Technical Analysis of Crytox Ransomware

scaler.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 <u>RTL publicly acknowledged</u> 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 <u>uTox messenger application</u> 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 <u>specific implementation of AES-CBC</u> shown in Figure 2.

W Pseudocod W Stri W Hex Vie W W Enu W MImpo W	882 ELSETE AES KEY	STZE EO 256
: int64 fastcall f Riindael SetEncryptKev(int *. int *)	883	
f_Rijndael_SetEncryptKey proc near	884 mov ecx, [e	si
	885 mov edx, [si+4]
arg_0= qword ptr 10h	886 mov [edi],	ecx
arg_o= dword ptr 180	887 mov [edi+4]	, edx
enter 80h, 0	888 mov ecx. [si+81
sub rsp, 60h	889 mov edx.	si+121
mov [rbp+arg_0], rcx	890 mov [edi+8]	ecx
mov [rbp+arg_8], rdx	891 mov [edi+1]	l. edx
mov rdi, [rbp+arg 8]	892 mov ecx.	si+161
lea rbx, [r12+1008h]	893 mov edv	si+201
push rdi	894 mov [edi+10	
mov ecx, 7	895 mov [edi+20	l ody
lodsd		si+2/1
stosd	897 mov eax	si+281
pop rdi	898 mov [edi+2/	
mov edx, 1	899 mov [edi+28	
		1, can
	901 mov edv 1	
loc 140250164:	902 mov ecx 6	
add rdi, 20h : '	903 1.2.	
xlat	904 add adi 30	
ron eax, 8	905 vlath	<i>11</i>
xlat		
xlat	907 wlath	
ror eax, 8		
xlat	and whath	
ror eax, 10h		
xor eax, eax	011 wlath	
sh1 dl, 1		
mov [rdi], eax	old ror eax, it)
jnb short loc_1402E918C	915 XOF eax, ec	1: 201
	olf abl dl 1	
xor dl, 1Bh	SIJ SIL dI, I	
	mov [ed1],	eax
loc_1402E918C:	yis xor dl, IBr	1.

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.

Recipe	•	Î	Ir	າpu	t							leng lin	th: es:	1343 1			+	C	כ	€	Ī		-	
AES Encrypt	0		E9 D1 00	E2 D8 3B	E2 4C 6B	D8 42 16	2B 68 87	8C DB B6	35 0D F0	41 11 26	88 41 F9	AF 2F BA	0C 94 F4	2C BD AF	D4 59 56	34 CE 9B	B2 35 08	E4 35 5E	2C C4 97	70 00 7E	10 95 67	DD 99 A4	05 3D 08	
a5 c6 63 63 84 f8 7c 7	'c 9	HEX	FØ BE E3	04 36 47	3B CA 3A	80 EB 14	F8 4B CA	25 B6 96	77 1B BØ	66 23 DC	B2 05 CE	55 32 E7	DC C2 2A	CF 5B F5	51 D9 61	FB 6E 49	74 71 C9	9F D3 93	F1 7E 4F	29 2B 43	4E D4 EB	DB 12 8B	13 9B 99	
000000000000000000000000000000000000000	000	HEX	13 D7	74 43	11 B6	07 C3	41 0C	CA FØ	9E 79	45 1F	AB 68	07 B9	A2	78 AC	FF 6C	A0 28	CB	93 E6	1A 32	90 3A	27 83	B0 D5	06 E7	
Mode Input CBC Hex	Ot Ra	utput aW	D8 1B	9B FF 7D	27 F1	B9 28	AA D2	CD B5	85 12	69 C5	2B 7B	6C 09	F3 B6	F2 73	0B 66	C7 C5	6D ØB	3C 8C	38 83 82	6B E1	4E 08	65 FB	24 48	
			9A C5 2C	F9 2B B8	E2 8B CA	8B 3C E0	91 E1 1D	11 42 2C	9C 2C 7A 47	7E 82 15	32 0F E9	69 74 77	1C 68 B8	F8 93 17	33 DA A8	F6 CE 40	E9 AF 7D	D3 0F 4A	72 78 A0	1C CD 76	5B 9D	93 DE D3	27 28 46	
			0B 34 A7	03 9C 24	A2 77 F0	17 47 EE	7B 74 52	85 97 63	45 55 9A	3C 09 4F	34 3C F9	34 96 2E	B4 B8 3E	05 67 F8	65 85 D7	50 F4 56	47 18 BB	E2 EA ØD	8B E1 B5	9C 9B 99	D6 9B 19	39 83 16	10 08 5A	
			A4 B5 B7	C6 A0 49	6B 89 63	F7 85 02	0D 41 BA	00 C9 F9	32 F2 B1	BF 53 25	03 25 1C	DF 4B 04	D7 9A 39	75 8A 2B	60 D7 4E	77 38 76	47 50 57	95 8E 1D	D5 6B D8	D8 FC 59	28 25 10 70	10 45 AB	95 13 6C	
			66	FD B4	98 48	8F 3A	DF	9F 2B	F3	AC	75	61	0	70	41	98	BO	FØ	bВ	65	76	60	АА	ļ
			0)utp	out							1	tin engt line	ne: th: es:	1ms 464 1		8	٦	Ċ	(†)	k		0	
			ו•• ו• ו• ו• ו• ו• ו• ו• ו• ו		Al .IJ. .Ï.u .i¿?. .r' vvù, sten eadM	.7 Āèč Đć Đ	Ó'% Án ÒÒª§ &h =TW/ Msł nta'		àÿË iùï ,ÂĐ ihïi Mio .exo vait	Eã. .udd .gty 3 cro: e ting	.Wäl ðô ýdb: ″u*- sof f g\sl	H»©ù á£ôf ît.(īt.(1 t.\W: hel!	ùªq(3.C: ŽÕf <u></u> 20m L\op	007 2h] _î" .6Ü ows` .eï/	'Ó{Å ¦ .«ñr .Åù' \Cur \Cur	i.j. oÚ.) r.re 'ëáe rrer nman	.ü.E %%.e 6.N. eâ=f ntVe ntVe	É.=û ∍×Óv Đññ. Đññ. ersi .op∉ f¥b	1./\ /4Dd .ë./ ion\ en.	/9»% 50%¥ Áø9¤ \Rur	{k§({.c. ₹.ë` ۱\.(0.=t #X 2000 C:\V	:/.Õ ⁻ *w,. ≜A. √indc exe.(-

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	<pre>cipher_footer_t</pre>	<pre>struc ; (sizeof=0x80, mappedto_29)</pre>
00000000		; XREF: sub_A73/r
00000000	blob_hdr	BLOBHEADER ?
80000008	key_size	dd ?
0000000C	key	db 32 dup(?)
0000002C	filesizehigh	dd ?
00000030	filesizelow	dd ?
00000034	encrypt_size	dd ?
0000038	unk_0x37	dd ?
0000003C	padding	db 68 dup(?)
00000080	<pre>cipher_footer_t</pre>	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

SANDBOX DETAIL REPORT Report ID (MD5): 0F78B860A06A5B960BACB19F872D086B	High Risk Moderate Risk Low Risk Analysis Performed: 06/09/2022 19:01:20			File Type: exe64
CLASSIFICATION	MITRE ATT&CK	55	VIRUS AND MALWARE	
Class Type Threat Score Malicious 90 Category 90 Malware & Botnet	This report contains 8 ATT&CK techniques mapped to 5 tactics		No known Malware found	
SECURITY BYPASS	NETWORKING	55	STEALTH	55
Allocates Memory In Foreign Processes Changes Memory Attributes In Foreign Processes Creates A Thread In Another Existing Process Writes To Foreign Memory Regions May Try To Detect The Virtual Machine To Hinder Analysis	URLs Found In Memory Or Binary Data		Disables Application Error Messages	
SPREADING	INFORMATION LEAKAGE		EXPLOITING	20
No suspicious activity detected	No suspicious activity detected		Known MD5	
PERSISTENCE	SYSTEM SUMMARY	22	DOWNLOAD SUMMARY	
Creates An Autostart Registry Key Drops PE Files In Application Program Directory But Not Started Or Loaded Drops And Executes PE Files Under Widows/System Directory Creates Temporary Files Drops PE Files Drops PE Files To Tha Windows Directory Dropsed PE Files Which Have Not Been Started Or Loaded	Contains Thread Delay One Or More Processes Crash FF ERe Contains More Sections Than Normal Binary Contains Paths To Debug Symbols Classification Label Creates Files Inside The System Directory Creates Mutexes		Original file Dropped files Packet capture	3 MB 4 MB No network traffic

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
- 3764200cfa673e8796e7c955454b57c20852c2a7931fb9f632ef89d267bbd4c8
- 6d4e75bc0cc095fef94b9d98a4e94ce9145890b435012b5624aa73621ba6e312
- 79aff06385c16a98594c6fd314c572bfbe07fbe923f30a627e9b86ac3ab7c071
- 8ee4a58699ecf02dca516dc6b5b72d93fd9968f672b2be6f8920dfec027d7815
- c5550f44332750552921cb5d685ccfbeefa2ab4b03aed8c51c5db52bbe2ff5d4
- d60dc6965f6d68a3e7c82d42e90bfda7ad3c5874d2c59a66df6212aef027b455

Files written

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

Registry keys

HKCR\.waiting\shell\open\command