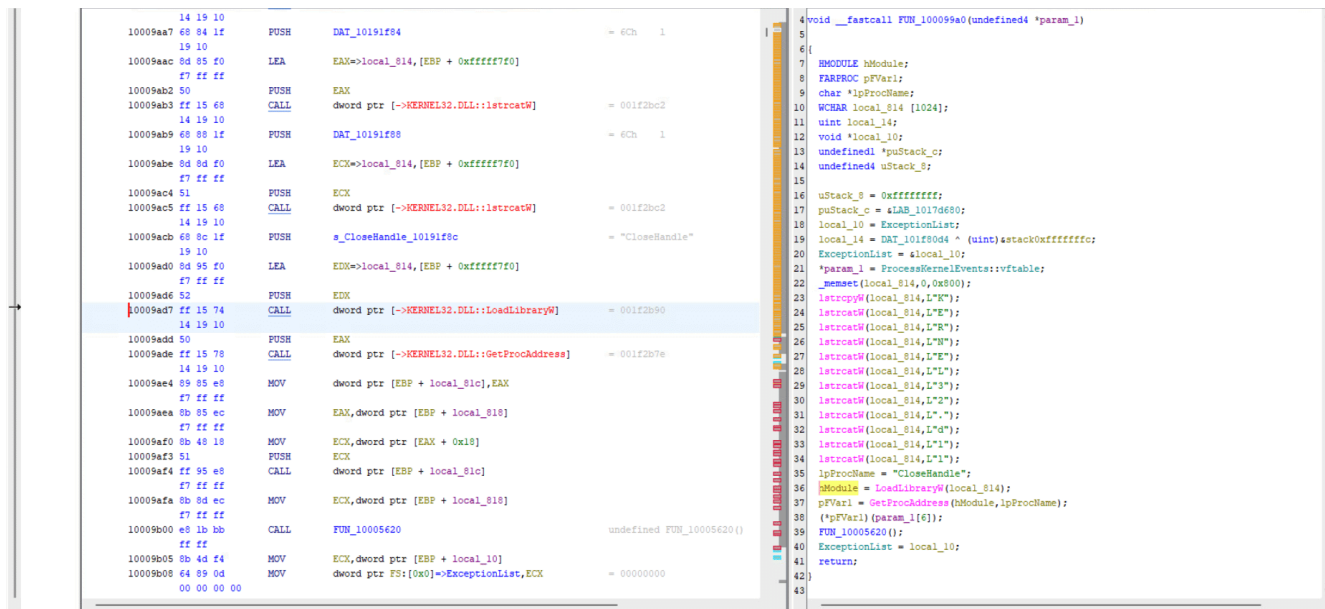


Figure 17. Dynamic resolution of Windows APIs during execution

With endpoint defenses impaired and user-mode monitoring reduced, the malware proceeds to retrieve shellcode and execute it inside a suspended svchost.exe process. The next section examines the thread-context hijacking workflow and the mechanisms SilverFox uses to restore execution when either the payload or loader is terminated.

Hijacking a Trusted Windows Process

After establishing communication with its command-and-control server at:

43[.]128[.]26[.]132

the malware downloads shellcode and executes it through thread-context hijacking.

The malware creates a new *svchost.exe* process in a suspended state using *CreateProcessA()* with the *CREATE_SUSPENDED* flag. It then allocates memory inside the suspended process with *VirtualAllocEx()*, writes the downloaded shellcode with *WriteProcessMemory()*, and retrieves the primary thread's execution context using *GetThreadContext()*.

The malware retrieves the suspended primary thread's processor context using *GetThreadContext()*, modifies the saved execution context so that control is transferred to the injected shellcode when the thread resumes, and commits the updated context with *SetThreadContext()*.

Finally, it invokes *ResumeThread()*, restoring the modified processor state and transferring execution to the injected payload.

The injection sequence is:

- Create a suspended *svchost.exe* process using *CreateProcessA()* with the *CREATE_SUSPENDED* flag.
- Allocate executable memory in the target process using *VirtualAllocEx()*.
- Copy the downloaded shellcode into the allocated region using *WriteProcessMemory()*.
- Retrieve the primary thread's processor context using *GetThreadContext()*.
- Modify the saved instruction pointer so that it points to the injected shellcode.
- Apply the modified processor context using *SetThreadContext()*.
- Resume the suspended primary thread using *ResumeThread()*, causing execution to continue from the injected shellcode.

Unlike techniques that create a new remote thread, thread-context hijacking reuses the target process's existing primary thread. This allows the malware to execute under the context of a trusted Windows process without introducing an additional execution thread.

Our reverse engineering identified three principal routines responsible for this workflow:

Routine	Responsibility
FUN_1000A500()	Creates the suspended <i>svchost.exe</i> process
FUN_1000A1A0()	Allocates remote memory and writes the shellcode
FUN_10009B50()	Redirects the primary thread and resumes execution

As shown in Figure 18, the malware creates a suspended Windows process, writes the shellcode into the target process's address space, modifies the suspended primary thread's execution context to transfer control to the injected payload, and then resumes the thread.

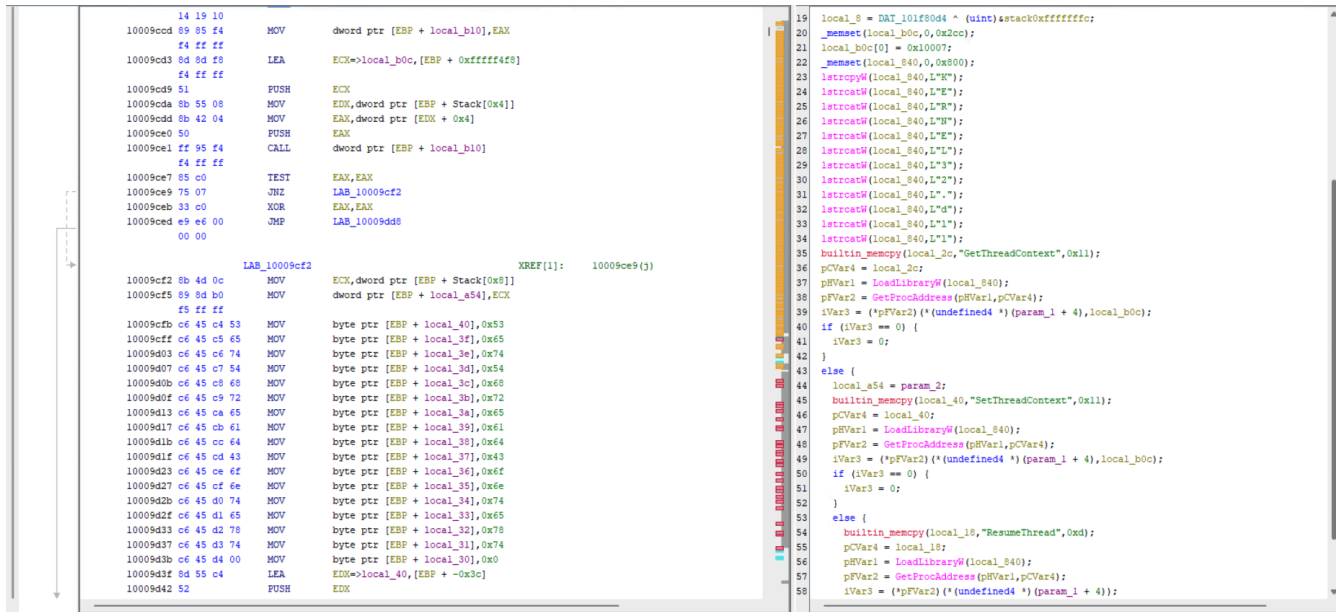


Figure 18. Thread-context hijacking of a suspended svchost.exe process

Persistence Through a Scheduled Task

To maintain execution across system reboots and user logons, the malware creates a Windows Scheduled Task that launches the second-stage loader automatically.

During execution, the malware invokes *schtasks.exe* to register a task that runs the dropped loader when a user logs on. Compared with persistence through Run keys or the Startup folder, a Scheduled Task can provide a more reliable way to restart the malware while blending with legitimate system administration activity.

Dynamic analysis confirmed that the task launches the second-stage executable, allowing the malware to re-establish the execution chain after a reboot or user logon.

Figure 19 shows the Scheduled Task created to launch the second-stage loader.

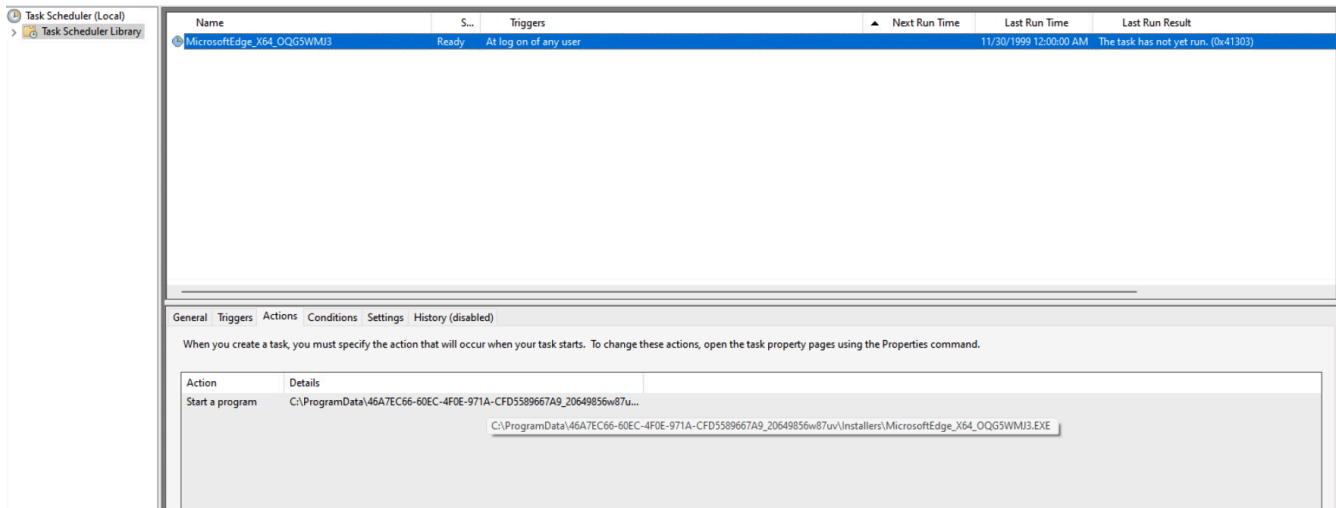


Figure 19. Scheduled Task used to relaunch the second-stage loader

Two Layers of Execution Recovery

SilverFox does not rely on a single mechanism to keep the malware running.

The malware combines internal payload supervision with an external watchdog script, creating two independent recovery layers.

The first layer monitors the injected payload. The second monitors the loader responsible for recreating that payload.

This design means that terminating only one component may not stop the intrusion:

- If the injected payload exits, the loader can recreate it.
- If the loader is terminated, the watchdog can relaunch it.

Internal Monitoring Recreates the Injected Payload

Our analysis identified *FUN_1000A350()*, a dedicated routine that checks whether the injected process remains active.

The routine reconstructs *kernel32.dll* at runtime and dynamically resolves:

- *OpenProcess()*
- *GetExitCodeProcess()*

It opens the injected process using *PROCESS_QUERY_INFORMATION* and retrieves the process exit code. If the returned value is *STILL_ACTIVE* (0x103, or decimal 259), the payload is considered operational.

The logic can be represented as:

```
hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);
```

```
if (!hProcess)  
    return FALSE;
```

```
GetExitCodeProcess(hProcess, &exitCode);
```

```
return exitCode == STILL_ACTIVE;
```

If the process is no longer active, execution returns to the injection workflow and the payload is created again.

This demonstrates that process injection is not performed only once. The loader continuously supervises the injected payload and can recreate it after interruption.

Figure 20 shows the routine used to determine whether the injected process remains active.

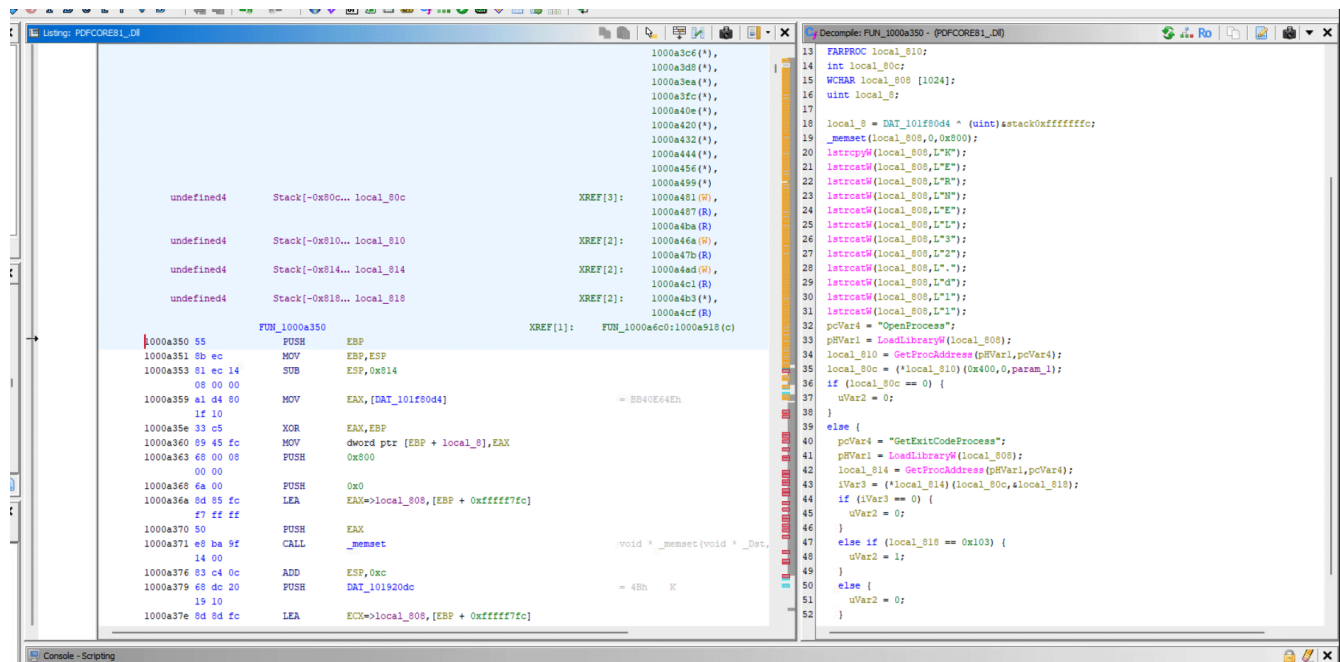


Figure 20. FUN_1000A350() checking whether the injected payload remains active

An External Watchdog Restarts the Loader

The second recovery layer is implemented through an embedded batch script extracted from the resources inside *PDFCORE8.dll*.

After launching the second-stage executable, the malware starts *cmd.exe* and executes the watchdog script. The script repeatedly uses native Windows utilities, including:

- *tasklist.exe*
- *find.exe*
- *timeout.exe*

Every 30 seconds, the script checks whether the monitored loader process is still running. If the process is no longer present, the script relaunches the executable and restores the second-stage execution chain.

Figure 21 shows the process tree created by the malware, including the loader, *cmd.exe*, and the embedded watchdog script.

Figure 21. Process tree showing the loader and watchdog execution chain

In Figure 22, we show the batch-script logic that checks the process state and restarts the loader after termination.

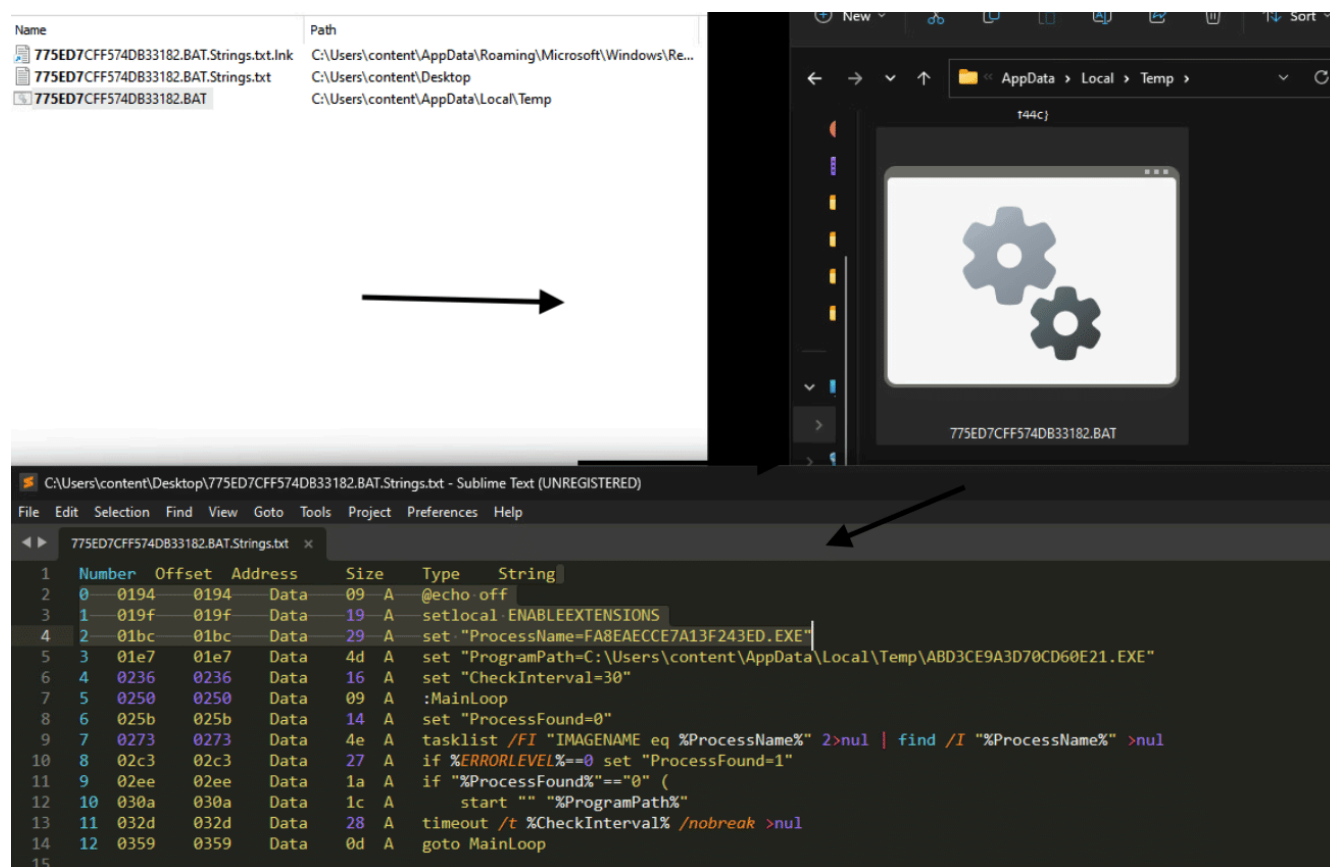


Figure 22. Watchdog script used to restore the SilverFox loader

Why the Dual-Watchdog Design Matters

Together, the two mechanisms protect different parts of the execution chain:

- The internal monitor protects the injected payload.
- The external watchdog protects the loader.

This layered design increases resilience because defenders must interrupt both components and prevent either from restoring the other stage.

The recovery architecture also reinforces the modularity observed throughout the sample. Driver deployment, security-process termination, injection, payload monitoring, and loader recovery are implemented as coordinated components rather than isolated techniques.

Registry-Resident Payload and Configuration

The malware stores the downloaded shellcode in the Windows registry as a REG_BINARY value under:

HKCU\Console\0

Rather than downloading the payload again each time it is needed, the malware reconstructs it directly from the registry. This reduces repeated network activity and avoids storing the payload as a separate file on disk.

Figure 23 shows the shellcode stored in the registry.

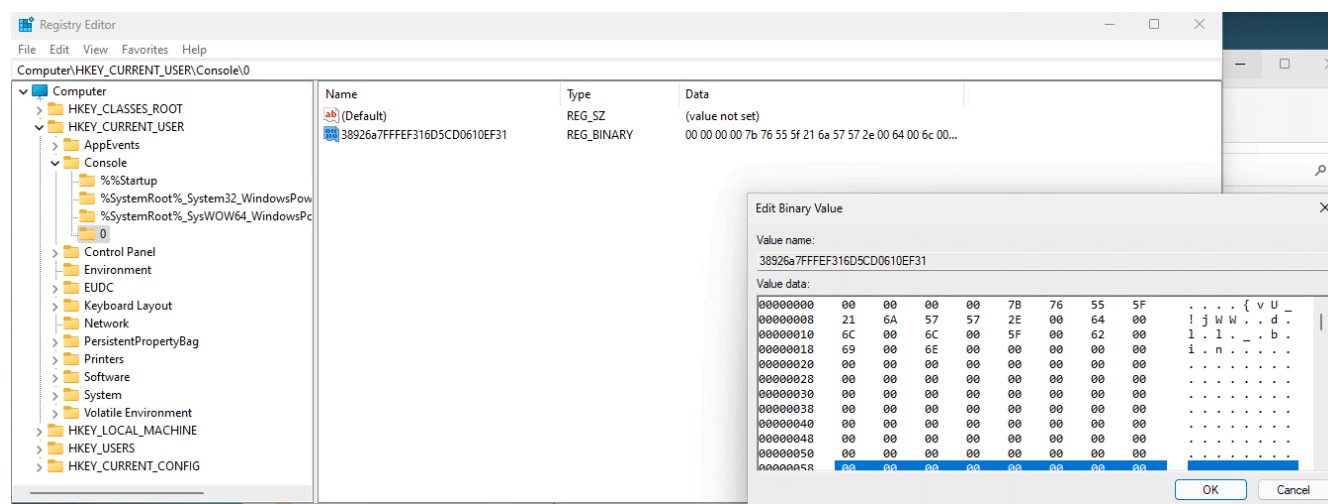


Figure 23. Final-stage shellcode stored under HKCU\Console\0

The malware stores its runtime configuration separately under:

HKLM\SOFTWARE\IpDates_sun

This REG_BINARY value contains command-and-control information, including:

- Server: 43[.]128[.]26[.]132
- Ports: 778 and 779

Separating the payload from its configuration allows the malware to modify communication settings without replacing the stored shellcode.

Figure 24 shows the configuration data stored in the registry.

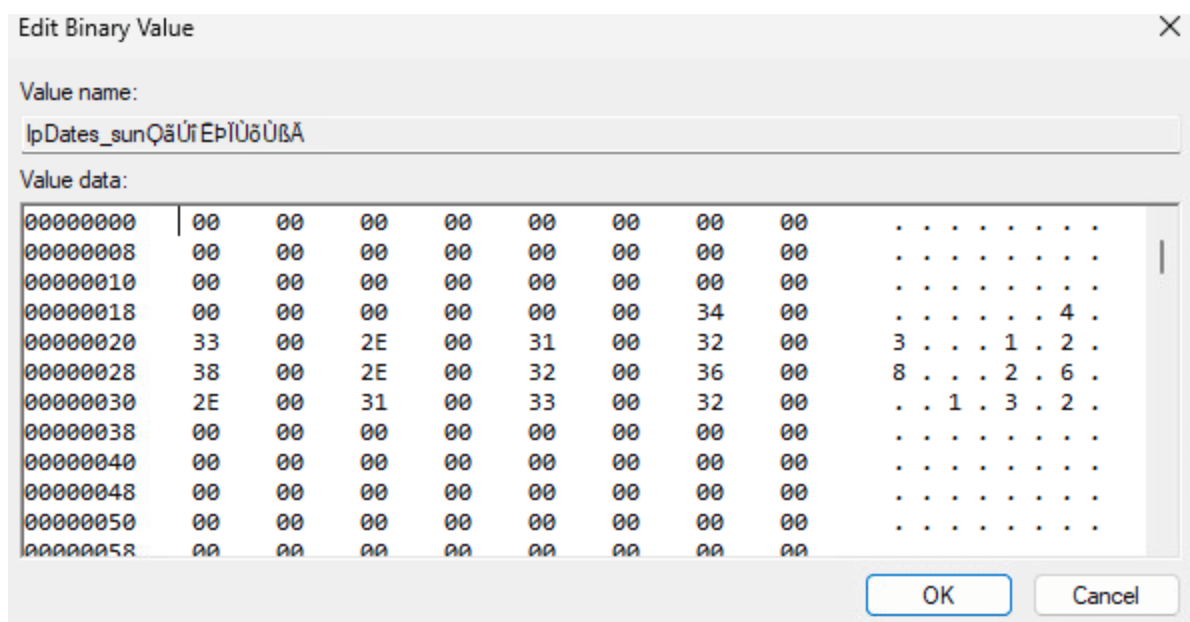


Figure 24. ValleyRAT command-and-control configuration stored in the registry

Patching the Payload Before Execution

Before executing the registry-resident shellcode, the malware searches it for the marker:

FaCai2024

The marker does not represent executable instructions. Instead, it identifies a reserved region inside the payload where the malware inserts its runtime configuration.

The framework retrieves the configuration from:

HKLM\SOFTWARE\IpDates_sun

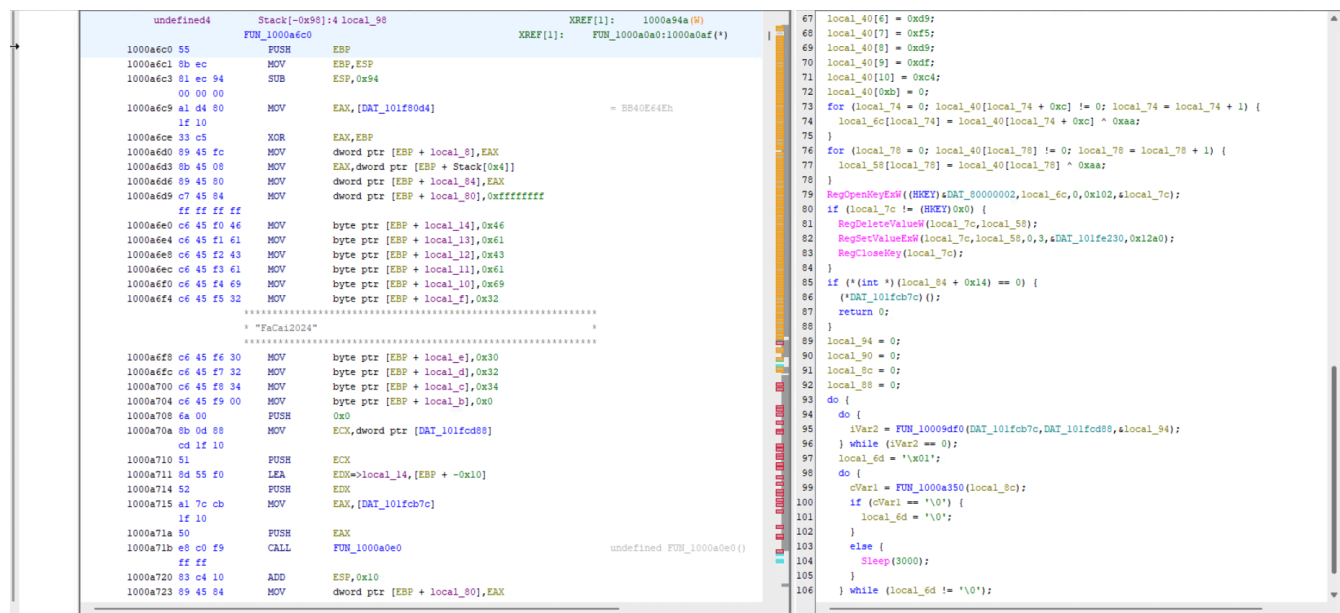
It then replaces the FaCai2024 placeholder with the recovered C2 information, producing a fully configured payload immediately before execution.

This separation gives the attacker operational flexibility. The same stored payload can be reused with updated server addresses or ports without changing the payload body itself.

The phrase “Fa Cai” (发财) is Mandarin Chinese for “become prosperous” or “make a fortune” and commonly appears in Lunar New Year greetings.

Earlier [Tencent Security research on the FaCai phishing group](#) documented the same FaCai2024 marker and related use of HKEY_CURRENT_USER\Console\0 to store binary payload data before in-memory execution. A [subsequent Tencent Security report](#) described continued use of the same registry location and memory-execution pattern in later FaCai activity. These overlaps provide additional behavioral context for the malware analyzed in our campaign.

Figure 25 shows the FaCai2024 marker and the configuration region patched before payload execution.



```
1000a6c0 55      PUSH     EBP
1000a6c1 8b ec    MOV     ESP, ESP
1000a6c3 81 ec 94 SUB     ESP, 0x94
00 00 00
1000a6c9 a1 d4 80 MOV     EAX, [DAT_101f80d4]
if 10
1000a6ce 33 c5    XOR     EAX, EBP
1000a6d0 89 45 fc MOV     dword ptr [EBP + local_8], EAX
1000a6d3 8b 45 08 MOV     EAX, dword ptr [EBP + Stack[0x4]]
1000a6d6 89 45 80 MOV     dword ptr [EBP + local_84], EAX
1000a6d9 c7 45 84 MOV     dword ptr [EBP + local_80], 0xffffffff
ff ff ff ff
1000a6e0 c6 45 f0 46 MOV     byte ptr [EBP + local_14], 0x46
1000a6e4 c6 45 f1 61 MOV     byte ptr [EBP + local_13], 0x61
1000a6e8 c6 45 f2 43 MOV     byte ptr [EBP + local_12], 0x43
1000a6ec c6 45 f3 61 MOV     byte ptr [EBP + local_11], 0x61
1000a6f0 c6 45 f4 69 MOV     byte ptr [EBP + local_10], 0x69
1000a6f4 c6 45 f5 32 MOV     byte ptr [EBP + local_f], 0x32
*****
+ "FaCai2024"
*****
1000a6f8 c6 45 f6 30 MOV     byte ptr [EBP + local_e], 0x30
1000a6fc c6 45 f7 32 MOV     byte ptr [EBP + local_d], 0x32
1000a700 c6 45 f8 34 MOV     byte ptr [EBP + local_c], 0x34
1000a704 c6 45 f9 00 MOV     byte ptr [EBP + local_b], 0x0
1000a708 6a 00   PUSH     0x0
1000a70a 8b 0d 88 MOV     ECX, dword ptr [DAT_101fcd88]
cd if 10
1000a710 51      PUSH     ECX
1000a711 8d 55 f0 LEA     EDI, local_14, [EBP + -0x10]
1000a714 52      PUSH     EDI
1000a715 a1 7c cb MOV     EAX, [DAT_101fcb7c]
if 10
1000a71a 50      PUSH     EAX
1000a71b e8 c0 f9 CALL     FUN_1000a0e0
ff ff
1000a720 83 c4 10 ADD     ESP, 0x10
1000a723 89 45 84 MOV     dword ptr [EBP + local_80], EAX
```

```
67 local_40[6] = 0x0;
68 local_40[7] = 0x0;
69 local_40[8] = 0x0;
70 local_40[9] = 0x0;
71 local_40[10] = 0x0;
72 local_40[0xb] = 0;
73 for (local_74 = 0; local_40[local_74 + 0xc] != 0; local_74 = local_74 + 1) {
74     local_6c[local_74] = local_40[local_74 + 0xc] ^ 0xaa;
75 }
76 for (local_78 = 0; local_40[local_78] != 0; local_78 = local_78 + 1) {
77     local_58[local_78] = local_40[local_78] ^ 0xaa;
78 }
79 RegOpenKeyEx((HKEY) &DAT_80000002, local_6c, 0, 0x102, &local_7c);
80 if (local_7c != (HKEY) 0x0) {
81     RegDeleteValue(local_7c, local_58);
82     RegSetValueEx(local_7c, local_58, 0, 3, &DAT_101fe230, 0x12a0);
83     RegCloseKey(local_7c);
84 }
85 if ((*int *) (local_84 + 0x14) == 0) {
86     (*DAT_101fcb7c)();
87     return 0;
88 }
89 local_84 = 0;
90 local_80 = 0;
91 local_8c = 0;
92 local_88 = 0;
93 do {
94     do {
95         iVar2 = FUN_10009df0(DAT_101fcb7c, DAT_101fcd88, &local_84);
96     } while (iVar2 == 0);
97     local_6d = '\x01';
98     do {
99         cVar1 = FUN_1000a350(local_8c);
100         if (cVar1 == '\0') {
101             local_6d = '\0';
102         }
103         else {
104             Sleep(3000);
105         }
106     } while (local_6d != '\0');
```

Figure 25. FaCai2024 marker used to patch configuration into the payload

ValleyRAT Establishes Remote Access

After patching is complete, the framework executes the reconstructed shellcode through the same thread-context hijacking workflow described earlier.

The resulting final-stage implant provides remote-access functionality, including command-and-control communication, task execution, and additional post-compromise capabilities.

ValleyRAT, also known as Winos 4.0, is the remote-access malware delivered in the final stage of the campaign. The campaign therefore progresses through a coordinated sequence:

1. Invoice-themed phishing and cloud-hosted delivery
2. DLL sideloading through trusted applications
3. Decryption and deployment of vulnerable drivers
4. Kernel-assisted termination of security products
5. NTDLL unhooking and dynamic API resolution
6. Shellcode injection into suspended svchost.exe
7. Internal and external recovery mechanisms
8. Registry-based payload reconstruction and configuration patching
9. ValleyRAT execution and remote access

Detection Opportunities

Defenders can detect this campaign by looking for combinations of file, process, driver, registry, and network behaviors across the attack chain. Individual indicators may appear benign in isolation, but together they can reveal the broader SilverFox execution framework.

Key detection opportunities include:

- Execution of ConvertToPDF.exe, PDFDirect.exe, or renamed copies from %TEMP% or other user-writable directories.
- Loading of PDFCORE8.dll from the same directory as the trusted executable.
- Creation and startup of kernel-driver services associated with BootRepair.sys, EnPortv.sys, or wsftprm.sys.
- Creation of a suspended svchost.exe process followed by remote memory allocation, memory writes, thread-context modification, and thread resumption.
- Writes of binary data to HKCU\Console\0 and configuration data to HKLM\SOFTWARE\lpDates_sun.
- Repeated execution of tasklist.exe, find.exe, and timeout.exe from cmd.exe, consistent with the external watchdog script.
- Network connections to attacker-controlled content hosted through QQ or Tencent Cloud infrastructure, especially when followed by local DLL sideloading or driver-service creation.

Detection logic should prioritize correlated behavior rather than relying only on a single filename, digital signature, domain, or driver hash. In this campaign, trusted applications and

legitimate cloud services are used as part of the attack chain, making sequence-based detection especially important.

How Cato Blocks the SilverFox Attack Chain

SilverFox gains operational advantage by combining trusted software, legitimate cloud-hosted services, vulnerable drivers, and in-memory execution. Each individual component may appear less suspicious in isolation, but the complete sequence reveals a coordinated malware-delivery and defense-evasion chain.

The Cato SASE Platform blocks the campaign before ValleyRAT can establish remote access. In the observed incident, Cato’s Next Generation Anti-Malware (NGAM) capability identified and prevented the malicious PDFCORE8.dll before the later driver-deployment, process-injection, and payload-execution stages could complete.

This prevention is important because the malicious DLL is the central execution component. It contains the encrypted drivers and watchdog script, coordinates BYOVD deployment, performs defense evasion, retrieves the final shellcode, and manages the recovery mechanisms described earlier.

In addition to automated prevention, [Cato Managed Detection and Response](#) provides continuous monitoring, investigation, and expert-led response to help organizations identify and contain suspicious activity across the attack chain.

As shown in Figure 26, Cato NGAM detected and blocked the malicious PDFCORE8.dll, interrupting the attack before the embedded framework could continue.

Target Actions				
Target	Type	Action	Related Events	
myqcloud.com	Anti Malware	Block	Gen:Variant.Tedy.967777	
43.132.105.206	Anti Malware	Block	Gen:Variant.Tedy.967777	
hrefbfdhfhgre-1422102728.cos.ap-hongkong.myqcloud.com	Anti Malware	Block	Gen:Variant.Tedy.967777	

Figure 26. Cato NGAM blocks the malicious PDFCORE8.dll

Cato XOps then correlates the relevant events into a single investigation story, helping security teams understand how the malicious file, network activity, and affected endpoint relate to the broader campaign.

Figure 27 shows the Cato XOps Story view, which brings the related security events together for investigation.

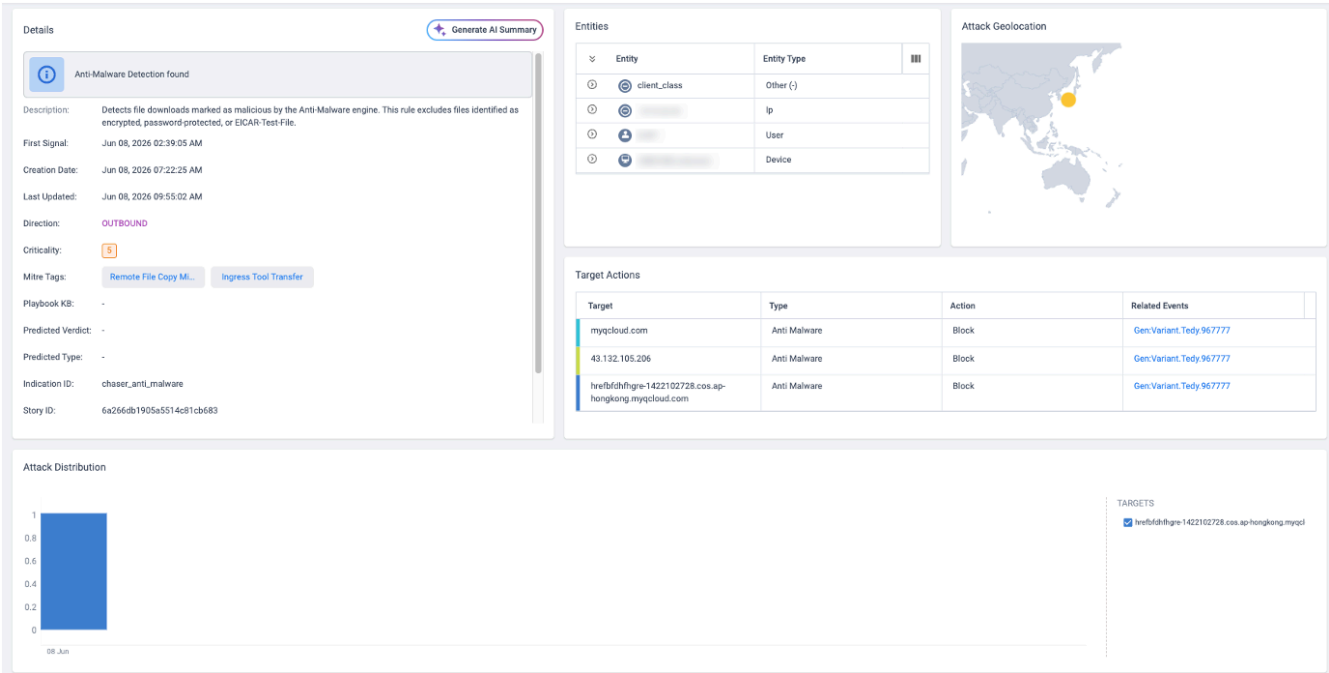


Figure 27. Cato XOps correlates the SilverFox campaign activity

This layered visibility is particularly important in campaigns that abuse legitimate services and signed applications. Blocking based only on a domain name, executable signature, or isolated process may be insufficient when the threat actor deliberately combines trusted infrastructure with malicious content. Correlating file, network, and behavioral evidence makes it possible to identify the broader attack sequence rather than evaluate each artifact separately.

Coordinated Disclosure and Infrastructure Abuse Reporting

During our investigation, we notified the relevant organizations about the abuse of their software or infrastructure. These reports covered the DLL-sideload behavior involving ConvertToPDF.exe and PDFDirect.exe, the malicious archive hosted through QQ file-delivery infrastructure, and the attacker-controlled objects hosted through Tencent Cloud.

In each case, the threat actor abused a legitimate product or service as part of the delivery or execution chain.

Date	Organization / Service	Reported Activity	Status
July 1, 2026	ZEON / Right PDF	Abuse of ConvertToPDF.exe and PDFDirect.exe for DLL sideloading	Reported under ticket 13493; no response received as of July 20, 2026.
July 12, 2026	Tencent	Abuse of QQ file-delivery infrastructure and Tencent Cloud object storage to host and deliver malicious campaign content	Reported under ticket 69528; no response received as of July 20, 2026.

Research Scope

Our findings are based on the samples, telemetry, and infrastructure available during the investigation. We observed the campaign targeting a Japanese organization in the industrial manufacturing sector, but the available evidence does not establish the campaign's complete victim scope. Technical conclusions in this report are limited to the behavior observed in the analyzed samples.

Conclusion

This campaign shows how SilverFox is expanding its defense-evasion and remote-access toolkit. The most significant additions are the previously undocumented SilverFox use of *BootRepair.sys* and *EnPortv.sys*, along with the abuse of *ConvertToPDF.exe* and *PDFDirect.exe* as DLL-sideloading hosts.

The malicious PDFCORE8.dll brings these capabilities together in one modular stage that combines vulnerable-driver deployment, security-process termination, NTDLL unhooking, thread-context hijacking, registry-based payload storage, and dual recovery mechanisms. This design improves resilience by allowing the malware to restore execution if either the payload or loader is interrupted.

For defenders, the campaign highlights the need to evaluate trusted applications and legitimate cloud services in context. Detection should focus on the full activity chain, including local DLL loading, kernel-driver service creation, suspended-process manipulation, registry-resident payloads, and repeated process recovery.

Indicators of Compromise

Network Infrastructure

Indicator	Service	Role
file[.]wx2[.]qq[.]com	QQ	Hosted or redirected users to the invoice-themed archive
hrefbfdhfhgre-1422102728[.]cos[.]ap-hongkong[.]myqcloud[.]com	Tencent Cloud	Hosted the malicious PDFCORE8.dll
43[.]128[.]26[.]132	—	Hosted the downloaded shellcode or next-stage payload

Registry Artifacts

Registry Path or Value	Purpose
HKCU\Console\0	Stores the downloaded shellcode as REG_BINARY data
HKLM\SOFTWARE\lpDates_sun	Stores C2 configuration
FaCai2024	Marker used to locate and patch the configuration region

Malicious File Hashes

Artifact	SHA-256
PDFCORE8.dll	01ccc662c4b5d6ff79ede2e5dd7e508c7e2029fcc13e988e22345b0e5413ed01
PDFCORE8.dll	0952f2aacbeeb3477b2465dfe3bac85d7906e958f866478cdbffec9be8cdb01c9
PDFCORE8.dll	0a0353fddca1ba1432b35411a8ce2536c9b35aad25d0819296748e13423c813b

Artifact	SHA-256
PDFCORE8.dll	18db6fbabcf73332eb9bb392f7ae0cc3381f3528983587e17b7ceba8dc37e9f6
PDFCORE8.dll	25c5ef94956a60ea6bb764f8abc82898906662207b59506b9647fd1c57bb b66b
PDFCORE8.dll	2a5c7af5106a2cbd6b0f5a96a3716b29271f0b476a3c21a7c9b59fdc7d5e 561b
PDFCORE8.dll	2b8a15c5d7e23a69b8052e1ec298c075324e8fa95ea06d191124326b848c b0ff
Downloader	0ef3119504f8aa3a534bfdbe20ed88727deabdf4e8f6e3e31298a34eef76 8662
Downloader	1355d2d73a1f21b4fa4e1b97432015052c004a0ca20a0b72d47011940fd5 1c6a
Downloader	31a276aae8a45873beecf4137bb27cd5a6dedde882a7b1e79f70ad0abeaa 3ee1
Downloader	36eb85e475954e2815c40e0c1546895de5419aa704cadf01d9e096a70ee8 4517
Batch watchdog script	5e9c2eb32e1c5e0e9f69c8eda4e5ff2ec9e46d43e75ce098b830016ae17f 291a

Vulnerable Drivers Observed in the Campaign

Filename	SHA-256
BootRepair.sys	c0ec1488902568be9a4960c670dfa5edac0e096d106d782fa6bc26ee56793b75
EnPortv.sys	1f0f08699f74eb2fb690edd064cba4f2fad06761f67d0bbb585e5cd03c77ea17
wsftprm.sys	381dbc9012d02b3a42916813b99e481b1cc89d15da918b58db40ce330cc13fca

These hashes identify vulnerable driver samples observed in the campaign. The drivers are not inherently malicious, but their presence should be evaluated together with service creation, driver loading, and attempts to impair security processes.

Associated Legitimate Applications

The campaign abused the legitimate ConvertToPDF.exe and PDFDirect.exe applications to load the malicious PDFCORE8.dll through DLL sideloading. The applications themselves are not malicious and should not be blocked based on filename alone. Detection should focus on the surrounding execution context, including an unexpected PDFCORE8.dll in the same directory and execution from an unusual path.

MITRE ATT&CK Mapping

Tactic	Technique	ATT&CK ID
Initial Access	Phishing	T1566
Command and Control	Application Layer Protocol: Web Protocols	T1071.001
Execution	DLL Side-Loading	T1574.002
Privilege Escalation	Exploitation for Privilege Escalation	T1068
Defense Evasion	Process Injection	T1055

Tactic	Technique	ATT&CK ID
Defense Evasion	Thread Execution Hijacking	T1055.003
Defense Evasion	Impair Defenses	T1562
Defense Evasion	Disable or Modify Security Tools	T1562.001
Defense Evasion	Obfuscated Files or Information	T1027
Persistence	Boot or Logon Initialization Scripts	T1037
Persistence	Scheduled Task/Job: Scheduled Task	T1053.005

Related Topics

[Cato CTRL](#)