

Security Research

Technical Analysis of Crytox Ransomware



ROMAIN DUMONT – Sr. Security Researcher
September 21, 2022 – 10 min read



SECURITY INSIGHTS



Contents

- 1 Article
- 2 More blogs

Copy URL

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.

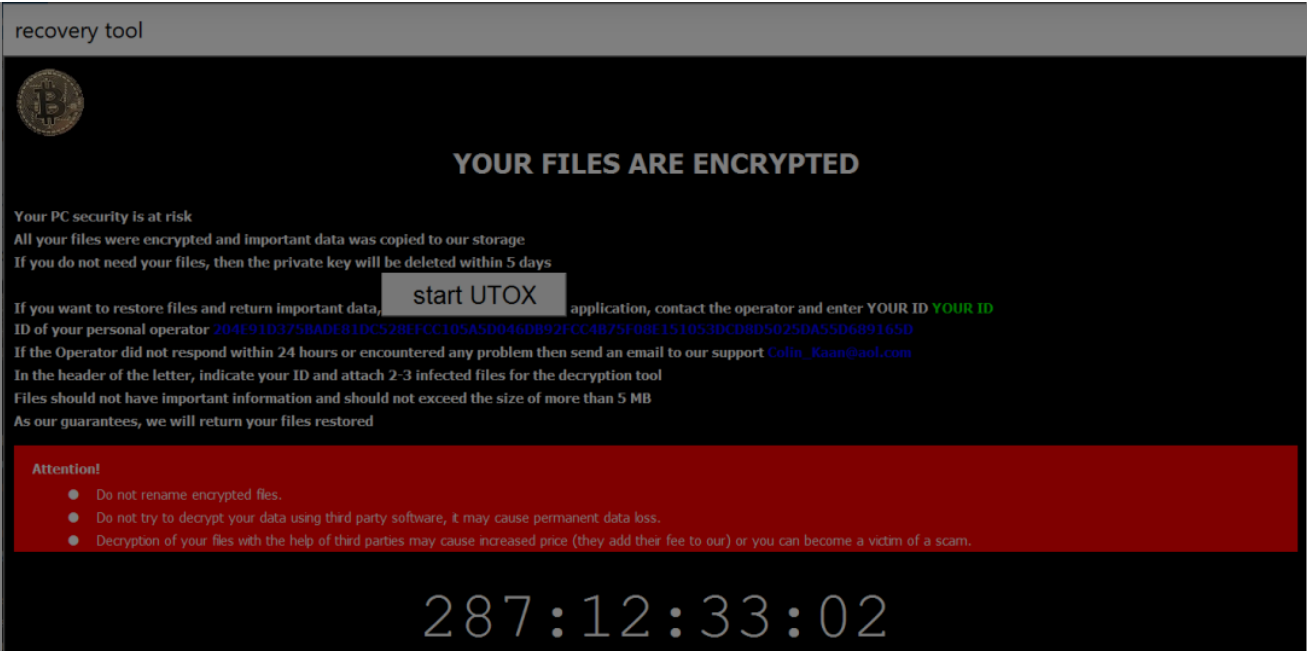


Figure 1. Crytox ransom note

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.

```

; __int64 __fastcall f_Rijndael_SetEncryptKey(int *, int *)
f_Rijndael_SetEncryptKey proc near

    arg_0= qword ptr 10h
    arg_8= qword ptr 18h

    enter 80h, 0
    sub rsp, 60h
    mov [rbp+arg_0], rcx
    mov [rbp+arg_8], rdx
    mov rsi, [rbp+arg_0]
    mov rdi, [rbp+arg_8]
    lea rbx, [r12+1008h]
    push rdi
    mov ecx, 7
    rep movsd
    lodsd
    stosd
    pop rdi
    mov edx, 1
    mov ecx, 6

    loc_1402E916A:
        add rdi, 20h ; ' '
        xlat
        ror eax, 8
        xlat
        ror eax, 8
        xlat
        ror eax, 8
        xlat
        ror eax, 10h
        xor eax, edx
        xor eax, [rdi-20h]
        shl dl, 1
        mov [rdi], eax
        jnb short loc_1402E918C

        xor dl, 18h
    
```

```

ELSEIF AES_KEY_SIZE EQ 256

    mov ecx, [esi]
    mov edx, [esi+4]
    mov [edi], ecx
    mov [edi+4], edx
    mov ecx, [esi+8]
    mov edx, [esi+12]
    mov [edi+8], ecx
    mov [edi+12], edx
    mov ecx, [esi+16]
    mov edx, [esi+20]
    mov [edi+16], ecx
    mov [edi+20], edx
    mov ecx, [esi+24]
    mov eax, [esi+28]
    mov [edi+24], ecx
    mov [edi+28], eax

    mov edx, 1
    mov ecx, 6
L2:
    add edi, 32
    xlatb
    ror eax, 8
    xlatb
    ror eax, 8
    xlatb
    ror eax, 8
    xlatb
    ror eax, 8
    xlatb
    ror eax, 16
    xor eax, edx
    xor eax, [edi-32]
    shl dl, 1
    
```

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 **a5c6636384f87c7c99ee77778df67b7bOdfff2f2bdd66b6bb1de6f6f5491c5c5** as shown in Figure 3.

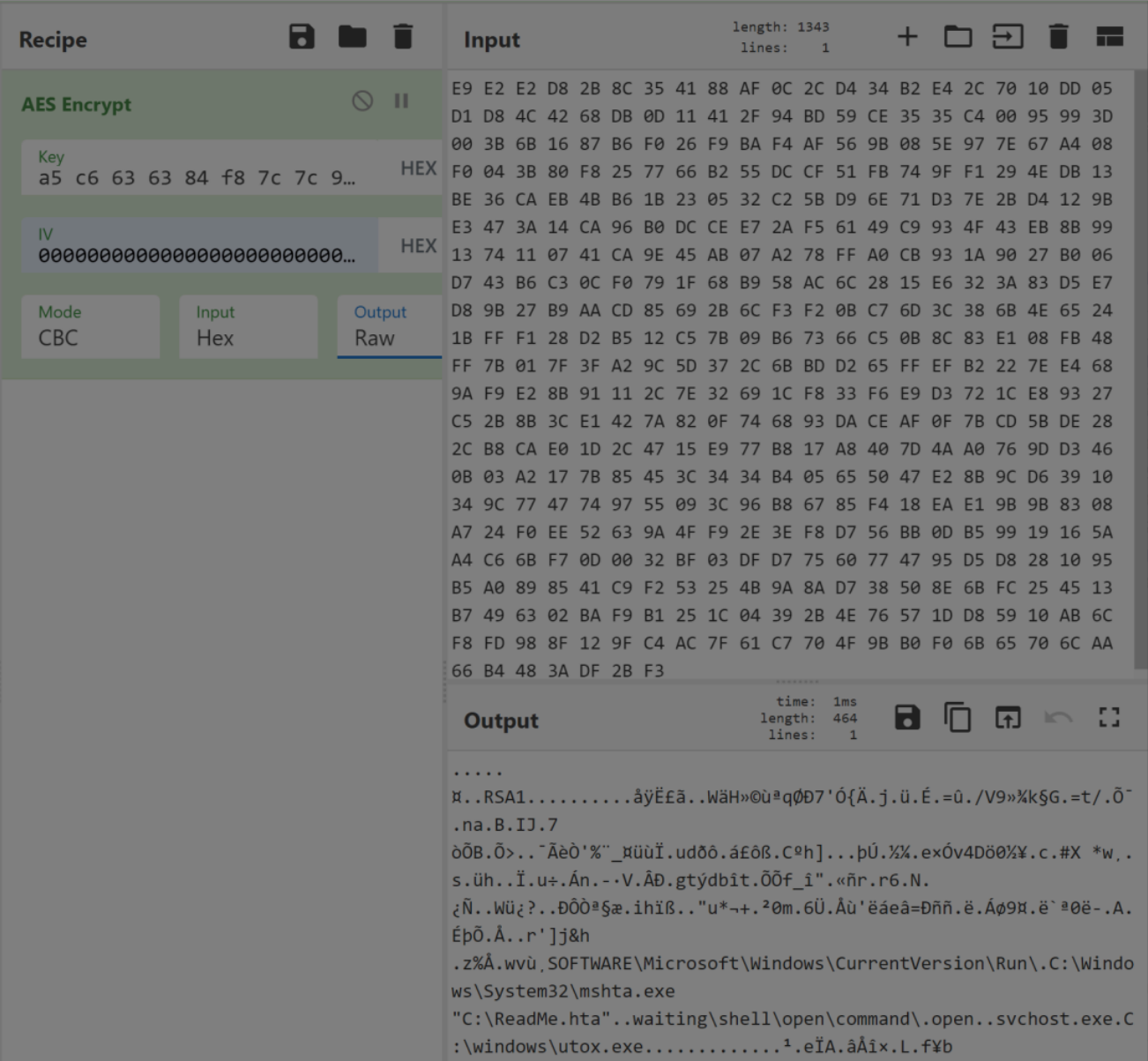


Figure 3. Crytox first–stage configuration after decryption

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.

"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 OxDCF164CD (explorer.exe) or Ox561F1820 (svchost.exe).

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

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.

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:

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.

Figure 6. Crytox 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

Figure 7. Process flowchart for Crytox 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 devise 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: 83A53E877OEDD38EDDD37DED63CEF2253FC16979) and the known plaintext was the Windows PE (MZ) file header *4D5A9000*.

Figure 9. Crytox 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.

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

- 1cObfOc2e7dOc34ecO38a8b717bb19d9c4cf3382ada1412fO55a9786d3O69d78
- 2115c4c859d497eec163ca33798c389649543d8a6e4db58O6a791c6186722b71
- 3O7c83924e9Of4627fO8c2f744cf51f18ec6e246687282aOc1794369ffO84f42
- 37642OOcfa673e8796e7c955454b57c2O852c2a7931fb9f632ef89d267bbd4c8
- 6d4e75bcOccO95fef94b9d98a4e94ce914589Ob435O12b5624aa73621ba6e312
- 79affO6385c16a98594c6fd314c572bfbeO7fbe923f3Oa627e9b86ac3ab7cO71
- 8ee4a58699ecfO2dca516dc6b5b72d93fd9968f672b2be6f892OdfecO27d7815
- c555Of4433275O552921cb5d685ccfbeefa2ab4bO3aed8c51c5db52bbe2ff5d4
- d6Odc6965f6d68a3e7c82d42e9Obfda7ad3c5874d2c59a66df6212aefO27b455

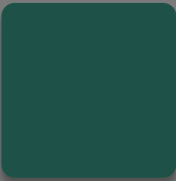
Files written

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

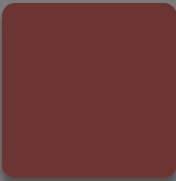
Registry keys

- HKCR\.waiting\shell\open\command

Was this post useful?



Yes, very!



Not really

Explore more Zscaler blogs



Agniane Stealer:
Dark Web’s Crypto
Threat



The Impact of the
SEC’s New
Cybersecurity



Security Advisory:
Remote Code
Execution

Read post

Read post

Read post

01 / 02



Get the latest Zscaler blog updates in your inbox

Email Address

Subscribe

By submitting the form, you are agreeing to our [privacy policy](#).

THE ZSCALER EXPERIENCE	PRODUCTS & SOLUTIONS	PLATFORM	RESOURCES	POPULAR LINKS
<p>Learn about:</p> <p>Your world, secured</p> <p>Zero Trust</p> <p>Secure Access Service Edge (SASE)</p> <p>Security Service Edge (SSE)</p> <p>Zero Trust Network Access (ZTNA)</p> <p>Secure Web Gateway (SWG)</p> <p>Cloud Access Security Broker (CASB)</p> <p>Cloud Native Application Protection Platform (CNAPP)</p> <p>Data Security Posture Management (DSPM)</p>	<p>Secure Your Users</p> <p>Secure Your Workloads</p> <p>Secure Your IoT and OT</p> <p>Secure Internet Access (ZIA)</p> <p>Data Protection (CASB/DLP)</p> <p>Digital Experience (ZDX)</p> <p>Industry & Market Solutions</p> <p>Partner Integrations</p> <p>Zscaler Client Connector</p>	<p>Zero Trust Exchange Platform</p> <p>Secure Digital Transformation</p> <p>Network Transformation</p> <p>Application Transformation</p> <p>Security Transformation</p>	<p>Resource Library</p> <p>Customer Success Stories</p> <p>Security Preview</p> <p>Threat Assessment Tools</p> <p>ThreatLabz Analytics & Insights</p> <p>Upcoming Events</p> <p>Blog</p> <p>Zscaler Academy</p> <p>CXO Revolutionaries</p> <p>Zpedia</p> <p>Ransomware Protection ROI Calculator</p>	<p>Pricing & Plans</p> <p>About Zscaler</p> <p>Leadership Team</p> <p>Career Opportunities</p> <p>Find or Become a Partner</p> <p>Customer Success Center</p> <p>Investor Relations</p> <p>Press Center</p> <p>News & Announcements</p> <p>ESG</p> <p>Compliance</p> <p>Contact Zscaler</p>



Zscaler is universally recognized as the leader in zero trust. Leveraging the largest security cloud on the planet, Zscaler anticipates, secures, and simplifies the experience of doing business for the world's most established companies.

English

Please enter your email to subscribe



WCAG 2.1 AA

Sitemap

Privacy

Legal

Security

© 2024 Zscaler, Inc. All rights reserved. Zscaler™ and other trademarks listed at [zscaler.com/legal/trademarks](#) are either (i) registered trademarks or service marks or (ii) trademarks or service marks of Zscaler, Inc. in the United States and/or other countries. Any other trademarks are the properties of their respective owners.



Zscaler uses cookies, pixels, and other tools to collect information you provide to us and to capture and record your interaction with our site. We use this information to enhance site navigation, personalize content, analyze your use of our website, and assist in our marketing efforts and customer service. To deliver the best experience and to assist with our efforts, Zscaler social media, advertising, analytics, and hosting service providers may have access to the information that you provide to us. By clicking "Accept All," you consent to our collection, use, and disclosure of such information and to our [Terms of Service](#). For more information about our data processing practices, please see our [Privacy Policy](#).