Conti Ransomware v2

chuongdong.com/reverse engineering/2020/12/15/ContiRansomware/

Chuong Dong December 15, 2020

Reverse Engineering · 15 Dec 2020

Overview

This is my full analysis for the Conti Ransomware version 2. Over the last few months, I have seen quite a few companies getting hit by this ransomware, so it’s been interesting analyzing and figuring how it works.

As one of the newer ransomware families, Conti utilizes multi-threading features on Windows to encrypt files on machines To the fullest extent, making itself a lot faster than most ransomware out there.

From the analysis, it’s clear that Conti is designed to target and encrypt business environments that uses SMB for file sharing and other services. Similar to the Sodinokibi family, Conti has the ability to scan existing ports and SMB shares on the network to spread its encryption, which can be a lot more impactful since it is not limited to the local machine.

By the time this blog post comes out, researchers have found newer samples of the version 3. Even though this is an old sample, I still think it’s beneficial to provide the community with a deeper understanding about this malware.

alt text

Figure 1: Conti overview

IOCS

Conti Ransomware version 2 comes in the form of a 32-bit PE file (either **.exe** or **.dll**).

MD5: 0a49ed1c5419bb9752821d856f7ce4ff

SHA256: 03b9c7a3b73f15dfc2dcb0b74f3e971fdda7d1d1e2010c6d1861043f90a2fecd

Sample:
https://bazaar.abuse.ch/sample/03b9c7a3b73f15dfc2dcb0b74f3e971fdda7d1d1e2010c6d1861043f90a2fecd/

Unpacked sample:
https://bazaar.abuse.ch/sample/d3c75c5bc4ae087d547bd722bd84478ee6baf8c3355b930f26cc19777cd39d4c/

alt text

Figure 2: VirusTotal result

Ransom Note

alt text

Figure 3: Conti Ransom Note

The ID appended at the end is actually hard-coded, so it’s not a victim’s ID. This ID is most likely just the ID of this particular Conti sample.

Below is the HTTPS version of the website for recovery service.

alt text

Figure 4: Conti Website

Dependencies

The ransomware only has ***Kernel32.dll, User32.dll, and WS2_32.dll*** as visible imported DLLs.

However, it does dynamically resolve a lot of DLLs through decrypting stack strings and calling **LoadLibrary** as seen here.

alt text

Figure 5: Conti resolving DLL string names (sub_571010)

Here is the full list of the imported DLLs.

- **Kernel32.dll**
- **Ntdll.dll**
- **Ole32.dll**
- **Shell32.dll**
- **Ws2_32.dll**
- **Shlwapi.dll**
- **Advapi32.dll**
- **Iphlpapi.dll**
- **Rstrtmgr.dll**
- **Netapi32.dll**
- **OleAut32_dll**
- **User32.dll**

PE Layout

The unpacked version of the malware is around 208 KB in size, which consists of the **.text, .rdata, .data, .rsrc, and .reloc** sections.

One of the main reasons why this executable is so big is because of the obsfucation method the developer uses. Instead of implementing a single string decryption function, they used one decrypting for loop for each encrypted string, which greatly increased the amount of raw code.

alt text

Figure 6: Conti Layout

Code Analysis

String Decryption

As mentioned above, Conti uses the method of building up a stack “string” that is encrypted and proceeds to decrypt it with a for loop. Every string is encrypted differently, so the for loop changes slightly for each of them.

alt text

Figure 7: String decryption of explorer.exe (sub_58B2D0)

Most of the decryption loops can be simplified to this single form where **buffer** is the encrypted string, **a** and **b** are positive numbers, and **c** is either **1 or -1**.

```
for i in range(len(buffer)):
    buffer[i] = (a * (c * (buffer[i] - b)) % 127 + 127) % 127
```

Dynamically Resolve API

When resolving APIs, Conti calls a particular function that takes in an integer representing the DLL to find, an API hash value, and an offset into the API buffer.

The DLL name is retrieved from the given integer through a switch statement.

- 15 ==> Kernel32.dll
- 16 ==> Ws2_32.dll
- 17 ==> Netapi32.dll
- 18 ==> Iphlpapi.dll
- 19 ==> Rstrtmgr.dll
- 20 ==> User32.dll
- 21 ==> Ws2_32.dll
- 22 ==> Shlwapi.dll
- 23 ==> Shell32.dll
- 24 ==> Ole32.dll
- 25 ==> OleAut32.dll
- 26 ==> Ntdll.dll

After getting the DLL name, Conti will manually locate the export directory of that DLL, loop through each API, hash the name, and compare it with the hash from the parameter. After finding the correct API with the right hash value, it will proceed to find the address to that function.

alt text

Figure 8: Function looping through export table and hash API name (sub_5737C0)

For the hashing algorithm, the constant **0x5BD1E995** gives this away that this is Murmur Hash

alt text

Figure 9: Conti’s Murmur Hashing implementation (sub_575970)

After finding the address of the API, the malware adds that into its API array at the provided offset. This helps reducing the time to look up an API’s address if the malware has already resolved it before.

Run-once Mutex

Conti attempts to decrypt the string **“jkbmusop9iqkamvcrewuyy777”** and use that as the name of a Mutex object.

Then, it checks if there is an instant of that Mutex running already. If there is, it will just wait until that thread exits before exiting.

alt text

Figure 10: Checking for Mutex (sub_587E20)

Command-line Arguments

Conti can only be ran with command-line arguments, so it must be launched by a loader. Upon execution, it will process these arguments and behave accordingly.

CMD Args	Functionality
-m local	Encrypting the local machine’s hard drive with multiple threads
-m net	Encrypting network shares via SMB with mutiple threads
-m all	Encrypting both locally and on the network with multiple threads
-p [directory]	Encrypt a specific directory locally with 1 thread
-size [chunk mode]	Large encryption chunk mode
-log [file name]	Logging mode. Log everything to the file with the given name
backups	Unimplemented for some reason

Encryption

Despite having 3 different encrypting schemes, the main mechanism is relatively the same.

First, it calls a function to populate a structure used to initialize information about the thread/threads of that encrypting schemes. These information includes the number of threads to spawn and a thread buffer that is used to store thread **HANDLE** objects.

alt text

Figure 11: Function initializing thread struct (sub_58BDB0)

Next, it calls this function to launch child threads. It checks the thread struct to see if the encrypting flag is set. If it is, loop from **0 to thread_count - 1** and spawn a thread to encrypt each time. It also adds these threads into the thread buffer for easy clean-up later.

alt text

Figure 12: Function launching encrypting threads (sub_58BE30)

Multi-threading

Beside when the argument **-p** is provided, multi-threading is involved for every other scheme of encryption. Conti will call **GetNativeSystemInfo** to retrieve information about the running system.

If the argument **“-m all”** is provided, the number of threads to spawn will be double the amount of processors because it needs to encrypt both locally and on the network.

For everything else, the number of threads to spawn is the same as the number of processors.

alt text

Figure 13: Determining how many threads to spawn from number of processor (sub_587E20)

Being able to thread its encryption, Conti utilizes all of the CPU threads available to simultaneously go through and encrypt the file system with incredible speed.

alt text

Figure 14: Realistic representation of what happens when Conti runs

The most interesting information in the thread structure is the string of the path to be encrypted. After having launch the threads, Conti’s main program will continuously traverse the file system and provide the thread structure with directory names. All of these threads will check this information and encrypt the updated path immediately. Because the workload is divided efficiently, Conti is able to speed up its traversing and encryption to a great extent.

alt text

Figure 15: Main thread providing the drives to be encrypted (sub_587E20)

Encrypting Locally

RSA Public Key

First, each thread will call **CryptAcquireContextA** with the cryptographic provider type **PROV_RSA_AES** to retrieve a handle of a CSP for RSA encryption. Using that CSP, it will call **CryptImportKey** to import from the hard-coded RSA public key.

alt text

Figure 16: RSA Public Key embedded in the .data section

alt text

Figure 17: CryptAcquireContextA and CryptImportKey called (sub_58BC20)

Next, it will enter an infinite loop to wait for the main thread to add a target drive path or to send a stop signal. This is accomplished solely through the shared thread struct that was created before launching these threads. Because the struct is shared between multiple threads, calls to **EnterCriticalSection** and **LeaveCriticalSection** are critical to maintain a thread-safe environment during encryption.

alt text

Figure 18: Each thread continuously polling for a path name and encrypt it (sub_58BC20)

In the main encrypting function, it will iteratively call **FindFirstFile** on the directory name to search for all files and folders inside, avoiding the two current path and parent path names **”.”** and **”..”** which can cause an infinite loop if processed.

Directory Check

If the file being checked is a directory, it will check to see if the directory name is valid or not. If it is, then the child thread will add that path to the thread struct for itself or any other available thread to encrypt.

alt text

Figure 19: Checking if the path is a valid directory (sub_586340)

These are the directory name that Conti will avoid encrypting.

```
tmp, winnt, temp, thumb, $Recycle.Bin, $RECYCLE.BIN, System Volume Information, Boot, Windows,
Trend Micro
```

Normal File Check

If the file is just a normal file, Conti will check to see if the file name is valid before proceed to encrypt it.

alt text

Figure 20: Checking if the path is a valid file (sub_586340)

Conti will avoid encrypting any file with these names or extensions.

```
CONTI_LOG.txt, readme.txt, .msi, .sys, .lnk, .dll, .exe
```

Normal File Encryption

First, Conti populates a structure in memory. I call this structure **CONTI_STRUCT**.

```
struct CONTI_STRUCT
{
  char *file_name;
  HANDLE hFile;
  LARGE_INTEGER file_size;
  int CHACHA8_const[4];
  int CHACHA8_256_KEY[8];
  int block_counter;
  int block_counter_ptr;
  int CHACHA8_none[2];
  int random1[2];
  int random2[8];
  BYTE encrypted_key[524]; // encrypted ChaCha8 key
};
```

Conti will call **CryptGenRandom** to generate 2 different random buffers and put them into the **CONTI_STRUCT**. Then, it populates the ChaCha8 constants which is just **“expand 32-byte k”** in hex form.

The first buffer is 256 bits, which is later used as the **ChaCha8** encrypting key, and the second one is 64 bits, which is used as the **ChaCha8** nonce.

Next, it will copy the key and nonce into the buffer at the end of the struct and encrypt it using the RSA key imported earlier. This is to ensure that the ChaCha key can not be recovered without the RSA private key.

alt text

Figure 21: Generating random number (sub_5805A0)

alt text

Figure 22: Populating ChaCha8 constants and encrypt the random numbers with the RSA key (sub_5805A0)

Conti has 3 file categories for encryption - small, medium, and large files. Small files are marked with the value of **0x24**, medium with **0x26**, and large with **0x25**.

Before encryption, Conti will write the encrypted ChaCha8 key from **CONTI_STRUCT**, this mark, and the file size to at the end of the to-be-encrypted file.

alt text

Figure 23: Writing the encrypted random ChaCha8 key, mark, and size to file (sub_57E4B0)

alt text

Figure 24: The key at the end of an encrypted file

1. Small File

Small files are files that are potentially less than 1MB in size. Conti looks for all files that is smaller than 1MB or by checking for these extensions below.

```
.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, .mud,
.mwb, .myd, .ndf, .nnt, .nrmlib, .ns2, .ns3, .ns4, .nsf, .nv, .nv2, .nwdb, .nyf, .odb, .oqy, .orx,
.owc,
.p96, .p97, .pan, .pdb, .pdm, .pnz, .qry, .qvd, .rbf, .rctd, .rod, .rodx, .rpd, .rsd, .sas7bdat,
.sbf, .scx,
.sdb, .sdc, .sdf, .sis, .spq, .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
```

Encrypting small files are straightforward. Since these files are small enough, it typically does not require to loop and encrypt more than once. The file content is read into a buffer and encrypted directly. Just to be safe, the malware author did limit the maximum buffer size to read to 5MB, but it’s unlikely that the files going into this function is that big.

alt text

Figure 25: Small File Encrypting mechanism (sub_580460)

2. Medium File

Medium files are files that are between 1MB to 5MB.

For these files, Conti only encrypts the first 1 MB of the files.

alt text

Figure 26: Medium File Encrypting mechanism (sub_5805A0)

3. Large file

Large files are files that are larger than 5MB. Conti specifically looks for these by checking for these extensions.

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

The large file encrypting function processes the **-size** chunk mode argument and uses it in a switch statement to determine the encrypting offset and the encrypting size.

According to Michael Gillespie, here are the chunk mode values:

- 0x14 (default) ==> represent 3 chunks of (file_size / 100 * 7)
- 0x32 ==> represent 5 chunks of (file_size / 100 * 10)

The mechanism of encrypting can be simplify to this. Basically, Conti will encrypt ***encrypt_length*** amount of bytes and skip the next ***encrypt_offset*** before encrypting again until it reaches the end of the file. This makes encryption quicker for large files because it does not have to encrypt everything.

Also according to Michael, Conti has a bug where the keystream sometime goes out of sync in-between chunks during encryption because the encrypted buffer size is rounded up to the nearest 64 which is the ChaCha state matrix size.

alt text

Figure 27: Large File Encrypting mechanism (sub_57FFD0)

4. ChaCha8 Encryption

The **ChaCha8** implementation is pretty straightforward. The 256-byte key that was randomly generated earlier is then used as the encrypting key.

alt text

Figure 28: Conti's ChaCha8 implementation (sub_575AC0)

In order to be able to decrypt the files, we need to know the random key that Conti uses for each file, and the only way to retrieve it is through the encrypted key buffer at the end of the file.

Since that buffer is encrypted with a public RSA key, we need the private RSA key to decrypt this.

Nonetheless, since they are using a hard-coded public key, if anyone pays the ransom for this Conti version, the private key can be retrieved. It will be simple to write a decrypting tool if that is the case, and all of the samples with this ID will become useless after.

This implementation clearly reflects how the Conti group mainly targets big companies instead of aiming to spread the malware to normal computer users. Once a company (or anyone) pays off the ransom, they have to discard all of the samples that use the private key and develop newer samples to spread.

alt text

Figure 29: Conti's Encryption method

Delete Shadow Copy with COM Objects

Before encrypting, Conti's main thread calls **CoInitializeEx, CoInitializeSecurity, and CoCreateInstance** to creates a single object of the class **IWbemLocator** with the specified CLSID *4590F811-1D3A-11D0-891F-00AA004B2E24*.

alt text

Figure 30: Initializing COM Object (sub_576B80)

Next, it checks if the processor architecture of the machine is **x86-64** . If it is, then Conti will call **CoCreateInstance** to create a single object of the class **IWbemContext** with the specified CLSID *674B6698-EE92-11D0-AD71-00C04FD8FDFF*.

With this **Call Context** object, it can modify the **__ProviderArchitecture** to force load the specified provider version which is 64-bit architecture.

alt text

Figure 31: Force load 64-bit if needed (sub_576B80)

Using the **IWbemLocator** object earlier, Conti calls its **ConnectServer** method to connect with the local *ROOT\CIMV2* namespace and obtain the pointer to an **IWbemServices** object.

alt text

Figure 32: Connecting to ROOT\CIMV2 to get IWbemServices object (sub_576B80)

With this IWbemServices object, it executes the SQL query **"SELECT * FROM Win32_ShadowCopy"** to retrieve a enumerator of all the shadow copies stored in the local server.

By enumerating through these informations, Conti extracts the ID of each shadow copy, add that to the format string **"cmd.exe /c C:\Windows\System32\wbem\WMIC.exe shadowcopy where "ID='%s'" delete"**, and create a new process to execute. This will eventually deletes all the shadow copy storage areas in the computer.

alt text

Figure 33: Building cmd string to delete shadowcopy based on ID (sub_576B80)

Network Encryption

For the network encryption, Conti calls **CreateIoCompletionPort** to spawn as many concurrently running threads as there are processors in the system, and these threads waits for a list of network shares to start encryption.

alt text

Figure 34: CreateIoCompletionPort to spawn network encrypting thread (sub_58A6F0)

The main thread then calls **NetShareEnum** to get an enumerator to extract information about shared network resources. This scans the system to see if there exists any existing SMB network shares.

After getting this “ARP” cache, it will check if the IP addresses of hosts in the list start with **“172.”, “192.168.”, “10.”, and “169.”**. Since it only cares about encrypting local systems, any other IP address ranges is ignored.

It will then scan and look for every shares with the name that is not **“ADMIN$”**, get the full path to the shares, and add it to an array of network shares.

alt text

Figure 35: Scanning SMB for all existing SMB network shares (sub_5898D0)

After extracting this, it will loop through and call the function from *Figure 15* to push these share names into the thread struct so the child threads can begin encrypting.

If scanning SMB for network hosts fails, Conti will perform just a port scan using **CreateIoCompletionPort GetQueuedCompletionStatus, and PostQueuedCompletionStatus**

alt text

Figure 36: Conti port scans (sub_58A370)

After this point, the encryption happens the same as the local encryption, with share names being pushed into the shared thread struct for the child processes to encrypt.

Key findings

Overall, Conti ransomware is a sophisticated sample with many unique functionalities. By sacrificing the tremendous increase in size, the Conti team has implement a really troublesome string encryption method, which ended up taking me a while to go through and resolve all of the strings.

The encryption scheme is a bit boring with a randomly generated key protected by a hard-coded public RSA key. However, the multi-threading encryption is implemented elegantly using a shared structure between all of the threads, which results in extreme encrypting speed. Conti also avoids encrypting large files entirely, so it’s obvious that the malware authors prioritize speed over encrypting quality. With its networking functionality, the ransomware actively looks for available shares on the network to spread its encryption. This mainly targets small business and enterprise fields that uses the SMB protocol for file sharing, as we have seen with Advantech, Riverside Community Care, Ixsight Technologies, Total Systems Services, …

NOTE: For anyone who wants to analyze this sample further, you should set up a folder on your machine and runs the ransomware with the command line argument *“-p [directory]”* to test encryption on that directory only. It’s a pretty neat way to set up a small environment for testing and dynamic analysis that the authors have provided us with, so huge shoutout to them for that!

YARA rule

```
rule ContiV2 {
        meta:
                description = "YARA rule for Conti Ransomware v2"
                reference =
"http://chuongdong.com/reverse%20engineering/2020/12/15/ContiRansomware/"
                author = "@cPeterr"
                date = "2020-12-15"
                rule_version = "v2"
                malware_type = "ransomware"
                malware_family = "Ransom:W32/Conti"
                tlp = "white"
        strings:
                $str1 = "polzarutu1982@protonmail.com"
                $str2 = "http://m232fdxbfmbrcehbrj5iayknxnggf6niqfj6x4iedrgtab4qupzjlaid.onion"
                $str3 = "expand 32-byte k"
                $string_decryption = { 8a 07 8d 7f 01 0f b6 c0 b9 ?? 00 00 00 2b c8 6b c1 ?? 99 f7
fe 8d 42 7f 99 f7 fe 88 57 ff }
                $compare_size = { ?? ?? 00 00 50 00 }
        condition:
                all of ($str*) and $string_decryption and $compare_size
}
```

References

https://twitter.com/Arkbird_SOLG/status/1337565128561225728

https://twitter.com/VK_Intel/status/1297252264126685185

https://www.bleepingcomputer.com/news/security/conti-ransomware-shows-signs-of-being-ryuks-successor/

https://www.carbonblack.com/blog/tau-threat-discovery-conti-ransomware/

https://id-ransomware.malwarehunterteam.com/identify.php?case=2c61281154a1c9df22081099c5c36503a63e9b01

https://twitter.com/demonslay335/status/1339975671817318400