Analysis of the BlackMatter ransomware

tesorion.nl/en/posts/analysis-of-the-blackmatter-ransomware/

By Gijs Rijnders

August 5, 2021

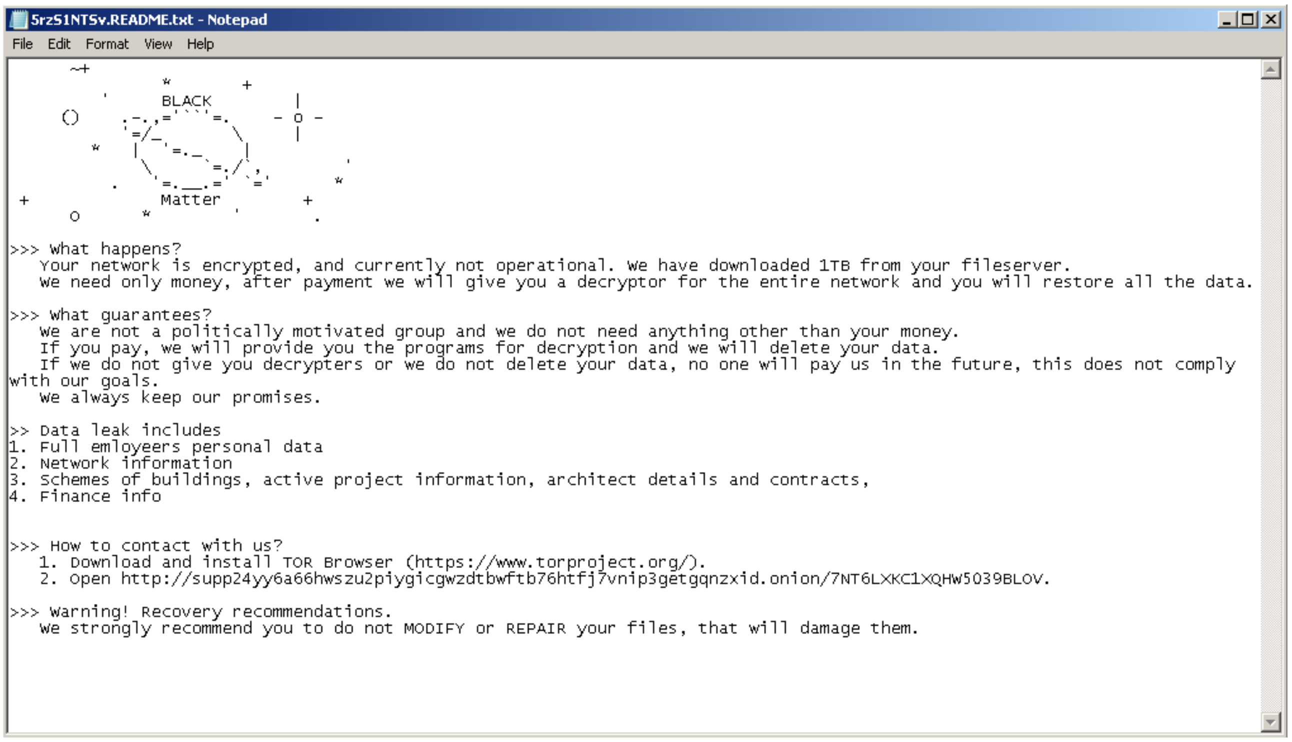
In late July, BleepingComputer stated that the notorious DarkSide ransomware gang has rebranded as BlackMatter. DarkSide disappeared after the high-profile attack on Colonial Pipeline this year, but they seem to be back under a new name.

We recently came across a sample of the BlackMatter ransomware, posted by GrujaRS. As not yet much information is available on this ransomware, we decided to write about its inner workings and technical details. As expected, we found some similarities between DarkSide and BlackMatter.

BlackMatter employs slight obfuscation such as dynamic API resolving and basic string encryption mechanisms. In this blog, we will give a general overview of the ransomware. Furthermore, we will discuss the obfuscations, as well as the file encryption and some other details.

Overview

The BlackMatter ransomware sample we analyzed is a 32-bit Windows executable. Files encrypted by it get the extension: ‘.5rzS1NTSv’, and the ransom note as shown is displayed to the user.

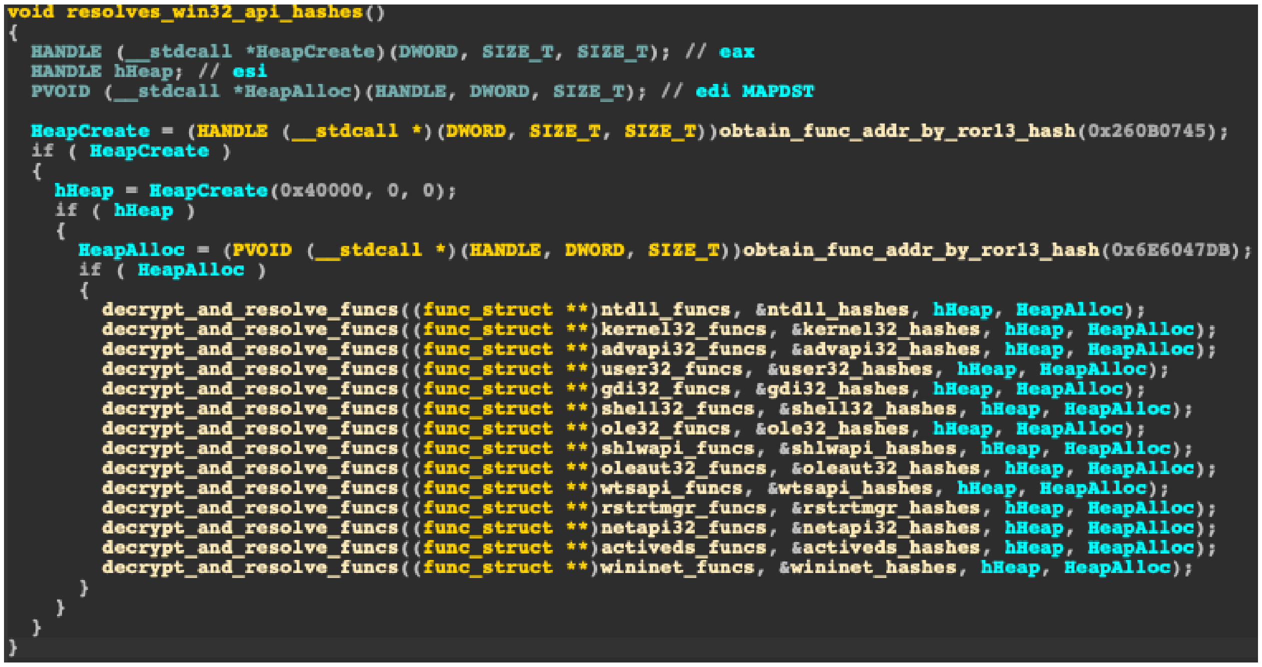


```
      ~+
               *       +
         '     BLACK        |
     ()    .-.,='``'=.    - o -
           '=/_       \     |
        *   |  '=._    |
             \     `=./`,        '
          .   '=.__.=' `='      *
 +             Matter        +
      O      *        '       .

>>> What happens?
   Your network is encrypted, and currently not operational. We have downloaded 1TB from your fileserver.
   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.

>> Data leak includes
1. Full emloyeers personal data
2. Network information
3. Schemes of buildings, active project information, architect details and contracts,
4. Finance info


>>> How to contact with us?
   1. Download and install TOR Browser (https://www.torproject.org/).
   2. Open http://supp24yy6a66hwszu2piygicgwzdtbwftb76htfj7vnip3getgqnzxid.onion/7NT6LXKC1XQHW5039BLOV.

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

Like many ransomwares do, BlackMatter also creates a mutex upon startup to prevent multiple instances from running in parallel. In the upcoming sections, we will discuss several aspects of the ransomware in detail.

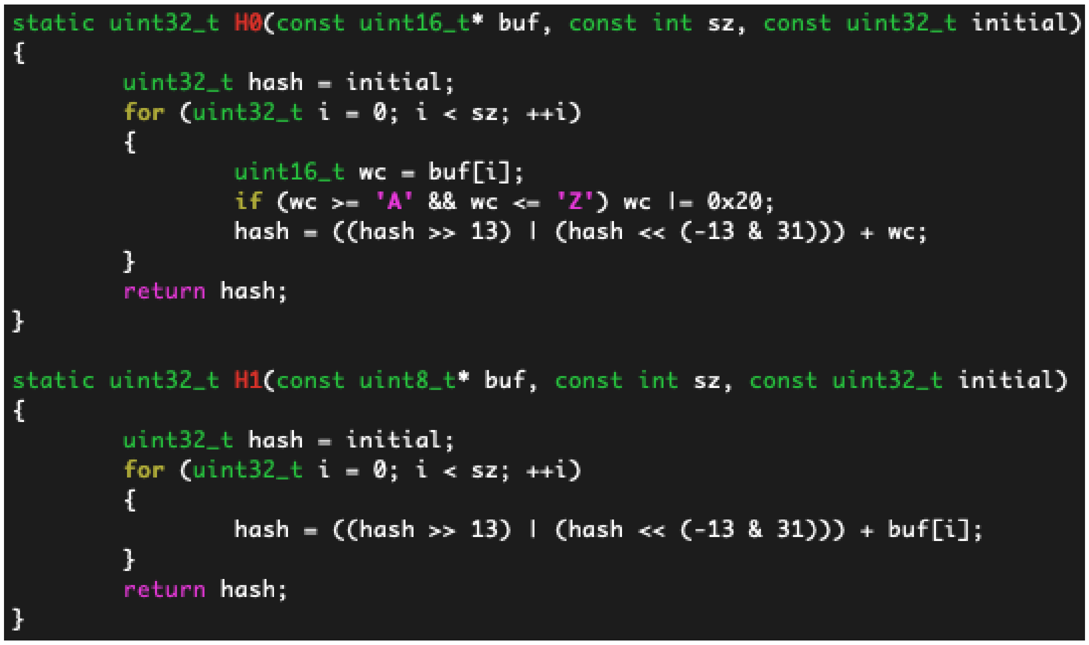
Dynamic Win32 API resolving

A common practice to avoid disclosing information about a malware’s behavior is to resolve Win32 API function addresses at runtime. This way, function names are not visible in the Import Address Table (IAT) of the binary, and static analysis tools cannot gain information from those. BlackMatter also employs this dynamic Win32 API resolving. It resolves the addresses of functions it uses first after starting. In the screenshot below, we see that it first resolves the HeapCreate and HeapAlloc functions, and proceeds to load functions from more than 10 different libraries.



```
void resolves_win32_api_hashes()
{
  HANDLE (__stdcall *HeapCreate)(DWORD, SIZE_T, SIZE_T); // eax
  HANDLE hHeap; // esi
  PVOID (__stdcall *HeapAlloc)(HANDLE, DWORD, SIZE_T); // edi MAPDST

  HeapCreate = (HANDLE (__stdcall *)(DWORD, SIZE_T, SIZE_T))obtain_func_addr_by_ror13_hash(0x260B0745);
  if ( HeapCreate )
  {
    hHeap = HeapCreate(0x40000, 0, 0);
    if ( hHeap )
    {
      HeapAlloc = (PVOID (__stdcall *)(HANDLE, DWORD, SIZE_T))obtain_func_addr_by_ror13_hash(0x6E6047DB);
      if ( HeapAlloc )
      {
        decrypt_and_resolve_funcs((func_struct **)ntdll_funcs, &ntdll_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)kernel32_funcs, &kernel32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)advapi32_funcs, &advapi32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)user32_funcs, &user32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)gdi32_funcs, &gdi32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)shell32_funcs, &shell32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)ole32_funcs, &ole32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)shlwapi_funcs, &shlwapi_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)oleaut32_funcs, &oleaut32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)wtsapi_funcs, &wtsapi_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)rstrtmgr_funcs, &rstrtmgr_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)netapi32_funcs, &netapi32_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)activeds_funcs, &activeds_hashes, hHeap, HeapAlloc);
        decrypt_and_resolve_funcs((func_struct **)wininet_funcs, &wininet_hashes, hHeap, HeapAlloc);
      }
    }
  }
}
```

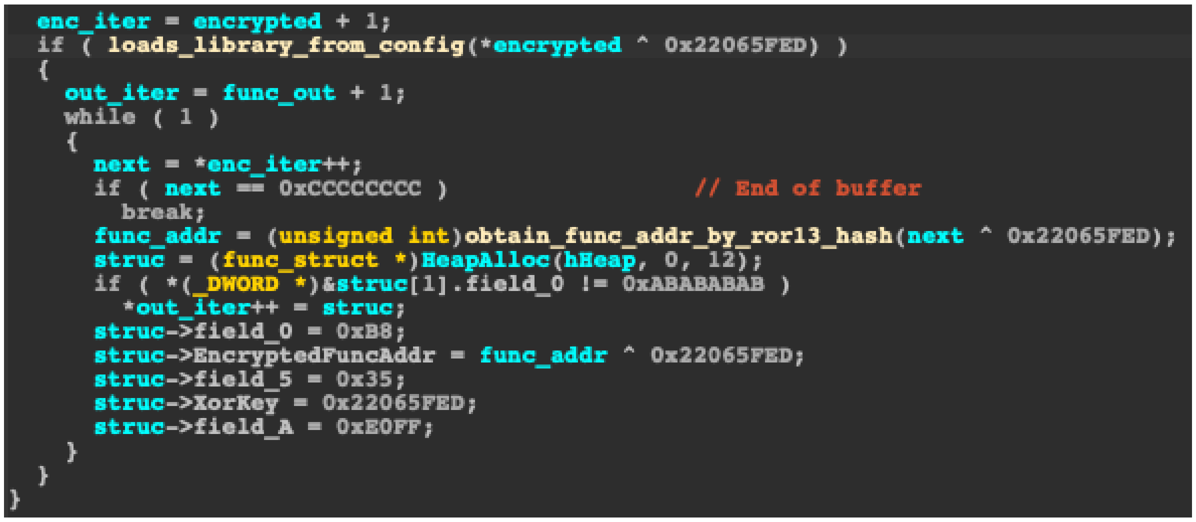
If we zoom in on the 'decrypt_and_resolve_funcs' function, we find that it decrypts and resolves multiple hashes from a given block. BlackMatter contains a block of encrypted hashes for each library it requires functions from, followed by a trailing 0xCCCCCCCC. Each encrypted hash is first decrypted by a 32-bit XOR with key: 0x22065FED. After decryption, the hash is resolved. The hash of a function name is computed using two similar hash functions H_0 and H_1, where H_0 computes the hash of a Unicode library name (including NULL terminator) and H_1 computes the hash of an ASCII function name (including NULL terminator). The two hash functions, both performing ROR-13 operations, are given in the screenshot below.



```
static uint32_t H0(const uint16_t* buf, const int sz, const uint32_t initial)
{
        uint32_t hash = initial;
        for (uint32_t i = 0; i < sz; ++i)
        {
                uint16_t wc = buf[i];
                if (wc >= 'A' && wc <= 'Z') wc |= 0x20;
                hash = ((hash >> 13) | (hash << (-13 & 31))) + wc;
        }
        return hash;
}

static uint32_t H1(const uint8_t* buf, const int sz, const uint32_t initial)
{
        uint32_t hash = initial;
        for (uint32_t i = 0; i < sz; ++i)
        {
                hash = ((hash >> 13) | (hash << (-13 & 31))) + buf[i];
        }
        return hash;
}
```

The hash for a function name is then computed by first calling x = H_0 for the library name. The initial value for this hash is zero. To compute the final hash, H_1 is then called using 'x' as the initial value.

For each Win32 API function it resolves, BlackMatter manually creates a so called ‘call stub’ for the function. This stub is a small block of Assembly code that proxies or relays an API function call. Sometimes a call stub solely jumps to the required API function, but in other cases it performs some additional operations such as decryption or decoding. In the screenshot below, we see that after resolving ‘func_addr’, the call stub is created.



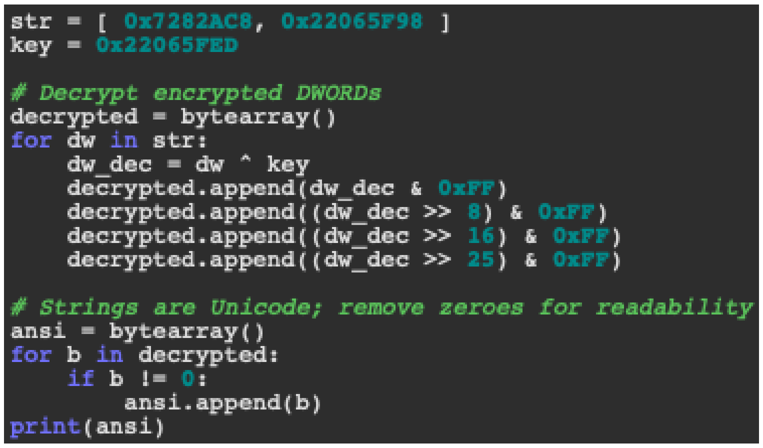
```
  enc_iter = encrypted + 1;
  if ( loads_library_from_config(*encrypted ^ 0x22065FED) )
  {
    out_iter = func_out + 1;
    while ( 1 )
    {
      next = *enc_iter++;
      if ( next == 0xCCCCCCCC )                   // End of buffer
        break;
      func_addr = (unsigned int)obtain_func_addr_by_ror13_hash(next ^ 0x22065FED);
      struc = (func_struct *)HeapAlloc(hHeap, 0, 12);
      if ( *(_DWORD *)&struc[1].field_0 != 0xABABABAB )
        *out_iter++ = struc;
      struc->field_0 = 0xB8;
      struc->EncryptedFuncAddr = func_addr ^ 0x22065FED;
      struc->field_5 = 0x35;
      struc->XorKey = 0x22065FED;
      struc->field_A = 0xE0FF;
    }
  }
}
```

When an API function is called, the call stub is called instead. The actual function address remains encrypted in memory, and the call stub decrypts it before relaying the call. As we can see in the screenshot below, the call stub is quite simple. It decrypts the function address using the same 32-bit XOR we previously encountered. The value 0xDEADBEEF in the screenshot is a placeholder for the actual API function address.

```
0:  b8 ef be ad de          mov    eax,0xdeadbeef
5:  35 ed 5f 06 22          xor    eax,0x22065fed
a:  ff e0                   jmp    eax
```

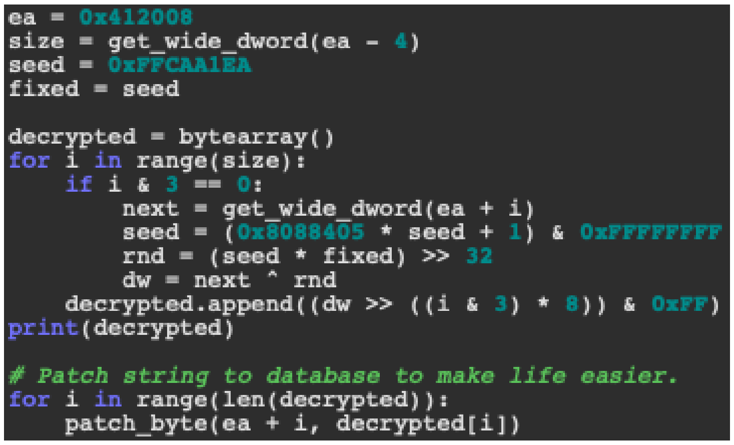
String encryption

String encryption is commonly employed in malware to thwart static analysis. BlackMatter also employs a simple string encryption algorithm, like what was used for API resolving. Throughout the code, encrypted strings are constructed on the stack by DWORDs instead of characters. The resulting buffers are decrypted inline using a simple XOR. The string decryption algorithm is given in the screenshot below.


```
str = [ 0x7282AC8, 0x22065F98 ]
key = 0x22065FED

# Decrypt encrypted DWORDs
decrypted = bytearray()
for dw in str:
    dw_dec = dw ^ key
    decrypted.append(dw_dec & 0xFF)
    decrypted.append((dw_dec >> 8) & 0xFF)
    decrypted.append((dw_dec >> 16) & 0xFF)
    decrypted.append((dw_dec >> 25) & 0xFF)

# Strings are Unicode; remove zeroes for readability
ansi = bytearray()
for b in decrypted:
    if b != 0:
        ansi.append(b)
print(ansi)
```

Obtaining the configuration

BlackMatter also contains an embedded configuration, and it is encoded in a similar way DarkSide encoded theirs. In this case, the configuration is located in the ".rsrc" segment. The configuration starts with a 32-bit initial state value for the decryption algorithm. In this case, this value is 0xFFCAA1EA. A DWORD indicating the size is next, followed by the encrypted configuration. The decryption algorithm is shown in the screenshot below and is based on a slightly customized linear congruential generator.


```
ea = 0x412008
size = get_wide_dword(ea - 4)
seed = 0xFFCAA1EA
fixed = seed

decrypted = bytearray()
for i in range(size):
    if i & 3 == 0:
        next = get_wide_dword(ea + i)
        seed = (0x8088405 * seed + 1) & 0xFFFFFFFF
        rnd = (seed * fixed) >> 32
        dw = next ^ rnd
    decrypted.append((dw >> ((i & 3) * 8)) & 0xFF)
print(decrypted)

# Patch string to database to make life easier.
for i in range(len(decrypted)):
    patch_byte(ea + i, decrypted[i])
```
The decrypted content is then decompressed using the aPlib algorithm. The decrypted and decompressed configuration starts with an RSA-1024 public key. Furthermore, the configuration includes a few Base64-encoded strings that contain:

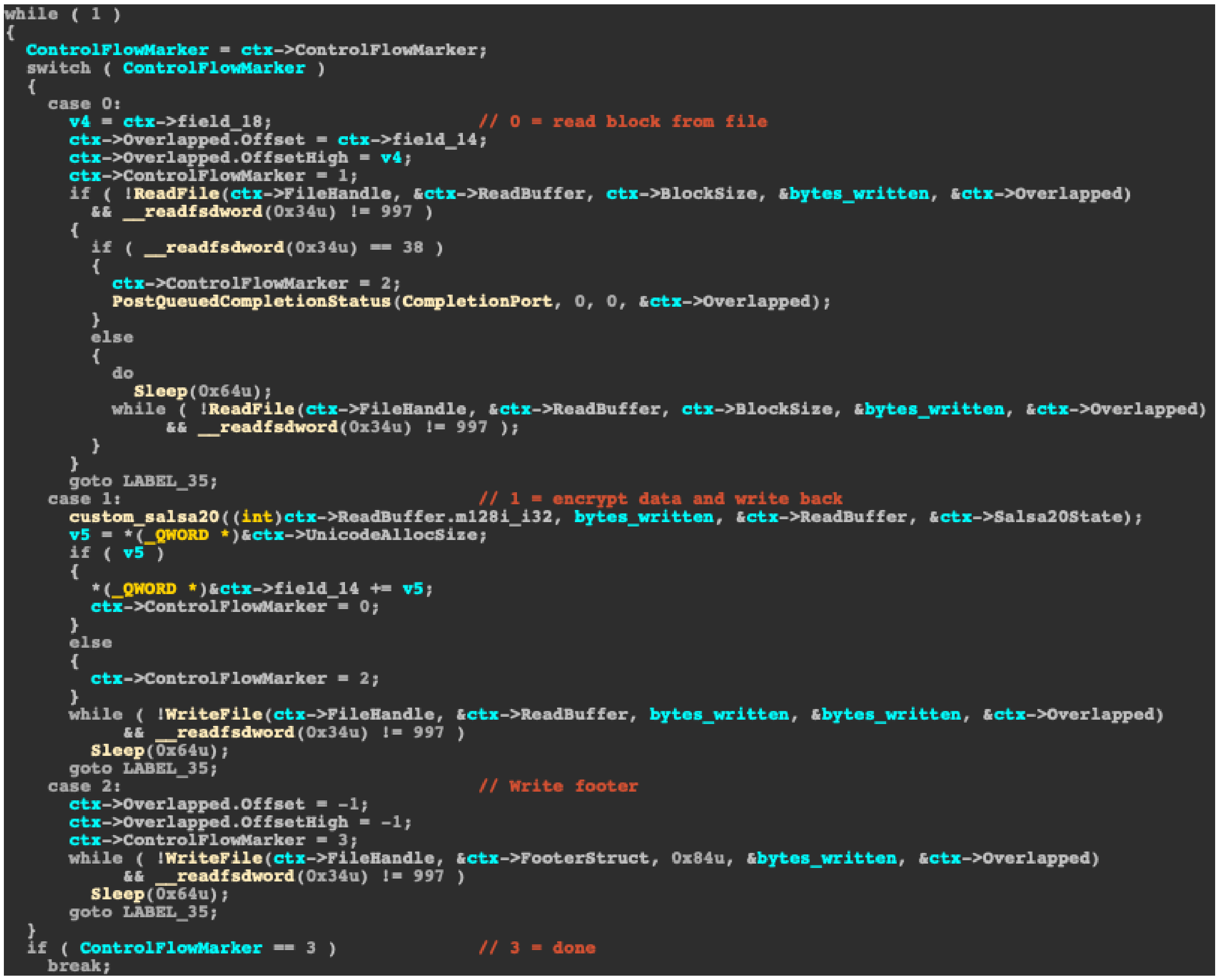
- Command & control hostnames
- AES-128 encryption key for command & control
- Names of services to kill
- File and directories to avoid
- Ransom note (also encrypted with the algorithm above)

File encryption

Naturally, we want to know how the file encryption process works in the BlackMatter ransomware. Like DarkSide, the BlackMatter ransomware first kills blacklisted processes and services, empties the recycle bins for all drives, and deletes shadow copies. It then finds files on the filesystem to encrypt.

The file encryption process is set up using I/O completion ports. These completion ports provide an efficient model for handling many concurrent I/O operations on a system with multiple CPUs. An I/O completion port is created with a given number of threads using the CreateIoCompletionPort function. After creation, a queue is created internally, to which threads can push completion statuses. One can queue a status using the PostQueuedCompletionStatus function, and a thread can pull this status from the queue using the GetQueuedCompletionStatus function.

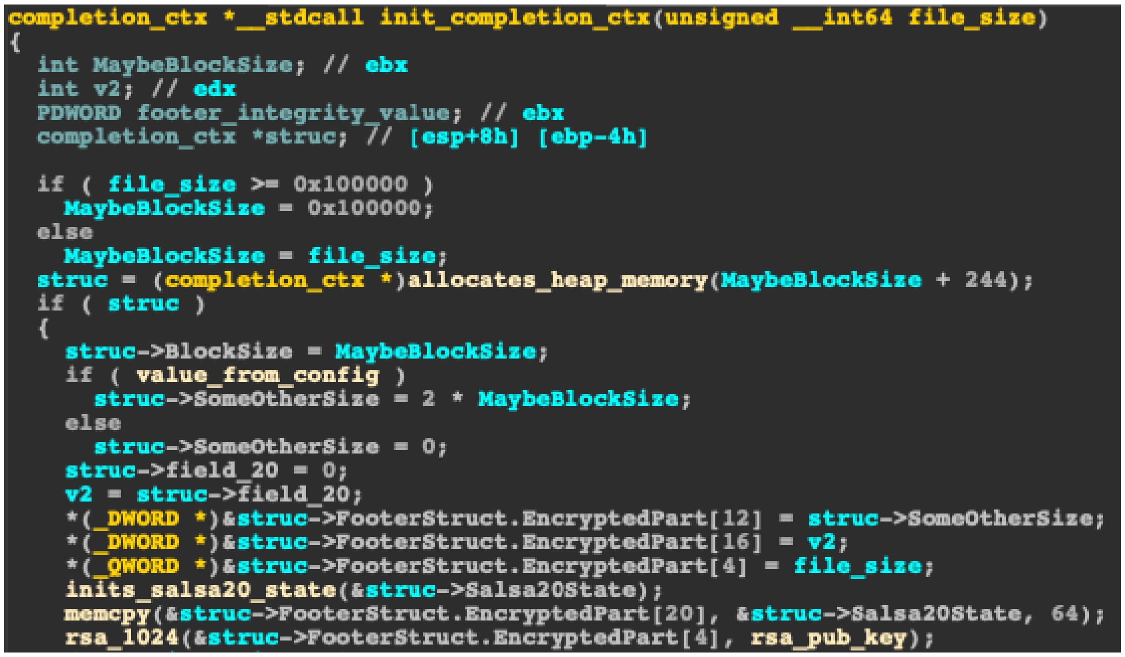
A user-defined data context structure can be attached to a status to keep track of variables during encryption if its first member is an OVERLAPPED structure. Using the completion status queue, a control flow of I/O actions can be constructed. The file encryption function in the BlackMatter ransomware uses the state numbers 0 to 3 to create a control flow. This control flow is shown in the screenshot below.



```
while ( 1 )
{
  ControlFlowMarker = ctx->ControlFlowMarker;
  switch ( ControlFlowMarker )
  {
    case 0:
      v4 = ctx->field_18;                   // 0 = read block from file
      ctx->Overlapped.Offset = ctx->field_14;
      ctx->Overlapped.OffsetHigh = v4;
      ctx->ControlFlowMarker = 1;
      if ( !ReadFile(ctx->FileHandle, &ctx->ReadBuffer, ctx->BlockSize, &bytes_written, &ctx->Overlapped)
        && __readfsdword(0x34u) != 997 )
      {
        if ( __readfsdword(0x34u) == 38 )
        {
          ctx->ControlFlowMarker = 2;
          PostQueuedCompletionStatus(CompletionPort, 0, 0, &ctx->Overlapped);
        }
        else
        {
          do
            Sleep(0x64u);
          while ( !ReadFile(ctx->FileHandle, &ctx->ReadBuffer, ctx->BlockSize, &bytes_written, &ctx->Overlapped)
               && __readfsdword(0x34u) != 997 );
        }
      }
      goto LABEL_35;
    case 1:                                 // 1 = encrypt data and write back
      custom_salsa20((int)ctx->ReadBuffer.m128i_i32, bytes_written, &ctx->ReadBuffer, &ctx->Salsa20State);
      v5 = *(_QWORD *)&ctx->UnicodeAllocSize;
      if ( v5 )
      {
        *(_QWORD *)&ctx->field_14 += v5;
        ctx->ControlFlowMarker = 0;
      }
      else
      {
        ctx->ControlFlowMarker = 2;
      }
      while ( !WriteFile(ctx->FileHandle, &ctx->ReadBuffer, bytes_written, &bytes_written, &ctx->Overlapped)
          && __readfsdword(0x34u) != 997 )
        Sleep(0x64u);
      goto LABEL_35;
    case 2:                                 // Write footer
      ctx->Overlapped.Offset = -1;
      ctx->Overlapped.OffsetHigh = -1;
      ctx->ControlFlowMarker = 3;
      while ( !WriteFile(ctx->FileHandle, &ctx->FooterStruct, 0x84u, &bytes_written, &ctx->Overlapped)
          && __readfsdword(0x34u) != 997 )
        Sleep(0x64u);
      goto LABEL_35;
  }
  if ( ControlFlowMarker == 3 )             // 3 = done
    break;
```

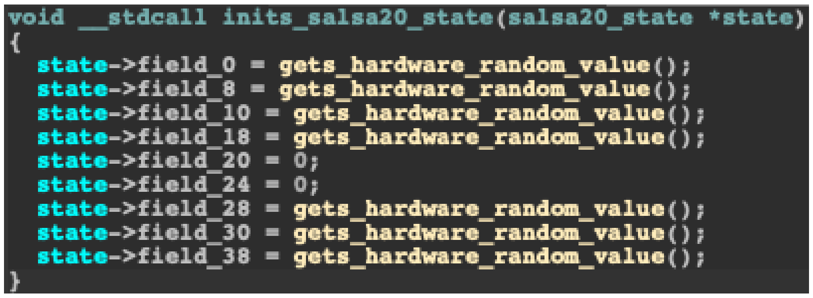
As we can see, the encryption starts with a zero: read the next data block from the file. The status then switches to 1: encrypting the block of data previously read and writing it back to the file. The file footer is written when the status is set to 2, and when all I/O tasks have been completed, the encryption is finished with status number 3.

The user-defined context structure is created in the screenshot below. It contains for example the number of bytes to encrypt in a file. If the file is smaller than 1MiB, the entire file is encrypted. If the file is larger, the first 1MiB is encrypted.



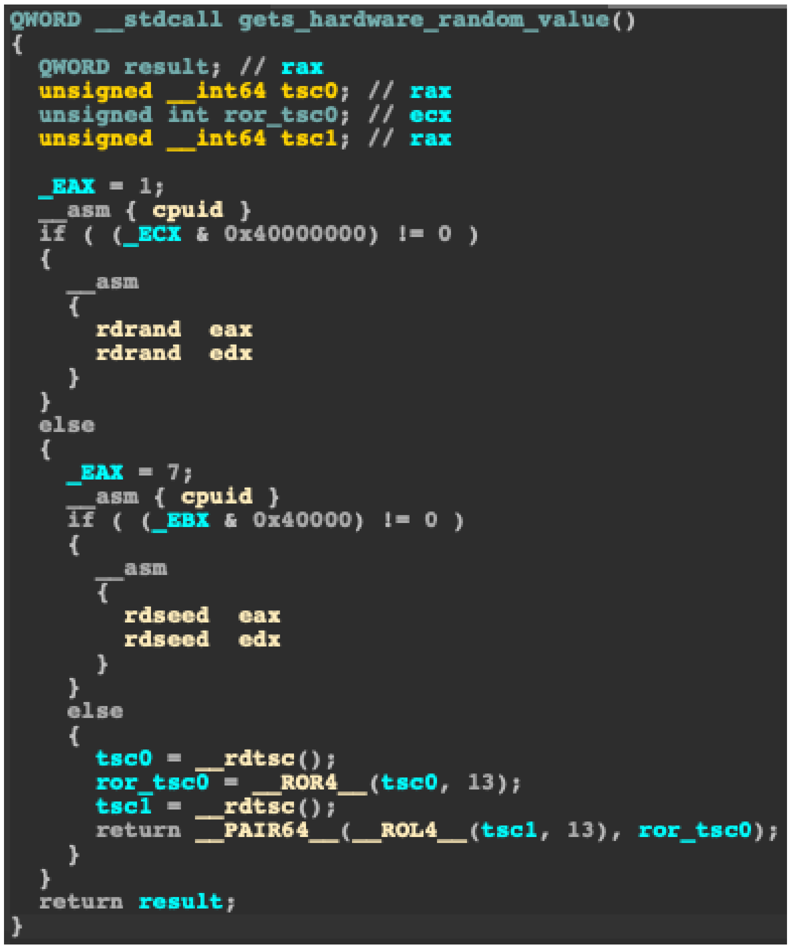
```
completion_ctx *__stdcall init_completion_ctx(unsigned __int64 file_size)
{
  int MaybeBlockSize; // ebx
  int v2; // edx
  PDWORD footer_integrity_value; // ebx
  completion_ctx *struc; // [esp+8h] [ebp-4h]

  if ( file_size >= 0x100000 )
    MaybeBlockSize = 0x100000;
  else
    MaybeBlockSize = file_size;
  struc = (completion_ctx *)allocates_heap_memory(MaybeBlockSize + 244);
  if ( struc )
  {
    struc->BlockSize = MaybeBlockSize;
    if ( value_from_config )
      struc->SomeOtherSize = 2 * MaybeBlockSize;
    else
      struc->SomeOtherSize = 0;
    struc->field_20 = 0;
    v2 = struc->field_20;
    *(_DWORD *)&struc->FooterStruct.EncryptedPart[12] = struc->SomeOtherSize;
    *(_DWORD *)&struc->FooterStruct.EncryptedPart[16] = v2;
    *(_QWORD *)&struc->FooterStruct.EncryptedPart[4] = file_size;
    inits_salsa20_state(&struc->Salsa20State);
    memcpy(&struc->FooterStruct.EncryptedPart[20], &struc->Salsa20State, 64);
    rsa_1024(&struc->FooterStruct.EncryptedPart[4], rsa_pub_key);
```

Files are encrypted using the Salsa20 stream cipher algorithm, and the corresponding keys are encrypted using RSA-1024. If we zoom in on the 'inits_salsa20_state' function, we find that the Salsa20 matrix is initialized at random without a key and nonce. In the screenshot below, the matrix is constructed using 8-byte random values. The nonce is left at zero.



```
void __stdcall inits_salsa20_state(salsa20_state *state)
{
  state->field_0 = gets_hardware_random_value();
  state->field_8 = gets_hardware_random_value();
  state->field_10 = gets_hardware_random_value();
  state->field_18 = gets_hardware_random_value();
  state->field_20 = 0;
  state->field_24 = 0;
  state->field_28 = gets_hardware_random_value();
  state->field_30 = gets_hardware_random_value();
  state->field_38 = gets_hardware_random_value();
}
```

BlackMatter first checks whether the CPU inside the victim's system supports the RDRAND instruction. If so, the next random number is generated by combining two calls to this instruction. If the victim's CPU does not support RDRAND, the CPU timestamp counter is used instead. The next random number is then constructed by combining the lower 32 bits of two timestamp counters. One is rolled 13 bits to the left, and one 13 bits to the right. The screenshot below shows how these random numbers are generated.



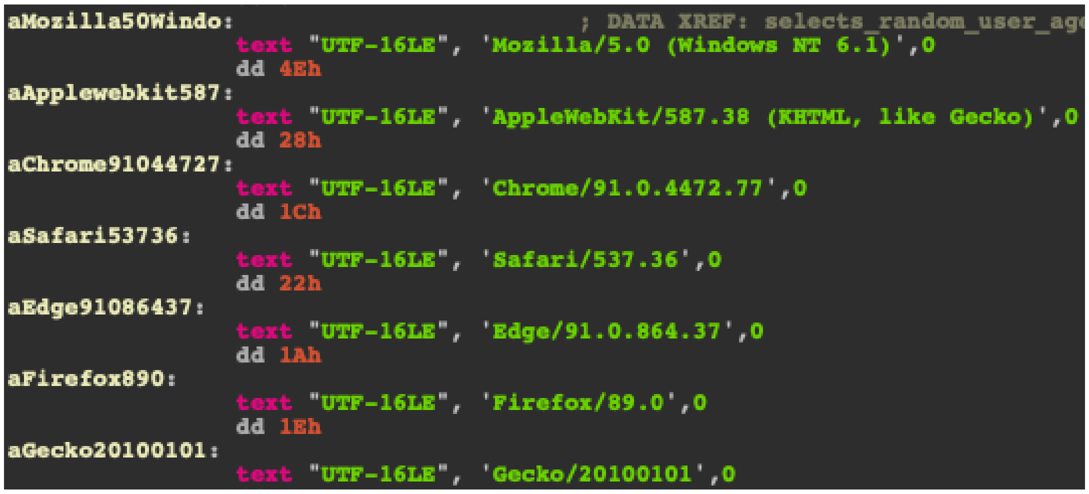
```
QWORD __stdcall gets_hardware_random_value()
{
  QWORD result; // rax
  unsigned __int64 tsc0; // rax
  unsigned int ror_tsc0; // ecx
  unsigned __int64 tsc1; // rax

  _EAX = 1;
  __asm { cpuid }
  if ( (_ECX & 0x40000000) != 0 )
  {
    __asm
    {
      rdrand  eax
      rdrand  edx
    }
  }
  else
  {
    _EAX = 7;
    __asm { cpuid }
    if ( (_EBX & 0x40000) != 0 )
    {
      __asm
      {
        rdseed  eax
        rdseed  edx
      }
    }
    else
    {
      tsc0 = __rdtsc();
      ror_tsc0 = __ROR4__(tsc0, 13);
      tsc1 = __rdtsc();
      return __PAIR64__(__ROL4__(tsc1, 13), ror_tsc0);
    }
  }
  return result;
}
```

The random Salsa20 matrix is encrypted with the RSA-1024 public key included in the configuration and written to the footer in the encrypted file. The footer contains some more information about the file, such as its original size and a magic value that allows BlackMatter to identify already encrypted files. The file encryption process is also quite like DarkSide's.

Command & control

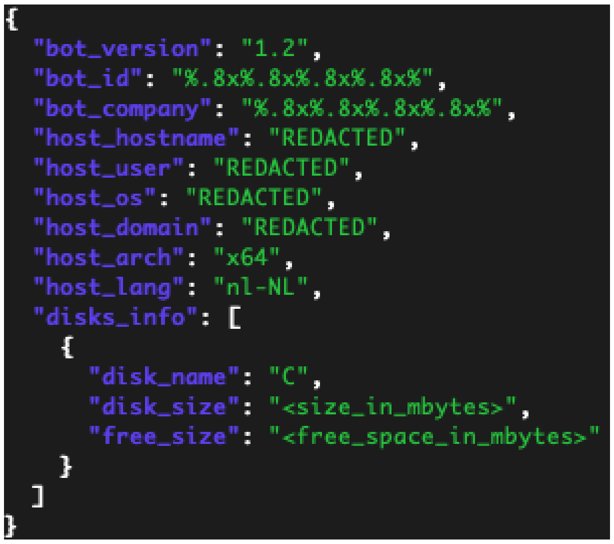
BlackMatter contacts command & control hosts to collect information about the victim. The hostnames it contacts are stored in the embedded configuration we previously discussed. BlackMatter first attempts to contact the command & control hosts using HTTPS and falls back to HTTP if it fails. The sent HTTP POST requests target a randomly generated URI consisting of key/value pairs with characters from the Base64 alphabet. The HTTP data associated to the requests also consists of similar key/value pairs. The difference is: one of the pairs contains encrypted data.

The HTTP requests are facilitated by the WinInet library, including functions such as InternetOpenW, InternetConnectW and HttpSendRequestW. An interesting characteristic of the requests is the User-Agent header. BlackMatter contains multiple hardcoded user agents, from which it randomly selects one to use in its connections. The user agents are shorter than the default ones, as we can see in the screenshot below.



```
aMozilla50Windo:                          ; DATA XREF: selects_random_user_ag
                  text "UTF-16LE", 'Mozilla/5.0 (Windows NT 6.1)',0
                  dd 4Eh
aApplewebkit587:
                  text "UTF-16LE", 'AppleWebKit/587.38 (KHTML, like Gecko)',0
                  dd 28h
aChrome91044727:
                  text "UTF-16LE", 'Chrome/91.0.4472.77',0
                  dd 1Ch
aSafari53736:
                  text "UTF-16LE", 'Safari/537.36',0
                  dd 22h
aEdge91086437:
                  text "UTF-16LE", 'Edge/91.0.864.37',0
                  dd 1Ah
aFirefox890:
                  text "UTF-16LE", 'Firefox/89.0',0
                  dd 1Eh
aGecko20100101:
                  text "UTF-16LE", 'Gecko/20100101',0
```

As we previously mentioned, the encrypted victim information is placed in one of the key/value pairs in the HTTP POST data. The position of the pair in the string is randomly determined. We can determine which pair to decrypt by Base64-decoding each pair in the data string and checking the size of the results. We attempt decryption on all pairs having a size aligned to a 16-byte boundary. The encryption algorithm used here is AES-128 in ECB mode (independent block-by-block encryption). The key is placed in the configuration, starting 16 bytes after the RSA-1024 public key. In the sample we analyzed, the key is (hex-encoded): A6F330B09CD47B4FB9214F7836AA46AD. After decrypting, we get the following JSON object.



```
{
  "bot_version": "1.2",
  "bot_id": "%.8x%.8x%.8x%.8x%",
  "bot_company": "%.8x%.8x%.8x%.8x%",
  "host_hostname": "REDACTED",
  "host_user": "REDACTED",
  "host_os": "REDACTED",
  "host_domain": "REDACTED",
  "host_arch": "x64",
  "host_lang": "nl-NL",
  "disks_info": [
    {
      "disk_name": "C",
      "disk_size": "<size_in_mbytes>",
      "free_size": "<free_space_in_mbytes>"
    }
  ]
}
```

As we can see, we analyzed version 1.2 of BlackMatter. The object contains host and user information, such as identifiers for the victim, computer name and username, and available disk space on the infected system. The information in the JSON object is similar to what DarkSide sent to their command & control hosts, as they also collected system, user and disk information.

Conclusion

In this blog, we presented a detailed analysis of the BlackMatter ransomware. Others suggested that BlackMatter is a rebrand of the DarkSide ransomware, and the code similarities we found support this suggestion. The sample is available for download at MalwareBazaar.

Indicators of Compromise (IoCs)

Indicator	Description
22d7d67c3af10b1a37f277ebabe2d1eb4fd25afbd6437d4377400e148bcc08d6	BlackMatter ransomware
paymenthacks[.]com	C2
mojobiden[.]com	C2