Technical Analysis of Crytox Ransomware

zscaler.com/blogs/security-research/technical-analysis-crytox-ransomware

Key points

- Crytox is a ransomware family consisting of several stages of encrypted code that was first observed in 2020
- The ransomware encrypts local disks and network drives and leaves a ransom note with a five day ultimatum, but does not exfiltrate data from the victim
- Crytox drops the uTox messenger application on the infected system that enables the victim to communicate and negotiate with the threat actors
- Crytox uses AES-CBC with a per file 256-bit key that is protected with a locally generated RSA public key
- File decryption may be possible via a known plaintext bruteforce attack

Summary

The threat actor using Crytox ransomware has been active since at least 2020, but has received significantly less attention than many other ransomware families. In September 2021, the Netherlands-based company RTL publicly acknowledged that they were compromised by the threat actor. The company paid Crytox 8,500 euros. Compared with current ransom demands, this amount is relatively low. Unlike most ransomware groups, the Crytox threat actor does not perform double extortion attacks where data is both encrypted and held for ransom.

The modus operandi of the group is to encrypt files on connected drives along with network drives, drop the uTox messenger application and then display a ransom note to the victim as shown in Figure 1.

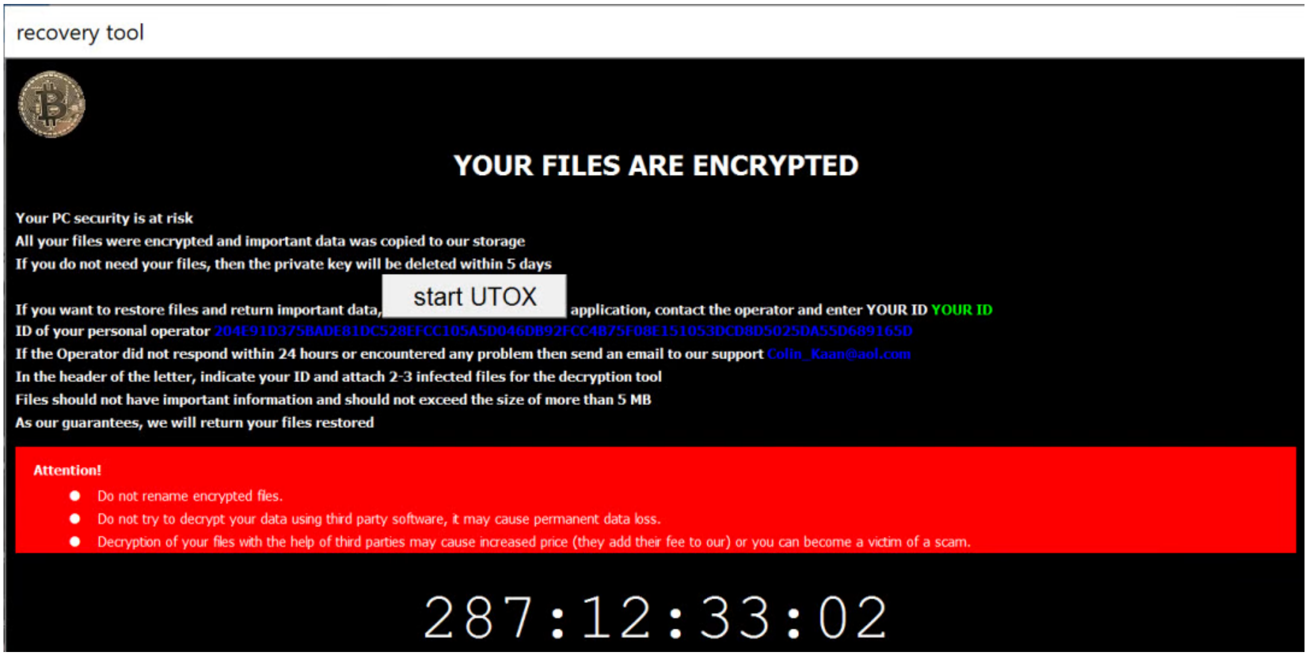


Figure 1. Crytox ransom note

The ransom demand period is set to five days to pressure the victim into paying as soon as possible

Technical analysis

The sample analyzed by ThreatLabz has the following SHA256 hash 32eef267a1192a9a739ccaaae0266bc66707bb64768a764541ecb039a50cba67. In most cases, Crytox samples are packed with UPX. Once decompressed, a sample usually weighs in around 1.23MB because the whole uTox client is embedded inside the malware.

Cryptox uses different techniques to thwart static analysis including the following:

- API hashing
- Encrypted configurations
- Encrypted shellcode
- Remote thread injection

Some parts of the malware look directly written in assembly. The most noteworthy thing is the use of a specific implementation of AES-CBC shown in Figure 2.

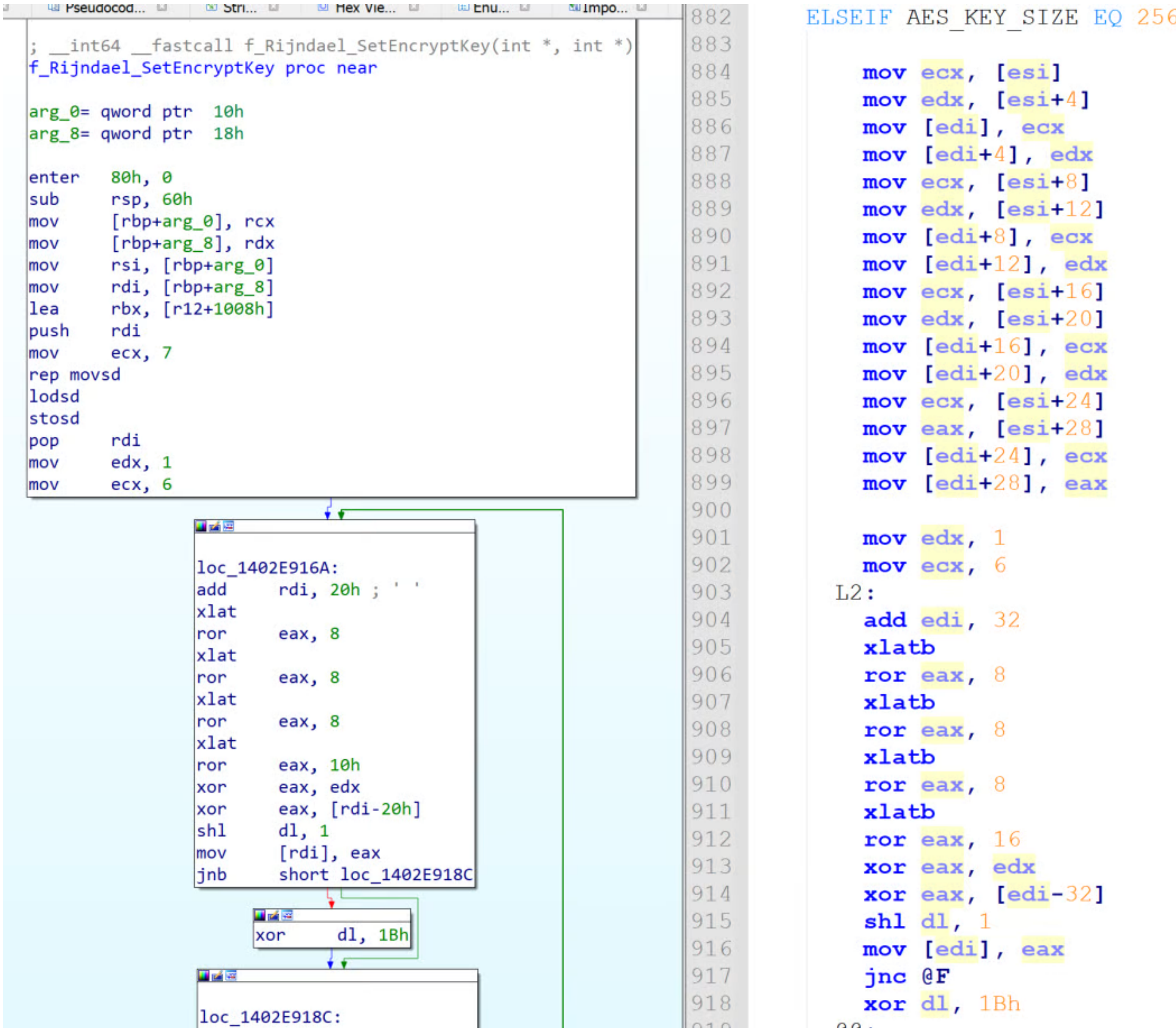


Figure 2. Crytox implementation of AES

The authors borrowed the AES code and modified some parts to meet their needs. They even added an alternative algorithm using Intel x86 AES instructions. Oddly enough, the authors chose to only implement the **Rijndael_Encrypt** routine to both *decrypt* their config and *encrypt* files. This means that when they embedded their configurations, they used the AES decryption routine to encrypt them. The key used for decrypting the Crytox configurations are either the first or second block of 32 bytes of the AES lookup table *Te1* using a NULL initialization vector (IV).

First-Stage

The malware encrypts the first-stage configuration using the aforementioned implementation of AES-CBC. Here, the AES key is the first 32 bytes of the *Te1* lookup table **a5c6636384f87c7c99ee77778df67b7b0dfff2f2bdd66b6bb1de6f6f5491c5c5** as shown in Figure 3.

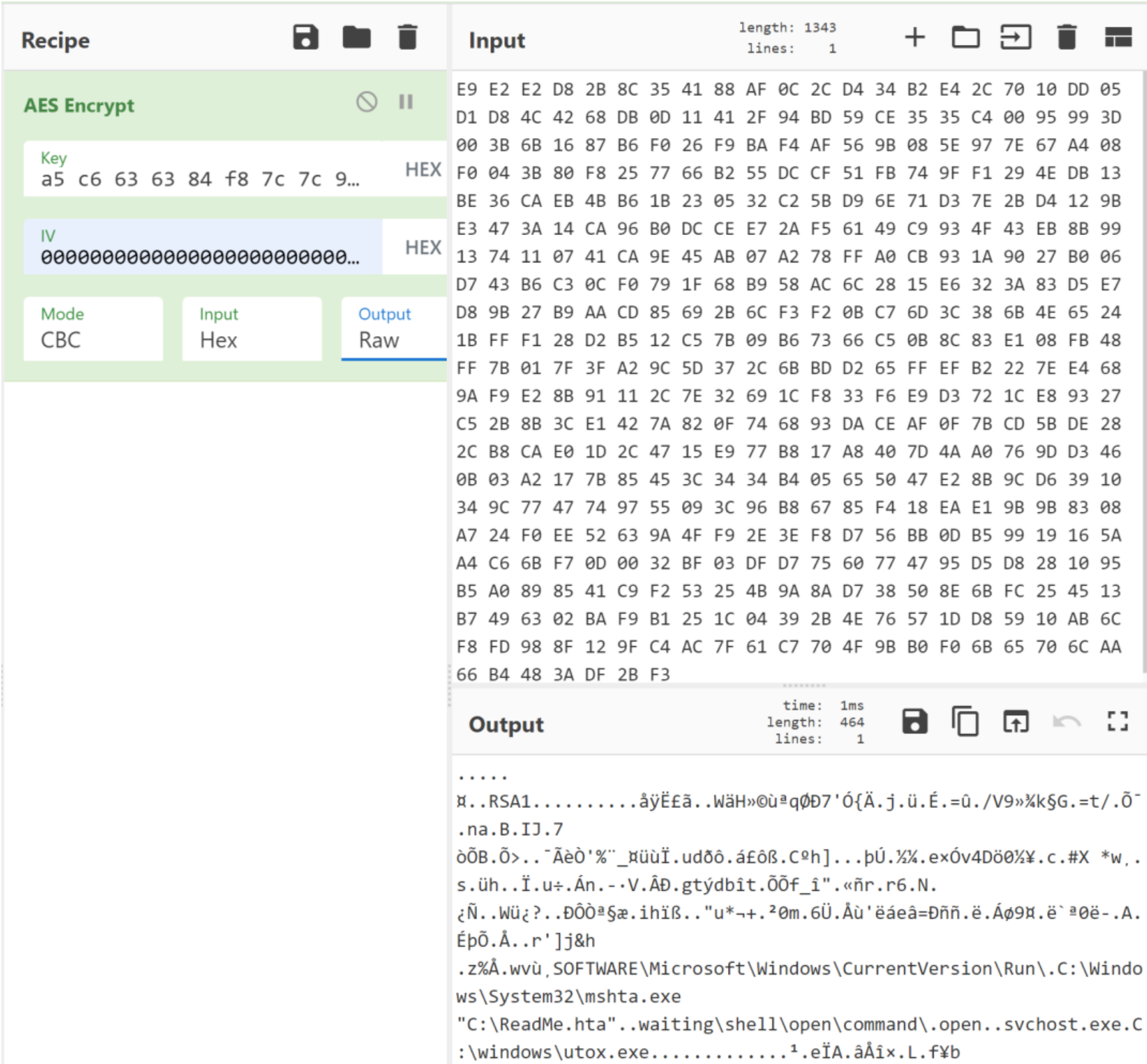


Figure 3. Crytox first-stage configuration after decryption

This configuration contains the following information:

- A hardcoded 2048-bit RSA public key
- The path to drop the uTox client application
- The *Run* registry value for the ransom note to be displayed at startup
- The process name to inject
- The class registry key to store the malware's configuration

After this configuration has been decrypted, the malware locally generates a 2048-bit RSA key pair using the **CryptGenKey** function. The generated RSA private key is then encrypted five times using the hardcoded public key.

Under the sub-key *HKCR\.waiting\shell\open\command*, the ransomware stores the following value-data pair shown in Table 1.

Value	Data
"en"	Generated RSA public key
"n"	Encrypted generated RSA private key
""	C:\Windows\System32\mshta.exe "C:\ReadMe.hta"

Table 1. Crytox registry configuration

In order to make sure the ransom note is displayed on startup, the registry value **open** along with the data "**C:\ReadMe.hta**" are created under **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run**

Once the Crytox configuration is stored, the code proceeds to locate a process to inject the second-stage. A remote thread is created to execute the first piece of shellcode.

Second-Stage

This stage decrypts a second configuration using AES-CBC with the following key 5060303003020101a9ce67677d562b2b19e7fefe62b5d7d7e64dabab9aec7676 (which is the second block of 32 bytes from the lookup table *Te1*). According to this decrypted configuration, the shellcode executes a batch file to delete shadow copies and remove events from the logs. Essentially the following commands are run::

```
for /F "tokens=*" %%1 in ('wevtutil.exe el') DO wevtutil.exe cl "%%1"
vssadmin.exe Delete Shadows /All /Quiet
diskshadow.exe /s ../pghdn.txt
```

The file pghdn.txt contains the line "**delete shadows all**".

Given the following hashing algorithm, the second-stage searches for the process ID (pid) of the process for which the hash of its name corresponds to 0xDCF164CD (explorer.exe) or 0x561F1820 (svchost.exe).

```
name = process_name + "\x00"
hash = 0
for c in name.upper():
    hash = ROTR32(hash, 0xD) + ord(c)
```

Inside a new thread, the shellcode creates a mutex by concatenating a hardcoded 4-letter word (e.g., "CSWS") with some random characters based on the pid of the targeted process as shown in Figure 4.

```
                mov     rcx, 4

loc_1402E95A1:
pid = rdx
                mov     al, dl
                and     al, 7
                add     al, 4Dh ; 'M'
                cld
                stosb
                shr     pid, 3
                loop    loc_1402E95A1
```

Figure 4. Crytox mutex creation

The thread then decrypts the content from the resource section of the original malware using the same algorithm and key as for the second configuration. This resource contains another shellcode, which is the final stage. This shellcode is injected inside the targeted remote process.

Third and Final Stage

Using the same encryption algorithm, with the first 32-bytes of the *Te1* lookup table as the AES key, this final stage decrypts the main configuration containing the following information:

- A seed for generating the file encryption key
- An .hta formatted ransom note
- A simple regular expression for listing all files on the system
- The encrypted file extension (e.g., *YOUR ID.waiting*)
- Privileges to remove (SeBackupPrivilege, SeRestorePrivilege)

First, the code tries to retrieve the configuration that the first stage stored in the registry hive. If this configuration doesn't exist, Crytox will create it. The code proceeds to set a *countdown* variable in the ransom note followed by replacing the string *YOUR ID* in the ransom note template. The latter value is replaced with a unique victim ID that is generated by the following pseudo-algorithm based on the encrypted locally generated RSA private key:

```
hash = *(_DWORD *)config_t->encrypted_priv_key;
counter = 9i64;
for ( initial_hash = 0x37; ; LOBYTE(initial_hash) = (hash & 0xF) + 0x4B )
{
  *(_WORD *)&config_t->generated_id[2 * counter] = initial_hash;
  config_t->readme_hta_content[counter + 0x27DF] = initial_hash;
  hash = __ROR4__(hash, 4);
  if ( !--counter )
    break;
}
```

Figure 5. Crytox victim ID generation algorithm

Before encrypting any files, the malware removes the SeBackupPrivilege and SeRestorePrivilege privileges. Using the functions **WNetOpenEnumW** and **WNetEnumResourceW**, the malware retrieves connected drives and for each drive found, a thread is created to encrypt files. The same process is applied for every logical drive using the function **GetLogicalDrives**. The malware then waits for a lock to be released before calling the **ShChangeNotify** function in order to change the icon and file association and to display the ransom note to the victim.

File encryption

The algorithm to discover all the files is relatively standard and relies on a recursive approach. The *Windows* directory is excluded from the search along with the ransom note and files with the .waiting extension. In addition, Crytox will only encrypt files that are larger than 16 bytes, which is the size of a block for AES. If the size of a file is not an exact multiple of 16 bytes, the malware will not pad and encrypt the last block of data. For large files, only the first 1,048,576 (0x100000) bytes are read and encrypted to optimize encryption speed.

For each file, a new 256-bit AES key is generated and the content of the file is encrypted using AES-CBC. Crytox then creates the following structure in Figure 6.

```
00000000 cipher_footer_t struc ; (sizeof=0x80, mappedto_29)
00000000                                              ; XREF: sub_A73/r
00000000 blob_hdr        BLOBHEADER ?
00000008 key_size        dd ?
0000000C key             db 32 dup(?)
0000002C filesizehigh    dd ?
00000030 filesizelow     dd ?
00000034 encrypt_size    dd ?
00000038 unk_0x37        dd ?
0000003C padding         db 68 dup(?)
00000080 cipher_footer_t ends
```

Figure 6. Cryptox cipher footer structure

The **BLOBHEADER** structure is set like this:

.bType = PLAINTEXTKEYBLOB
.bVersion = CUR_BLOB_VERSION
.aiKeyAlg = CALG_AES_256

Since the structure is not initialized, the *padding* structure is filled with random data.

This structure is encrypted with the locally generated RSA public key. The resulting cipher is concatenated to the end of the encrypted file followed by the encrypted generated RSA private key. The encrypted file is renamed by appending *YOUR ID.waiting* to the original filename with *YOUR ID* replaced by the victim ID computed as described previously.

A ransom note is written to every directory after encrypting all files that are present. A process flow chart for Crytox is illustrated in Figure 7.

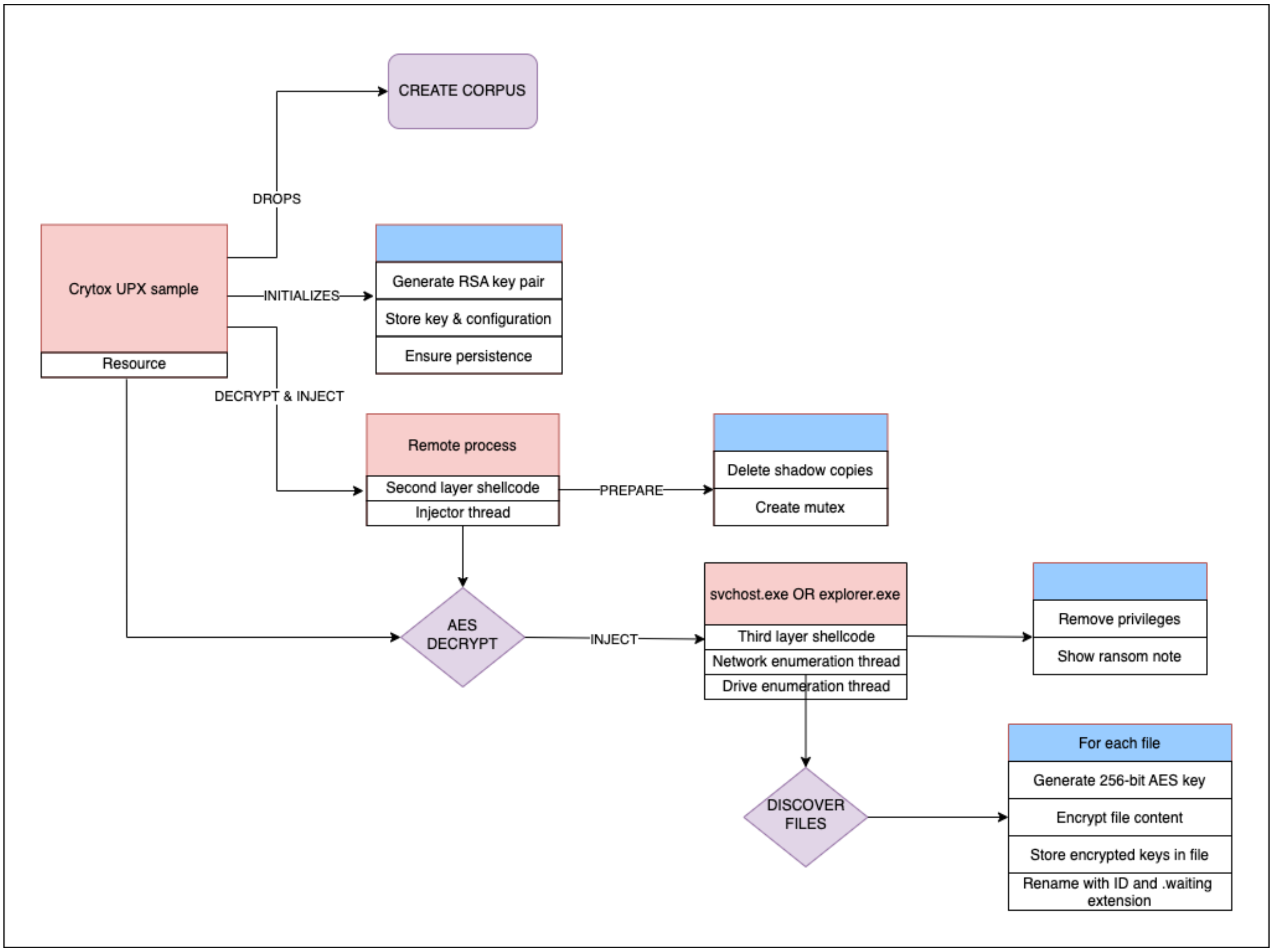


Figure 7. Process flowchart for Cryptox encryption

Key Generation Algorithm and Weakness

As stated previously, a 256-bit AES key is generated for each file that is encrypted. The following algorithm in Figure 8 is used for the key generation.

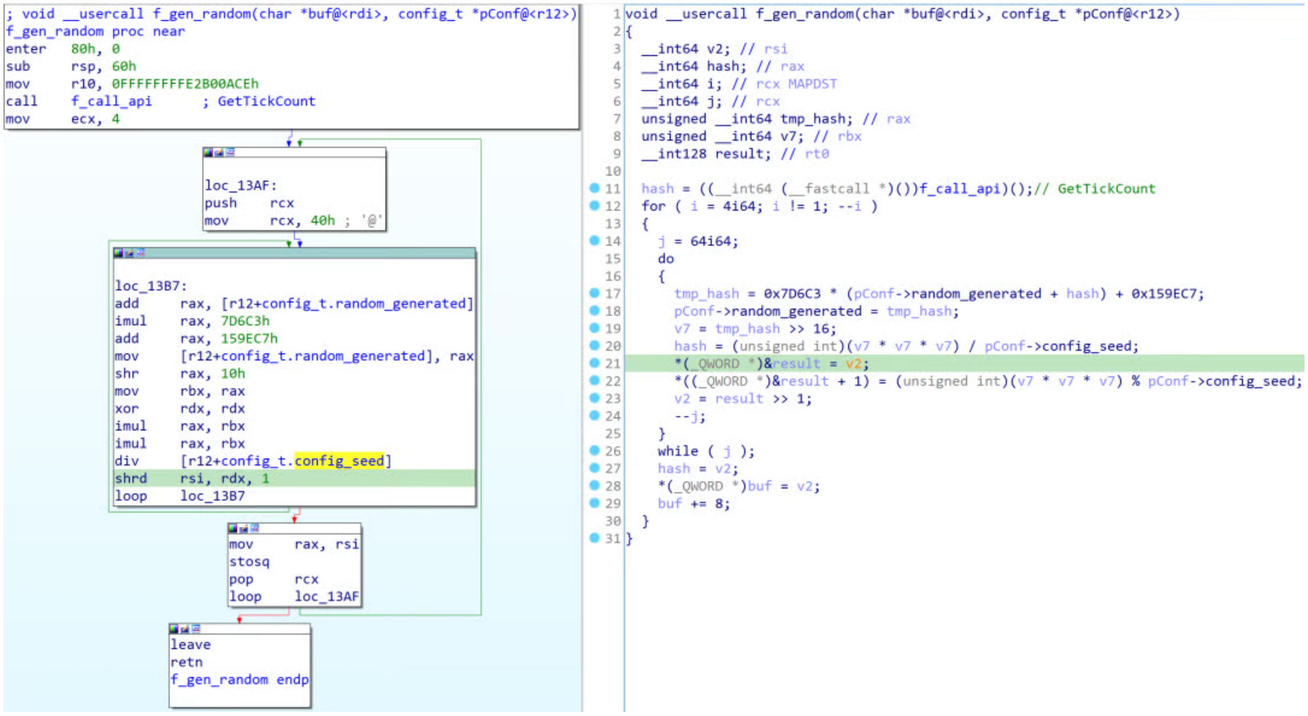


Figure 8. Crytox key generation algorithm

The custom pseudo random key generator functions relies on the variables below:

- A seed value determined by calling **GetTickCount**
- A 64-bit integer *config_t.random_generated* initially set to 0
- A 32-bit integer constant *config_t.config_seed*

The last value is stored inside the malware's configuration. This value has been the same across samples analyzed by ThreatLabz. The only unknown value necessary to determine the AES key is the value of **GetTickCount** at the time of encryption. However, if some plaintext of a file is known, efforts to bruteforce the AES key are feasible.

Based on file magic values, one can divise a bruteforce program with the following logic:

1. Set a *counter* to 0
2. Let the random generator create a key with the counter as the rotating seed
3. Decrypt the first block of the encrypted file
4. Compare a known magic value with the decrypted data
5. If the value matches, the initial value of **GetTickCount** and the key have been successfully identified. Else, increment *counter* and loop back to 2.

Figure 8 shows an bruteforce program running on a machine with 16 logic cores. Here, the encrypted file was *dotnet-sdk-3.1.416-win-x64.exe* (SHA1: 83A53E8770EDD38EDDD37DED63CEF2253FC16979) and the known plaintext was the Windows PE (MZ) file header *4D5A9000*.

```
PS>(Measure-Command {.\crtx2.exe dotnetsdk_encrypted.mem dotnet-sdk-3.1.416-win-x64.exe 0x4 | Out-Default}).ToString()
Found seed 0x0a4a4f6c with corresponding key:ec187b9530dbf0febf3f96b044604c1728652e38a104833e220dde5d26d9db57

00:12:42.3531808
```

Figure 9. Cryptox example bruteforce key recovery

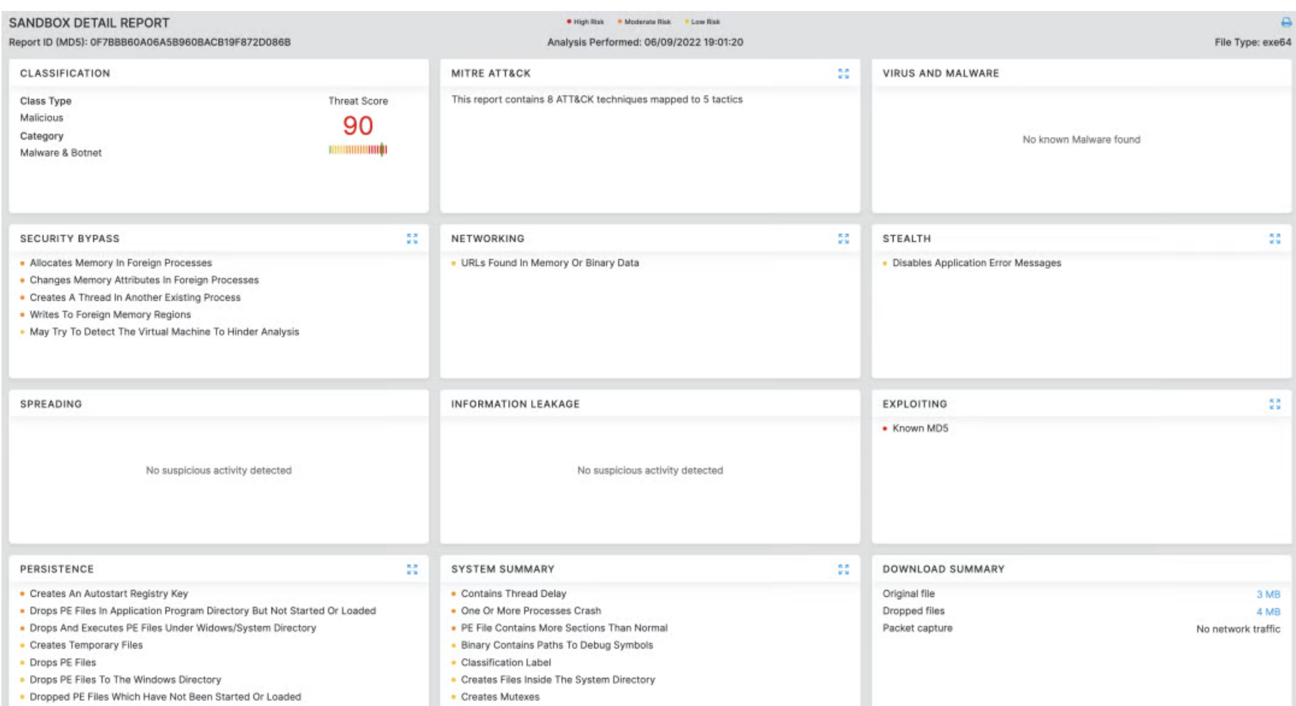
The method relies on knowing a part of the plaintext at a specific offset. Thus, only specific file types may be decrypted. Because the seed is based on **GetTickCount**, if one has access to the master file table (MFT) and is able to locate and decrypt the first and last file encrypted, then the range of **GetTickCount** values can be deduced. Therefore, the bruteforce range can be greatly reduced to decrypt all files.

Conclusion

Crytox exposes some interesting features to hinder static analysis by self-decrypting itself several times, injecting shellcode inside different processes, encrypting its configurations and using API hashing. The main file encryption logic of Crytox is standard using a unique AES key per file that is protected with RSA. However, the author(s) chose to rely on a weak random generator to create new AES keys. Using a 32-bit integer as the seed is not sufficient with today's computational power.

Ransomware families have a lot in common due to their shared goals and most use secure encryption schemes. However, there may still be implementation weaknesses that enable file decryption without having access to a private key. The bruteforce methods described in this blog could be reused for similar scenarios.

Cloud Sandbox Detection



In addition to sandbox detections, Zscaler's multilayered cloud security platform detects indicators related to the campaign with the following threat name:

Win64.Ransom.Crytox

Indicators of Compromise

Hashes

- 1c0bf0c2e7d0c34ec038a8b717bb19d9c4cf3382ada1412f055a9786d3069d78
- 2115c4c859d497eec163ca33798c389649543d8a6e4db5806a791c6186722b71
- 307c83924e90f4627f08c2f744cf51f18ec6e246687282a0c1794369ff084f42
- 3764200cfa673e8796e7c955454b57c20852c2a7931fb9f632ef89d267bbd4c8
- 6d4e75bc0cc095fef94b9d98a4e94ce9145890b435012b5624aa73621ba6e312
- 79aff06385c16a98594c6fd314c572bfbe07fbe923f30a627e9b86ac3ab7c071
- 8ee4a58699ecf02dca516dc6b5b72d93fd9968f672b2be6f8920dfec027d7815
- c5550f44332750552921cb5d685ccfbeefa2ab4b03aed8c51c5db52bbe2ff5d4
- d60dc6965f6d68a3e7c82d42e90bfda7ad3c5874d2c59a66df6212aef027b455

Files written

- **C:\ReadMe.hta**
- Files with ".waiting" extension

Registry keys

HKCR\.waiting\shell\open\command