

Targeted Attack on Middle East Govts (Part 1)

zscaler.com/blogs/security-research/targeted-attack-government-entities-middle-east-part-1

Sudeep Singh

July 20, 2026



Technical Analysis

In the following sections, ThreatLabz provides a technical analysis of the campaign, detailing the malware tooling it leverages and the post-compromise activity.

The attack chain begins with an ISO file containing a legitimate *RegSchdTask.exe* file from ASUSTek that sideloads a malicious DLL named *AsTaskSched.dll*. The figure below illustrates the campaign's attack flow used to distribute TELESIM, MIXEDKEY, and BINDCLOAK.

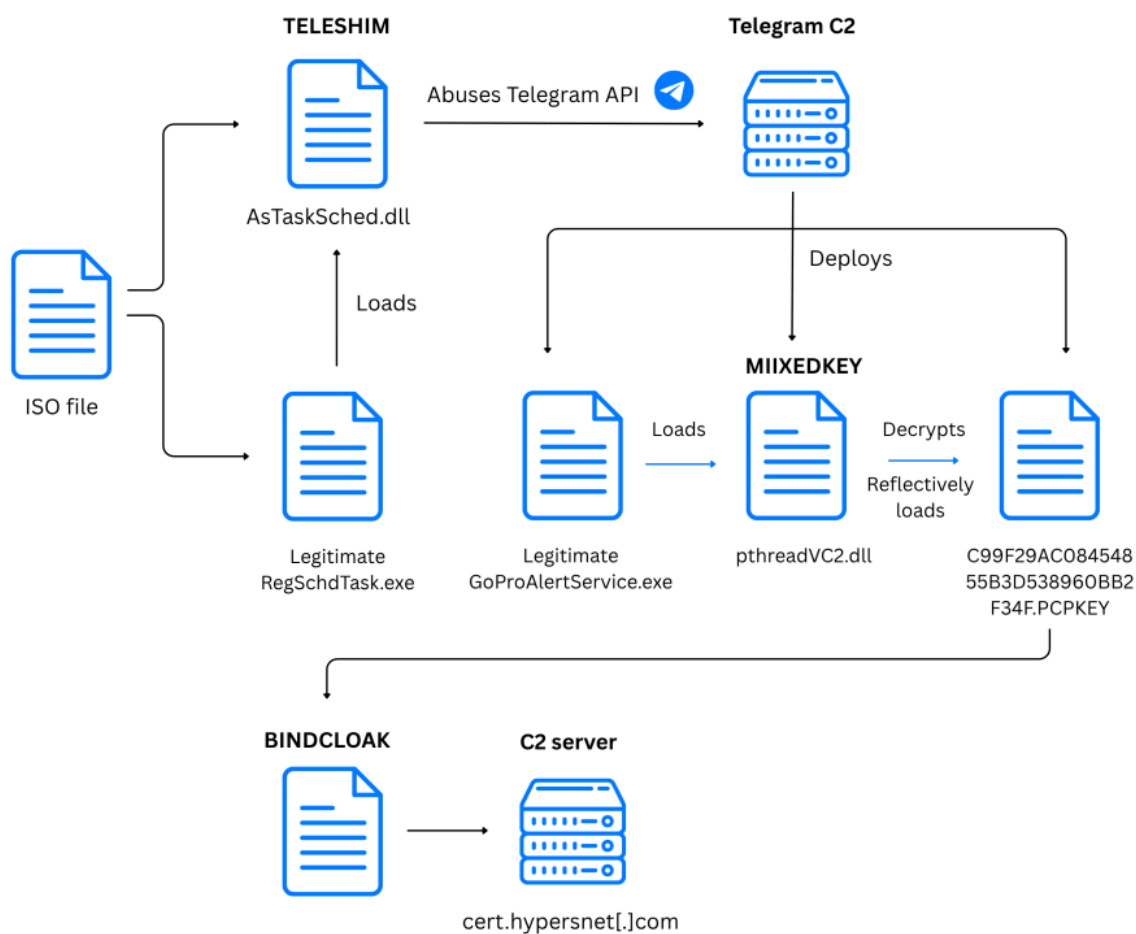


Figure 1: Multi-stage attack chain leading to the deployment of BINDCLOAK.

TELESHIM backdoor

TELESHIM is a 32-bit C++ Windows DLL that is used in the first stage of the attack. ThreatLabz identified three unique instances of *TELESHIM*. While two of these instances were compiled in 2025, the variant used in this campaign was compiled in July 2026. This variant introduced heavy code obfuscation techniques as well as encrypted strings.

Hook installer (indirect execution)

When the `DllMain` is invoked, the code locates the base address of the host executable and installs the following 7-byte hook at offset `0x1394` as shown below.

```
B8 <address>    MOV EAX, <main_payload_function>    ; 5 bytes
FF D0          CALL EAX                          ; 2 bytes
```



Figure 2: 7-byte hook at offset `0x1394`.

Before installing the hook, `DllMain` calls `VirtualProtect` to mark this memory region as writable and restores the original protection after writing the hook. The offset `0x1394` is hardcoded, indicating that the attacker reverse-engineered the legitimate host application to identify an instruction executed early on during its normal initialization. When the host executable reaches that point in its startup, the trampoline fires and execution enters the implant's main payload. By redirecting execution via an installed hook, the code is executed in the context of the host executable to evade security tools.

Mutex check

TELESHIM checks for the presence of a mutex named ----
WebKitFormBoundary7MA4YWxkTrZu0g on the victim's machine to ensure only a single instance of the malware is running at any given time.

String decryption

All strings relevant to the malicious activity are stored encrypted in the binary. Below are two string decryption variants.

Method 1

In this variant, each string has its own dedicated CFF-obfuscated decryption function. The XOR key and ciphertext are stored contiguously in a single `.rdata` blob. Every function is structurally identical; only the decryption key length and plaintext length operands change.

```
decryption_key = blob[:key_len]
ciphertext    = blob[key_len:]
```

```
plaintext = bytearray(plaintext_len)
for i in range(plaintext_len):
    plaintext[i] = decryption_key[i % key_len] ^ ciphertext[i]
```

Method 2

In this variant, each string is first Base64-decoded and then decrypted using the following 44-byte rolling XOR key.

```
8F 38 0C DA 29 6F 34 DE 27 69 7A 1A 53 05 18 49 B6 9D 59 E5 28 D7 E6
69 F1 7C F8 D3 CF 22 0B 66 96 DA 77 65 34 40 1C 8A 0F 0C 31 C6
```

This string decryption algorithm is also used to protect the network communication, which is discussed in more detail later.

Anti-analysis techniques

TELESHIM uses multiple anti-analysis techniques to evade automated analysis environments and hinder static reverse engineering.

I/O file stress test

TELESHIM delays execution by performing intense I/O file activity using a function that writes and reads back ~1 MB of randomly generated data to %TEMP%\CVR9EEA.tmp in a loop of 1000 iterations, generating roughly 1 GB of total disk I/O per invocation. This is likely to stall the execution in emulated, virtual, and analysis environments.

Hypervisor detection using CPUID

The next anti-analysis check calls the CPUID instruction with EAX=1 and checks bit 31 of the ECX register to detect the presence of a hypervisor. Since all major hypervisors set this bit, it can be used to detect virtualization-based analysis environments. If bit 31 of ECX is set, execution terminates.

RAM speed check using WMI

TELESHIM leverages Windows Management Instrumentation (WMI) to execute the following WMI query to check RAM speed.

```
wmic memorychip get speed
```

In a virtualization environment, this command usually returns either 0 or an undefined value. If the return value is 0 or cannot be parsed, then the execution terminates. The WMI query string itself is stored encrypted using string decryption method 2.

CFF and MBA usage

TELESHIM uses CFF and complex MBA expressions to deter reverse engineering. The figure below shows the CPUID-based anti-analysis technique leveraging MBA expressions.

```
1// CPUID EAX=1 -> ECX feature flags checked via MBA obfuscation (anti-hypervisor)
2bool sub_100847E0()
3{
4    unsigned int v5; // eax
5    unsigned int v6; // ebx
6    int v7; // edi
7    unsigned int v8; // edx
8    int v9; // ebp
9    unsigned int v10; // esi
10    unsigned int v11; // ecx
11    unsigned int v12; // eax
12    unsigned int v13; // eax
13    int v14; // ebx
14    int v15; // edx
15    int v16; // edi
16    unsigned int v17; // esi
17    int v18; // ebp
18    int v19; // ebx
19    int v20; // esi
20
21    _EAX = 1;
22    __asm { cpuid }
23    v5 = (_ECX ^ 0xE747B642) + (_ECX ^ 0x18B849BD) - (_ECX | 0xE747B642);
24    v6 = ~(_ECX & 0xE747B642);
25    v7 = ~_ECX + 2 * ~_ECX;
26    v8 = (v6 & 0x9CD06B7E) + (_ECX & 0x84402242) + (_ECX & 0xE747B642 ^ 0x632F9481) - (v6 | 0x9CD06B7E);
27    v9 = ~v5;
28    v10 = v5 & _ECX & 0xE747B642;
29    v11 = (v6 & 0x632F9481) - (v6 & 0x9CD06B7E) + (v6 | 0x632F9481) - 1664062593;
30    v12 = (~((v5 + (v9 | 0x9CD06B7E) + 1) | v9 & 0x632F9481)
31        + ((v5 + (v9 | 0x9CD06B7E) + 1) ^ v9 & 0x632F9481)
32        + v5
33        + (v9 | 0x9CD06B7E)
34        + 1
35        - (v5 | 0x9CD06B7E)) ^ ((v8 & v11) + ~v11 - v8 + 2 * (v8 | v11) + 1);
36    v13 = (~(v12 | v10) + ((v12 | v10) ^ 0xE747B662) - ~(v12 | v10) | 0x18B8499D) | ((v12 & 0x18B8499D)
37        + (v12 | 0xE747B662)
38        + (v12 & 0x18B8499D ^ 0x18B8499D)
39        + 1);
40    v14 = ((v7 + 1) & (v13 + 2 * ~v13 + 1))
41        + ((v13 + 2 * ~v13 + 1) ^ (v7 + 1))
42        + v7
43        + v13
44        + 2 * ~v13
45        + 1
46        + 2 * ~(v7 + 1) | (v13 + 2 * ~v13 + 1));
47    v15 = v14 + 3;
48    v16 = ~v7 - 2;
49    v17 = v13 + (v16 & v13);
50    v18 = ~4 - v14;
51    v19 = (v16 | v13) + (v13 | ~v16);
52    v20 = (v15 | v17 + ~v19)
53        + ~(((v17 + ~v19) & v18) + (v15 | (v19 - v17)))
54        + 2 * (((v17 + ~v19) & v18) + (v15 | (v19 - v17))) - (v15 | (v17 + ~v19));
55    return v20 + ~(2 * ((v20 + 1) | ~v13)) != ((v20 + 1) | v13);
56}
```



Figure 3: MBA expressions used to obfuscate the CPUID hypervisor bit check in TELESHIM.

Similar MBA expressions are used in other critical sections of the code, such as string decryption.

Staging payloads

TELESHIM creates a directory at C:\programdata\shimngen_Data\ for staging payloads. The legitimate executable (*RegSchdTask.exe*) is copied to this path as *shimngen.exe* and the malicious DLL is copied to this path as *AsTaskSched.dll*.

Persistence via scheduled tasks

TELESHIM creates a scheduled task named *shimgen* that runs every 6 minutes and executes the binary from `C:\programdata\shimgen_Data\shimgen.exe`.

C2 communication

TELESHIM abuses the Telegram API for C2 communication, a technique used to blend in with legitimate internet traffic. The hostname `api.telegram.org`, the Telegram bot token, and the chat ID are stored encrypted and encoded in the binary using string encryption method 2. These values are decrypted at runtime to enable C2 communication.

The `GetAdaptersInfo` API is used to retrieve the infected machine's MAC address, which uniquely identifies the machine to the C2 server.

In the next phase, TELESHIM enters a polling loop by sending HTTP GET requests to the following URL to fetch updates: `https://api.telegram.org/bot[BOT_TOKEN]/getUpdates?offset=[N]`.

Each HTTP GET request uses the hardcoded User-Agent below to impersonate web browser traffic:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8) AppleWebKit/534.31 (KHTML, like Gecko)
Chrome/13.0.748.0 Safari/534.31
```

The JSON response from the Telegram server is parsed using the [nlohmann::json](#) library written in C++ to extract the relevant fields.

Chat ID validation

The `message.chat.id` field parsed from the JSON response is validated against the hardcoded `chat_id` in the binary to prevent hijacking.

C2 message types

There are two C2 message types supported by TELESHIM that are processed with different handlers, which include the following:

- **Control messages:**

- TELESIM decrypts `message.text` using string decryption method 2. If the decrypted message is the string 13, TELESIM XOR-encrypts and Base64-encodes the machine's MAC address and sends it in response to the Telegram bot to register the infection.
- If the decrypted `message.text` is not the string 13, it is parsed as a command and executed using `cmd.exe /C` by calling `CreateProcessA`. TELESIM only executes commands addressed to its MAC address in the format `[MAC_ADDRESS] [SHELL_COMMAND]`. The command output is encrypted using method 2 and sent in chunks if it is larger than 1,000 bytes.

- **Download and execute messages:**

TELESIM decrypts `message.caption` using method 2; the result is in the format `[MAC address] [destination file path]`. It validates the MAC address against the machine's own MAC address and, if it matches, uses `document.file_id` parsed from the JSON response to fetch the file via `/bot[TOKEN]/getFile?file_id=[FILE_ID]`. The downloaded file is decrypted using the 44-byte rolling XOR key, and scheduled tasks are used to launch the dropped binary.

Post-compromise activity

At the time of analysis, ThreatLabz captured post-compromise activity from the C2 operator, including system, user, and network reconnaissance commands, along with the deployment of next-stage payloads. Most of the activity took place between July 7, 2026 and July 9, 2026. Using the original timestamps at which the threat actor issued the C2 commands, ThreatLabz performed a timing analysis and observed that all C2 commands were executed only between 4 AM UTC and 12 PM UTC, with a heavy concentration between 7 AM and 11 AM UTC. The figure below shows the time-series plot.

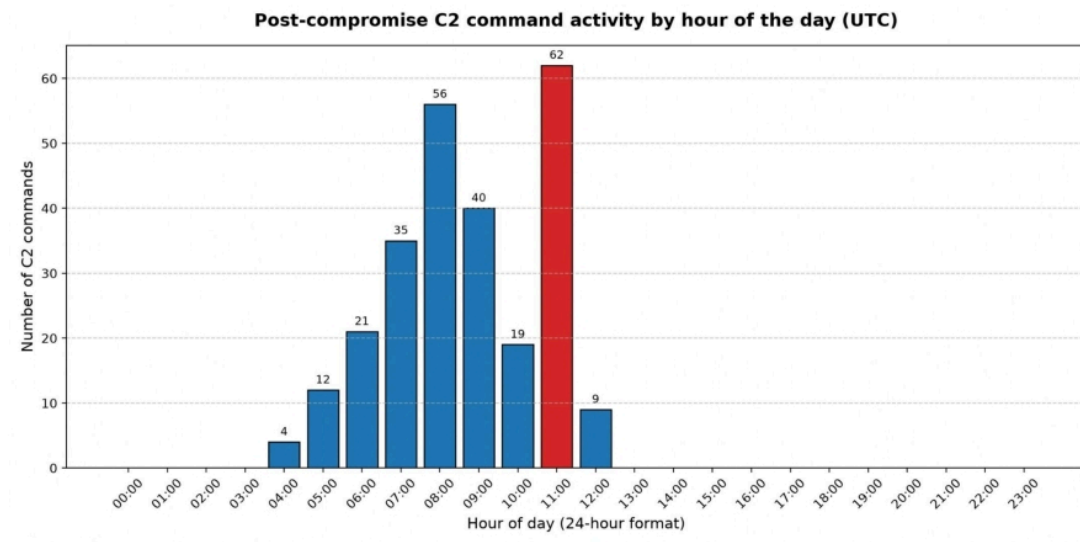


Figure 4: Time plot of C2 commands executed by the TELESIM threat actor.

The table below summarizes some of the C2 commands executed by the threat actor.

Category	C2 Command(s)	Description
System reconnaissance	net user tasklist hostname	Discover information about current users, list of running processes, and the hostname of the infected machine.
Network reconnaissance	ipconfig /all ipconfig /displaydns netstat -ano	Discover information about the system's network configuration and active network connections.

Category	C2 Command(s)	Description
File reconnaissance	dir c:\Users	Retrieve a list of files in key directories to gain intel, choose an appropriate staging directory, and validate successful deployment of next stage payloads.
	dir c:\Users\desktop	
	dir c:\Users\Downloads	
	dir C:\ProgramData\Crypto\DSS\	
	dir C:\ProgramData\	
	dir C:\ProgramData\Lenovo	
	dir C:\ProgramData\Intel	
Persistence	schtasks /create /f /sc minute /mo 10 /tn "Feedback" /tr "C:\ProgramData\Intel\winProAlertService.exe"	Create a scheduled task that runs every 10 minutes to execute the next stage.
Command verification	schtasks /Query /TN Feedback /v	Verify the scheduled task installation.
Network connection verification	ping cert.hypersnet[.]com -n 1	Verifies network connectivity
	ping ssl.blsouqs[.]com -n 2	
	ping contacts.ftabnews[.]com -n 2	

Table 1: Post-compromise commands executed by the TELESHIM threat actor.

Deployment of next stage payloads

For each infected machine, the threat actor enumerated the directories under the path C:\ProgramData\ to choose an appropriate staging directory. The following next-stage payloads were then deployed to the staging directory.

- Legitimate GoPro binary `GoProAlertService.exe`. For each infected machine, the legitimate binary was dropped with an appropriate name to blend in with the name of the staging directory.
- Legitimate `MSVCP120.dll`
- Legitimate `MSVCR120.dll`
- Malicious DLL - `pthreadVC2.dll` sideloaded by the legitimate binary

On each infected machine, the threat actor created the directory `C:\ProgramData\Crypto\DSS\` and dropped the final encrypted C2 implant named `C99F29AC08454855B3D538960BB2F34F.PCPKEY`. The directory name and the file extension were carefully chosen by the threat actor to impersonate files related to Microsoft's Platform Crypto Provider (PCP) in order to appear benign.

Once all the next-stage payloads were dropped, the threat actor created a scheduled task named *Feedback* that runs every 10 minutes and launches the legitimate binary, which sideloads the malicious DLL named *pthreadVC2.dll* present alongside in the same directory. ThreatLabz named this next stage loader MIXEDKEY.

MIXEDKEY reflective loader

MIXEDKEY is a Windows 64-bit DLL whose main purpose is to decrypt the contents of `C99F29AC08454855B3D538960BB2F34F.PCPKEY`, and reflectively load it.

Similar to the TELESIM backdoor, MIXEDKEY heavily uses MBA operations to generate junk instructions and opaque predicates, bloating the size of the binary and deterring reverse engineering.

String decryption

Unlike TELESIM, MIXEDKEY constructs decrypted strings at runtime by computing each byte using MBA expressions over hardcoded values in the `.data` section. These bytes are written out of order to a buffer to assemble the final string. To compute a single byte, MIXEDKEY executes approximately ~1,000 instructions.

Final payload decryption

The final payload is encrypted using two layers of XOR encryption. The second layer of encryption uses environmental keying. The threat actor used the volume serial number of the victim's machine to derive a key to encrypt the payload before deploying it. This ensures the payload decrypts and executes only on the intended target and complicates decryption by analysts who do not have the correct volume serial number.

MIXEDKEY decrypts the payload and reflectively loads it using the following steps:

- Fetches the 4-byte volume serial number by calling `GetVolumeInformationA`.
- Derives a 20-byte rolling XOR key from the 4-byte volume serial number by repeating the 4-byte serial number five times.
- Reads the contents of `C:\ProgramData\Crypto\DSS\C99F29AC08454855B3D538960BB2F34F.PCPKEY` to load the encrypted payload.
- Uses the first 311 bytes of this file as a rolling XOR key to decrypt the rest of the file.
- The decrypted output of the previous step is once again decrypted using the 20-byte rolling XOR key derived from the volume serial number.
- The final decrypted output contains a portable executable (PE) file with the "MZ" signature stripped off and the 4-byte payload size prefixed to the payload.

Finally, MIXEDKEY reflectively loads the PE file and invokes its export function to continue the next stage of the attack chain.

BINDCLOAK C2 implant

The final payload is a 64-bit C2 implant written in C++ that ThreatLabz tracks as *BINDCLOAK*. It beacons to the C2 server at `cert.hypersnet[.]com`. A detailed technical analysis of BINDCLOAK will be shared in a follow-up blog post.