

Weyhro C2: Because Ransomware Wasn't Paying the Bills Anymore

 lumma-labs.com/veyhro-c2-because-ransomware-wasnt-paying-the-bills-anymore-b136fd7ef100

Lumma Labs

December 5, 2025

| Acknowledgment: Thank you [@whoamix302](#) for sharing the sample!

WEYHRO C2



Возможности

SHELL

Интерактивный ввод для выполнения команд в режиме реального времени



SOCKS5

Быстрый и стабильный прокси, обеспечивает доступ к внутренней сети



HVNC

Запускает браузер с данными профиля и обеспечивает быструю, качественную отрисовку



KERBEROS DUMPER

Собирает данные билетов Kerberos (текущего сеанса или всех, если запущен с админ правами)



ADVANCED EVASION

STATIC

Polymorphic
Compression
AES/ChaCha20

RUNTIME

Unhooking
AMSI/ETW Bypass
Memory/CallStack Scan Evasion



ОГРАНИЧЕНИЯ

ПО не должно запускаться на компьютерах CIS



ВОЗВРАТ

Только в случае полной неработоспособности ПО FUD не гарантируется



ЦЕНА

\$3000 в месяц

Включает документацию, обновления, исправления ошибок и поддержку



On December 3rd, 2025, a someone using the handle **weyhro** posted an advertisement on a cybercrime forum for a product called **Weyhro C2**. The product is marketed as an “advanced pentesting toolkit designed for stealth and defense evasion”, equipped with a full C2 infrastructure. According to the advertisement, the agent is designed to evade detection and bypass modern AV/EDR solutions on corporate networks, enabling capabilities across the entire spectrum of operations, from initial access and persistence to lateral movement and target engagement.

The advertisement highlights a modular architecture with the following core features:

- Reverse shell with panel control
- Fast proxy for proxychains or residential proxy use
- — Hidden VNC with browser session capture (cookies, passwords, tabs)
- — Ticket extraction from current/all sessions (LSASS dump coming soon)
- — Polymorphic code, AES encryption, hook removal, AMSI/ETW patching, memory scan bypass
- — File manager with upload/download capability

The seller claims the agent runs as shellcode entirely from memory, is “completely encrypted”, and launched via a loader. The license is priced at **\$3,000/month** with support, updates, and fixes included. Payment is accepted in BTC or USDT, with a restriction that the software must not run on CIS (Commonwealth of Independent States) systems.

Digging a bit further into the threat actor posting this tool — it appears they may be behind the Weyhro ransomware group. The Weyhro ransomware group reportedly became active sometime around March 2025. The timing of account creation preceding ransomware activity, and now the release of a full C2 toolkit suggests this is the same threat actor expanding their operations from ransomware to selling offensive tooling to other criminals.

Leaks

Sensitive data from various companies, there is something for everyone.

Aug 10, 2025 Chemtron RiverBend \$10M, USA 25K	Aug 10, 2025 Community Services of Missouri \$5M, USA 24K	May 31, 2025 Adriatic Glass & Mirrors \$10M, Canada 48K
May 30, 2025 Terra Caribbean \$30M, Barbados 49K	May 26, 2025 Synergy Investments \$13M, USA 48K	May 8, 2025 101 Arch Street \$8M, USA 55K
Mar 31, 2025 Valens Bank/Pay/Exchange \$13M, Germany 63K	Mar 21, 2025 Montgomery Little & Soran, PC \$16M, USA 61K	Mar 10, 2025 McMillan James Equipment Company (MJEC) 60K

Weyhro leak site

So, What is Weyhro C2?

ChaCha20 String Decryption

The payload stores the encrypted strings in a global table. Each string is identified by a 32-bit String ID (NOT a hash of the plaintext) and decrypted using ChaCha20.

Each entry contains everything needed to decrypt its strings: a pointer to the encrypted data, the cryptographic key, and nonce.

The decryption steps:

- Iterate through the string table comparing each entry's String ID with the requested ID
- Retrieve encrypted data pointer, ChaCha20 key (32 bytes), and nonce (12 bytes)
- Initialize state with "expand 32-byte k" constant, perform 20 rounds, XOR keystream with ciphertext

Example:

```
buffer[] = { }; (, buffer, (buffer));
```

Key String IDs identified:

ID=0x6776C0F8: kernel32.dllID=0xA0BBDB3: ntdll.dllID=0x565E35E0: user32.dllID=0x9F894C6C:
advapi32.dllID=0x15971A3E: secur32.dllID=0xBFBDAB86: ws2_32.dllID=0xFA852BBF:
CreateFileAID=0xFCF366AF: GetFileSizeID=0x85AB24C4: ReadFileID=0x45AD6CF1:
CloseHandleID=0xC9834F5F: OpenProcessTokenID=0x54B5F192: GetCurrentProcessID=0xFA16AE04:
GetTokenInformationID=0x054921F3: VirtualAllocID=0xE038FF15: VirtualAllocExID=0xD85B1113:
VirtualProtectExID=0x27895BAD: CreateProcessWID=0x585AEB5C:
GetCurrentProcessIdID=0xFB274884: GetProcessHeapID=0x8D095AC3: HeapAllocID=0xF4CFA78F:
HeapFreeID=0xA4CA99F4: LsaConnectUntrustedID=0x58F05DB2:
LsaRegisterLogonProcessID=0x87BF6929: LsaLookupAuthenticationPackageID=0xAF3758EF:
LsaEnumerateLogonSessionsID=0x80DF6DD3: LsaGetLogonSessionDataID=0x9DD10C70:
LsaCallAuthenticationPackageID=0x7EAE305: LsaFreeReturnBufferID=0x5D286218:
DuplicateTokenExID=0x7D011A19: ImpersonateLoggedOnUserID=0xA2C5320B:
RevertToSelfID=0x14C545C7: NtAllocateVirtualMemoryID=0x8DB2F7D3:
NtProtectVirtualMemoryID=0xA38A84FE: NtCloseID=0x8F4D8C6A: FindWindowWID=0x698F3F83:
LoadLibraryAID=0x74028CBA: WriteConsoleAID=0x5904F8CA: GetModuleFileNameWID=0x83572B88:
GetStdHandleID=0x3CC19F79: WSASStartupID=0xFFFC1AD2: WSACleanupID=0x5A91A8F5:
WSASocketWID=0xD091D3D9: connectID=0x1D294A01: sendID=0x23BC9501: recvID=0x0CE82532:
closesocketID=0xAEA8C55D: CreatePipeID=0x3905EABB: DuplicateHandleID=0xA3309BD0:
WaitForSingleObjectID=0xBA557E2E: NtWriteVirtualMemoryID=0x2B9B05BA:
NtReadVirtualMemoryID=0xDA72FBB8: NtQueryInformationProcessID=0x44DFA62D:
NtOpenProcessID=0x418178B0: NtCreateThreadExID=0xB209EA4C: NtSuspendThreadID=0x5D5A3E1C:
NtResumeThreadID=0x3B54C106: NtFreeVirtualMemoryID=0x194DBA69:
NtMapViewOfSectionID=0x2C19C7AB: NtUnmapViewOfSectionID=0x639D9E97:
NtReadFileID=0xD8472464: EtwEventWriteID=0x39AB148E: AmsiScanBufferID=0x0D5406A3:
kernelbase.dllID=0xDE895451: amsi.dllID=0x749AA3C2:
C:\Windows\System32\ntdll.dllID=0xCC2CE509: LdrGetDllHandleID=0xBE78AB40:
WriteProcessMemoryID=0x39B5C443: ResumeThreadID=0x57CE9C93: GetThreadContextID=0x78E12844:
SetThreadContextID=0xD9F6AD83: IsDebuggerPresentID=0x12BF02D5:
CreateToolhelp32SnapshotID=0x1B105040: Process32FirstWID=0xD08B9B65:
Process32NextWID=0x70DD9366: OpenThreadTokenID=0x42657133: GetCurrentThreadID=0xE1518A7F:
GlobalMemoryStatusExID=0x8F063A3A: NetGetJoinInformationID=0x4DF495A7:
RegOpenKeyExWID=0xF5944F88: RegCloseKeyID=0xD5B50261: VirtualProtectID=0x54ABC644:
RegOpenKeyExAID=0x35BC2C28: RegQueryValueExWID=0xA1FBF2A4: NtOpenKeyID=0xD6145070:
NtCreateKeyID=0xD58E9518: GetProcessIdID=0x5B1ED11D: EnumWindowsID=0xDB83907E:
GetWindowThreadProcessIdID=0x1B8BE250: DestroyWindowID=0x71F73233:
CreateRemoteThreadID=0xC79ABA16: CreateThreadID=0x174BFE01: WriteFileID=0xBDC11BED:
SleepID=0x8307F657: PeekNamedPipeID=0x7FDFDB0C: FlushFileBuffersID=0x7E1DCA0E:
socketID=0x5E667354: ioctlsocketID=0xEACD5301: selectID=0x9E80EE1E:
WSAGetLastErrorID=0xD78C25D8: NtCreateNamedPipeFileID=0x15E8CE02:
NtWriteFileID=0xE6D03D06: GetUserNameAID=0x3A03B251: GetComputerNameAID=0xD9AB418F:
RtlGetVersionID=0x145FDA43: getsocknameID=0x447929D9: bindID=0xC6B770FF:
listenID=0x492D1331: acceptID=0x399B44D1: setsockoptID=0xE5E579D9:
getsockoptID=0x1440A192: gethostbynameID=0x35648B37: gdi32.dllID=0xFB4A7F05:
InitializeSecurityDescriptorID=0xC493F8F2: SetSecurityDescriptorDaclID=0xDC393E2:
CreateDesktopAID=0xCB4ACFAD: SetThreadDesktopID=0xA5911DE8: CloseDesktopID=0x1C8C3300:
GetGuiThreadInfoID=0x9AAF4453: GetThreadDesktopID=0xC0FBA7DD: GetWindowRectID=0xDAC57377:
ScreenToClientID=0x186E188A: GetClientRectID=0xD39F1F69: FindFirstFileWID=0x438B6025:
FindNextFileWID=0x48E9CC8D: FindCloseID=0xB5EDF6AE: CreateFileWID=0x5411E5CA:
DeleteFileWID=0x18B521D1: CreateDirectoryWID=0xD9DFEC46: MoveFileWID=0x99E2E690:
CopyFileWID=0xAC23A6F8: GetFileAttributesWID=0x4A545BAB: GetFileTimeID=0xB51B3B0F:
FileTimeToSystemTimeID=0xF1682DA2: SystemTimeToFileTimeID=0x369B65D6:
SetFilePointerID=0x321189A8: cmd.exeID=0xF5822B0C: PostMessageAID=0x760E88E4:

GetCurrentThreadIdID=0xAFD4DAC7: CreateProcessAID=0xC6C22D53: GetLastErrorID=0x458B2170:
OpenProcessID=0x9683953F: TerminateProcessID=0x3C9D8FA6: Thread32FirstID=0x594F8166:
Thread32NextID=0x91CBB3F2: Process32FirstID=0xFBB9F934: Process32NextID=0xDDA147F6:
FreeConsoleID=0x6B3C4E81: GetDCID=0x99C99F83: ReleaseDCID=0x4263A3C3:
CreateCompatibleDCID=0x51D2202F: DeleteDCID=0x3A90E064:
CreateCompatibleBitmapID=0x290B3CAA: DeleteObjectID=0x7A1123AE: SelectObjectID=0x62034019:
BitBltID=0x71093716: GetDIBitsID=0xBB8517F5: GetWindowDCID=0x4A8FF765:
NetApiBufferFreeID=0xDB6F3320: NetWkstaGetInfoID=0xCD94BB1E: Netapi32.dllID=0x561C5F24:
NtQueryInformationTokenID=0x4C8C2C4D: NtQuerySystemInformationID=0x42755D7B:
GetModuleHandleWID=0x4D3B07B9: GetProcAddressID=0x94ECA6C7: RtlCreateHeapID=0xF342DE1E:
RtlDestroyHeapID=0x4941F646: RtlAllocateHeapID=0x647015F6: RtlFreeHeapID=0x2E3C4C11:
RtlReAllocateHeapID=0x1E4A758F: CreateFileMappingWID=0x1BC2EC6C:
MapViewOfFileID=0xCD67729E: NtSetValueKeyID=0x60C24771: shell32.dllID=0x6FDC6558:
SHGetFolderPathWID=0x33F7E0C1: NtWaitForSingleObjectID=0xB3160562:
Microsoft\Windows\SecurityID=0x187C5415: msdefcore.exeID=0xCACF8C87:
WMIProviderHostID=0x6E0419B9: Software\Microsoft\Windows\CurrentVersion\RunID=0xCE93839B:
ReadProcessMemoryID=0x722285DC: C:\Windows\System32\cmd.exeID=0xD5D51295:
LsaDeregisterLogonProcess

EDR/AV Unhooking Mechanism

The snippet below implements an anti-EDR technique: loading a copy of `ntdll.dll` directly from disk to obtain unhooked function bytes. EDR and AVs typically hook `ntdll.dll` functions in memory by inserting JMP instructions at function prologues, but they cannot modify the file stored on disk. By reading the original file, the payload obtains clean, unmodified function bytes that can later be used to overwrite the hooked versions in memory.

The code first decrypts the target file path using ChaCha20 encryption. The String ID `0x749AA3C2` (shown as decimal `1956291522` in the decompiled code) serves as a lookup key into the encrypted string table, which decrypts to the full path `C:\Windows\System32\ntdll.dll`.

Before performing any file operations, the payload must resolve the required Windows API functions — `CreateFileW`, `GetFileSize`, `VirtualAlloc`, `ReadFile`, and `CloseHandle`. All five APIs reside in `kernel32.dll`, identified by String ID `0x6776C0F8`. Each API name is itself stored as a ChaCha20-encrypted string with its own String ID: `0xFA852BBF` for `CreateFileW`, `0xFCF366AF` for `GetFileSize`, `0x85AB24C4` for `VirtualAlloc`, `0x45AD6CF1` for `ReadFile`, and `0x054921F3` for `CloseHandle` (refer to the table of string IDs above).

With the API pointers resolved, the payload opens `ntdll.dll` from disk with `CreateFileW` with `GENERIC_READ` access and `FILE_SHARE_READ` sharing mode. The `OPEN_EXISTING` flag ensures the call fails if the file doesn't exist. After obtaining a valid file handle, it queries the file size using `GetFileSize`, then allocates a memory buffer of exactly that size using `VirtualAlloc`. The allocation uses `MEM_COMMIT | MEM_RESERVE` (0x3000) with `PAGE_EXECUTE_READWRITE` (0x40) protection, giving the buffer read, write, and execute permissions.

The payload then reads the entire `ntdll.dll` file into the allocated buffer, passing the file handle, buffer pointer, and expected size. After the read completes, it immediately closes the file handle and verifies that the number of bytes actually read matches the expected file size. If this validation succeeds, the buffer now contains an exact byte-for-byte copy of the clean, unhooked `ntdll.dll` from disk. This clean copy acts as a reference for the following unhooking operation. The payload parses the PE structure of the clean disk copy, locates the `.text` section by searching for the magic bytes `0x747865742E` (".text"), and then overwrites the entire hooked `.text` section in memory with the clean bytes from disk with the patch memory function below. This effectively removes any EDR inline hooks by restoring the original function prologues.

```

1 char __fastcall mw_PatchMemory(__int64 targetAddress, __int64 sourceBuffer, __int64 size, unsigned int newProtect)
2 {
3     unsigned int (__fastcall *pfnVirtualProtect_tmp)(__int64, __int64, __int64, unsigned int *); // rdx
4     unsigned int (__fastcall *VirtualProtect)(__int64, __int64, __int64, unsigned int *); // rdi
5     unsigned int (__fastcall *WriteProcessMemory_tmp)(__int64, __int64, __int64, __int64, __int64 *); // rdx
6     unsigned int (__fastcall *WriteProcessMemory)(__int64, __int64, __int64, __int64, __int64 *); // r12
7     unsigned int protectResult; // ecx
8     char result; // al
9     unsigned int oldProtect; // [rsp+2Ch] [rbp-3Ch] BYREF
10    __int64 bytesWritten[7]; // [rsp+30h] [rbp-38h] BYREF
11
12    if ( (mw_ResolveAPIByStringID(1735835896, -709557663) & 1) == 0 // kernel32.dll!VirtualProtect
13        || (VirtualProtect = pfnVirtualProtect_tmp, (mw_ResolveAPIByStringID(0x6776C0F8, 0xBE78AB40) & 1) == 0) // kernel32.dll!WriteProcessMemory
14    ) {
15        sub_7FF766806F63();
16    }
17    WriteProcessMemory = WriteProcessMemory_tmp;
18    oldProtect = 0;
19    if ( !VirtualProtect(targetAddress, size, 4, &oldProtect) )
20        return 0;
21    bytesWritten[0] = 0;
22    if ( !WriteProcessMemory(-1, targetAddress, sourceBuffer, size, bytesWritten) || bytesWritten[0] != size )
23    {
24        ((void (__fastcall *))(__int64, __int64, _QWORD))VirtualProtect)(targetAddress, size, oldProtect);
25        return 0;
26    }
27    protectResult = VirtualProtect(targetAddress, size, newProtect, &oldProtect);
28    result = 1;
29    if ( !protectResult )
30        return 0;
31    return result;
32}

```

Patch Memory function

ETW and AMSI Patching

The ETW patch is simple: it finds `EtwEventWrite` (or similar ETW function) in `ntdll.dll` and overwrites the first byte with `0xC3` (RET). This makes the function return immediately without doing anything, effectively disabling Event Tracing for Windows telemetry.

```

562 v39 = (const void *)mw_ChaCha20Decrypt(0xD8472464, (__int64)v306, 0x40u); // EtwEventWrite
563 if ( !v39 )
564     goto LABEL_328;
565 if ( (mw_CompareMemory(v38, v39, v40, 1) & 1) != 0 )
566 {
567     v309[0].m128i_i8[0] = 0xC3;
568     v42 = mw_PatchMemory(v41, (__int64)v309, 1, 0x20u);
569 }
570 else
571 {
572     v42 = 0;
573 }

```

ETW patching

The payload bypasses AMSI by patching the `AmsiScanBuffer` function directly in memory. If `amsi.dll` is not loaded in the process, the patch is skipped and the payload logs `[~] AMSI patch skipped: amsi.dll not loaded`.

If `amsi.dll` is present, the payload locates the function `AmsiScanBuffer`. Once found, it overwrites the first three bytes of the function with `31 C0 C3`. The bytes `31 C0` disassemble to `XOR EAX, EAX` which sets the return value to zero, and `C3` is the `RET` instruction which returns immediately to the caller. Since AMSI uses the return value to determine scan results, returning zero indicates `AMSI_RESULT_CLEAN`, meaning the scanned content is safe.

```
615 v48 = (const void *)mw_ChaCha20Decrypt(0x39AB148E, (__int64)v306, 0x40u); // AmsiScanBuffer
616 if ( v48 )
617 {
618     if ( (mw_CompareMemory(v47, v48, v49, 1) & 1) != 0 )
619     {
620         v309[0].m128i_i16[0] = 0xC031;
621         v309[0].m128i_i8[2] = 0xC3;
622         v51 = mw_PatchMemory(v50, (__int64)v309, 3, 0x20u);
623     }
624     else
625     {
626         v51 = 0;
627     }
```

AMSI patching

Persistence

The payload establishes persistence by copying itself to `%APPDATA%\Microsoft\Windows\Security\msdefcore.exe`. It first calls `SHGetFolderPathW` with `CSIDL_APPDATA` (26) to get the user's `AppData\Roaming` folder, then appends the path `Microsoft\Windows\Security\msdefcore.exe`. Before copying, it checks if a file already exists at that location using `GetFileAttributesW`, and if so, deletes it with for a clean installation. It then copies the currently running executable to the destination.

After placing the file, the payload writes a registry value to ensure it runs on every user login. It opens `HKEY_CURRENT_USER` with `RegOpenKeyExA`, then uses `NtCreateKey` to create or open the subkey `Software\Microsoft\Windows\CurrentVersion\Run`. Finally, it calls `NtSetValueKey` to write a value named `WMIPProviderHost` pointing to the dropped executable.

Process Injection

The payload uses a process hollowing technique to inject its payload into a legitimate Windows process. It decrypts the target process path `C:\Windows\System32\cmd.exe` using `ChaCha20`, then resolves the necessary APIs including `CreateProcessA`, `VirtualAllocEx`, `WriteProcessMemory`, `VirtualProtectEx`, and `ResumeThread`. It spawns `cmd.exe` in a suspended state using the `CREATE_SUSPENDED` flag (0x4), which prevents the process from executing until explicitly resumed.

Once the suspended process is created, the payload allocates executable memory in the target process using `VirtualAllocEx`, then writes the decrypted payload using `WriteProcessMemory`. It queries the process information via `NtQueryInformationProcess` to locate the PEB, validates the payload's PE headers, and determines whether the payload is x86 (0x14C) or x64 (0x8664). The payload then constructs a small shellcode stub and calls `ResumeThread`.

For x64:

```
mov rax, x<payload_address> ;
```

For x86:

```
mov eax, x<payload_address> ;
```

HVNC?

As described in the forum post, the payload includes Hidden Virtual Network Computing (HVNC). Unlike traditional VNC which mirrors the victim's visible screen, HVNC creates an invisible desktop session that runs completely hidden from the user. The payload accomplishes this by calling `CreateDesktopA` to spawn a new desktop object that exists in a separate session from the user's active desktop, then uses `SetThreadDesktop` to switch its threads to this hidden workspace. Any applications launched on this invisible desktop will execute without any visual indication to the victim.

The screen capture and streaming functionality is supported by a full suite of GDI (Graphics Device Interface) APIs present in the encrypted string table: `GetDC/GetWindowDC` to obtain device contexts, `CreateCompatibleDC` and `CreateCompatibleBitmap` to set up off-screen buffers, `BitBlt` to perform fast screen captures, and `GetDIBits` to extract the raw pixel data for transmission to the C2 server. Supporting functions like `SelectObject`, `DeleteObject`, `DeleteDC`, and `ReleaseDC` handle proper resource management. The `GetWindowRect` and `GetGUIThreadInfo` APIs allow the threat actor to enumerate and interact with windows on the hidden desktop. While the actual HVNC implementation code was not found in this payload, the presence of these 16+ HVNC-specific API entries in the encrypted string table confirms this payload has full hidden desktop and screen capture capabilities ready for deployment.

Kerberos Dumper?

As described in the forum post, the payload includes Kerberos ticket dumping capability. The encrypted string table contains the complete API infrastructure required for extracting Kerberos tickets from the Local Security Authority (LSA):

- `LsaCallAuthenticationPackage` - Sends ticket retrieval requests to the Kerberos SSP
- `LsaLookupAuthenticationPackage` - Locates the Kerberos authentication package
- `LsaConnectUntrusted` / `LsaFreeReturnBuffer` - Establishes connections to LSA and manages memory

- `NtQueryInformationToken` - Retrieves token details including logon session info
- `NtQuerySystemInformation` - Enumerates system-wide logon sessions
- `OpenProcessToken` - Opens process tokens for inspection
- `ImpersonateLoggedOnUser` - Impersonates user contexts to access their tickets

The Kerberos dumping workflow typically involves: connecting to LSA via `LsaConnectUntrusted`, looking up the Kerberos package with `LsaLookupAuthenticationPackage`, then calling `LsaCallAuthenticationPackage` with `KerbRetrieveEncodedTicketMessage` to extract TGTs and service tickets from the current session or all sessions when running with elevated privileges. The presence of `NtQuerySystemInformation` suggests capability to enumerate all logon sessions system-wide.

Join Medium for free to get updates from this writer.

While the actual Kerberos dumping implementation code was not found in this binary, the presence of 40+ credential/token-related API entries in the encrypted string table confirms this payload has the capabilities for full Kerberos ticket extraction.

Secondary Payload Decryption

The payload uses a custom AES-128-CTR implementation to decrypt its secondary payload, followed by LZSS decompression to extract the secondary payload.

Custom AES Implementation

The payload implements its own AES-128 encryption, computing the S-Box at runtime using the standard GF(2⁸) multiplicative inverse algorithm. This runtime generation avoids embedding the recognizable 256-byte AES S-Box table in the binary.

The key expansion follows the standard AES schedule but uses the custom S-Box for the SubWord operation. The encryption operates in CTR (Counter) mode with an 8-byte IV and an 8-byte counter.

The AES key (`9B E2 55 CF 18 D8 D1 13 51 42 8B 32 11 A9 E8 02`) and IV (`74 DC 9A EE 3E D9 94 17`) are hardcoded in the binary.

LZSS Decompression

After decryption, the output is a LZSS-compressed data that must be decompressed to recover the secondary payload. The implementation uses a bit-stream encoding:

- : LSB is 0, followed by 8-bit literal byte value
- : LSB is 1, followed by 8-bit length and 16-bit offset

Once APIs are resolved, the loader parses the incoming PE file's headers, starting with the DOS header's `e_lfanew` field to locate the NT headers. It then allocates memory at the PE's preferred `ImageBase` using `VirtualAlloc` with `MEM_COMMIT | MEM_RESERVE` flags. If the preferred address is unavailable, it falls back to allocating at any available location. The loader proceeds to map each section from the raw PE data to its corresponding virtual address, copying `.text`, `.data`, `.rdata`, and other sections to their proper offsets within the allocated memory region.

After imports are resolved, the loader applies appropriate memory protections to each section based on their characteristics flags, converting PE section attributes to Windows memory protection constants like `PAGE_EXECUTE_READ`, `PAGE_READONLY`, and `PAGE_READWRITE`. A call to `FlushInstructionCache` helps the CPU to recognize the newly written executable code. Finally, the loader transfers control to the agent by calling its entry point with `DLL_PROCESS_ATTACH`.

The Agent

The agent has the same functionality as in the initial payload to decrypt strings via ChaCha20 and resolve Windows APIs dynamically using encrypted string IDs. The following string IDs were identified during the decryption:

```
0xA00163E1: kernel32.dll0x4FBEACC0: ntdll.dll0xE89D8CFB: user32.dll0xA508724D:
advapi32.dll0x1B5C2B09: ws2_32.dll0x32B6440D: CreateFileA0xE006D837:
GetFileSize0x5A14D573: ReadFile0xDF166002: CloseHandle0xE40747F3: VirtualAlloc0x7DB3FC25:
VirtualAllocEx0x64E8E531: VirtualProtectEx0x3B4A43E0: CreateProcessW0x07482DF7:
GetCurrentProcessId0xED8676E3: NtAllocateVirtualMemory0x87EDC5DC:
NtProtectVirtualMemory0xC7A11913: NtClose0xBED3FAED: FindWindowW0xA53B62D4:
LoadLibraryA0xDEF5536C: WriteConsoleA0x95FB4873: GetModuleFileNameW0x65E1B0D8:
GetStdHandle0x6C2A34B2: WSASStartup0x3154B1B0: WSACleanup0xD510A9EE: WSASocketW0x929CF63A:
connect0x70322EE5: send0x4E2921A8: recv0x2916ABC5: closesocket0x488BB71C:
CreatePipe0x5E256D58: DuplicateHandle0x252C7570: CreateRemoteThread0x1745E5BE:
Sleep0xD6A0D3D0: PeekNamedPipe0x791C1B7C: OpenProcessToken0xA8EF236F:
GetUserNameA0x19E1BAE1: Software\\Microsoft\\Windows\\CurrentVersion\\Run0x093BFC4D:
cmd.exe
```

Commands:

```
file - File operations (list/read/write/delete)
injt - Process injection via CreateRemoteThread
sock - SOCKS5 proxy
shll - Reverse shellping
heartbeat - Heartbeat (responds "pong")
proc - Process/network enumeration
```

File Operations

The file operations module supports multiple sub-commands for interacting with the victim's file system. Based on the error strings embedded in the binary, the supported operations include directory enumeration via `FindFirstFile/FindNextFile`, file reading with configurable position seeking, file writing for data exfiltration staging or payload dropping, and file deletion.

- "Failed to find first file" - Directory listing (FindFirstFileW)- "Failed to set file position" - File seeking (SetFilePointer)- "Failed to read file" - File reading (ReadFile)- "Failed to open file" - File access (CreateFileW)- "Failed to write file" - File writing (WriteFile)- "Failed to delete file" - File deletion (DeleteFileW)

The handler parses sub-command parameters from the C2 packet payload, extracting the operation type, target path, and any additional parameters such as file offsets or data buffers. All file paths are processed through `GetFullPathNameA`.

Process Injection Command (`injt`)

The injection handler implements classic CreateRemoteThread-based process injection, a technique commonly used by payload for code execution in the context of a legitimate process.

The handler parses a structured packet from the C2 server with a specific format: after skipping the 4-byte command identifier ("injt"), the code reads the target process ID as a DWORD at offset +0, the shellcode length as a DWORD at offset +4, and the raw shellcode bytes starting at offset +8.

The handler validates the packet size before processing: it first checks that at least 8 bytes exist beyond the command (minimum header), then verifies the total packet size accommodates the declared shellcode length.

```
0x18000D22B: mov edx, 0xAA953F88 ; GetProcAddress/GetModuleHandle0x18000D245: mov edx, 0x7DB3FC25 ; VirtualAllocEx0x18000D25F: mov edx, 0x89F3E8B9 ; WriteProcessMemory0x18000D27B: mov edx, 0x64E8E531 ; VirtualProtectEx0x18000D295: mov edx, 0x252C7570 ; CreateRemoteThread0x18000D2B6: mov edx, 0xDF166002 ; CloseHandle
```

The injection sequence is as follows: The handler first calls `OpenProcess` with `PROCESS_ALL_ACCESS` (0x1F0FFF) to obtain a handle to the target process. Memory is then allocated in the target process via `VirtualAllocEx` with `MEM_COMMIT | MEM_RESERVE` (0x3000) flags. The retrieved payload is written using `WriteProcessMemory`, followed by a memory protection change via `VirtualProtectEx` to `PAGE_EXECUTE_READWRITE` (0x40). Finally, `CreateRemoteThread` executes the injected code, and the process handle is closed for cleanup.

SOCKS5 Proxy Command (`sock`)

The `sock` command starts a SOCKS5 proxy on the victim, allowing the attacker to route traffic through the compromised machine to access internal resources behind the firewall or pivot to other systems on the network.

Upon receiving the `sock` command, the handler initializes a SOCKS5 server instance that listens for incoming proxy requests from the C2 infrastructure. The implementation follows the [RFC 1928 SOCKS5 protocol](#) specification, supporting the `CONNECT` command for TCP relay. Error handling strings `socks5_invalid` and `[agent] Winsock FAIL` indicate built-in validation and network error recovery.

Reverse Shell Command (**sh11**)

The reverse shell handler gives an interactive command-line access to the compromised system.

The shell implementation creates a hidden **cmd.exe** process with redirected standard handles. The payload uses **CreatePipe** for setting up anonymous pipes, and **CreateProcessW** for spawning the command interpreter. The pipes redirect the shell's stdin, stdout, and stderr, enabling bidirectional communication with the C2 server.

```
- CreatePipe() for stdin redirection- CreatePipe() for stdout/stderr capture -  
CreateProcessW("cmd.exe", ..., CREATE_NO_WINDOW)- Continuous read/write loop with C2
```

Heartbeat Command (**ping**)

The ping handler implements a simple heartbeat mechanism for connection keepalive and implant health verification.

When the C2 server sends a **ping** command, the handler responds with a **pong** message. This is pretty common in malware (like RATs) to verify the implant is still alive and maintain an active connection with the C2 server.

Process Enumeration Command (**proc**)

The **proc** handler collects network and system information for reconnaissance. The handler resolves **GetAdaptersInfo** to enumerate network adapters and IP addresses.

C2 Communications Protocol

A notable weakness in this payload is its use of unencrypted plaintext for all command-and-control communications. Despite the ChaCha20 and AES encryptions used to obfuscate strings within the binary and decrypt the secondary payload, the actual network traffic between the implant and C2 server is transmitted in cleartext.

The protocol follows a simple length-prefixed format. The function **blow** prepares outbound data by converting the payload length to big-endian (network byte order), sends the 4-byte length prefix, then transmits the raw payload data via the Winsock **send()**. I am not seeing any encryption occurring at any point in this chain.

```

9   v3 = data_len;
10  v4 = data_buffer;
11  v5 = socket;
12  len_prefix[0] = _byteswap_ulong(data_len);
13  result = mw_send_dec_str(socket, (__int64)len_prefix, 4u);
14  if ( (_DWORD)result == 12 )
15      return mw_send_dec_str(v5, v4, v3);
16  return result;
17 }

```

Sending data with 4-byte length prefix

Weyhro C2 Panel

Pivoting to the C2 address using one of my favorite tools — Censys, we can find the panel easily.

46.37.123.214

As of: Dec 04, 2025 9:50pm UTC | Latest

[Summary](#) [History](#) [WHOIS](#) [Explore](#)

Basic Information

Forward DNS upleake.pro

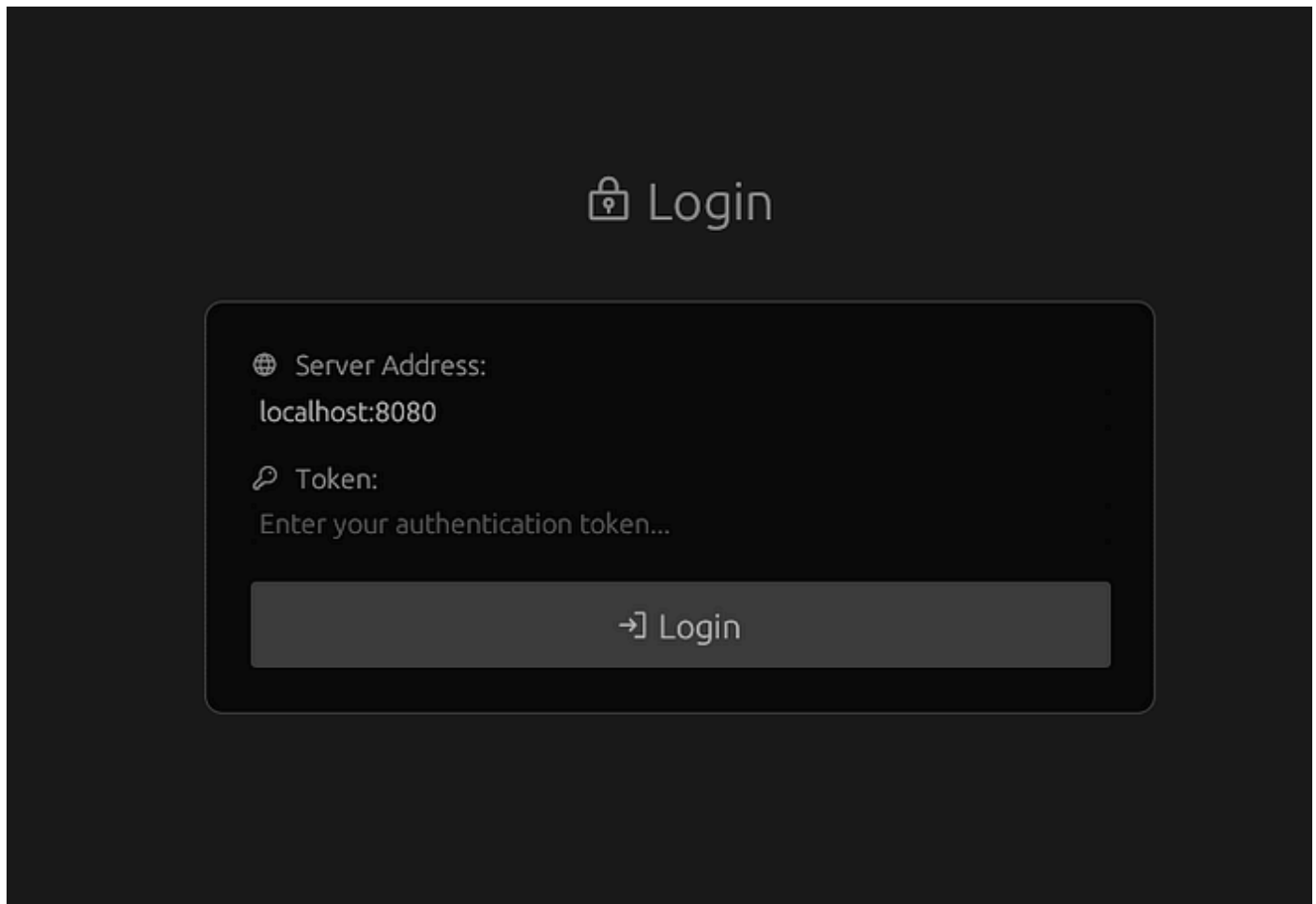
Routing 46.37.123.0/24 via FREAKHOSTING _ GB (AS215703)

OS Microsoft Windows

Services (7) 80/HTTP, 135/DCERPC, 139/NETBIOS, 443/HTTP, 445/SMB, 3389/RDP, 5357/HTTP

Labels [FILE SHARING](#) [NETWORK ADMINISTRATION](#) [REMOTE ACCESS](#)

Leveraging Censys to discover the panel



Weyhro C2 Login Panel

Looking at the source code, we can see some comments in Russian, referring to copying static files for a noVNC web-based + VNC viewer setup, as well as **weyhro c2** in title.

```

1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5     <meta charset="UTF-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=no">
7     <title>Weyhro C2</title>
8
9     <!-- Копируем статические файлы (noVNC + vnc_viewer.html) -->
10
11
12 <style>
13     html,
14     body {
15         margin: 0;
16         padding: 0;
17         width: 100%;
18         height: 100%;
19         overflow: hidden;
20         background: #1e1e1e;
21         position: fixed;
22     }
23
24     #c2-canvas {
25         width: 100vw;
26         height: 100vh;
27         display: block;
28     }

```

Weyhro C2 Panel Source Code

Yara Rule

[Yara-Rules/WeyhroC2/win_mal_veyhroc2.yar at main · RussianPanda95/Yara-Rules](#)

[Repository of Yara Rules. Contribute to RussianPanda95/Yara-Rules development by creating an account on GitHub.](#)

github.com

Indicators of Compromise

Payloads:

ec4ab4e4d700c9e5fdda59eb879a2bf18d0eefd825539d64677144d43a744cee
357e649b3b03ffe0d083092c0ed870c5185d64f14f5735ae43a8343269488dc3
8838278fff9a80d027951999d9f7c811ec2928afc39367f42f48f4411d1416b9

C2: 46.37.123[.]214