MANDIANT
NOW PART OF Google Cloud

FLARE-ON CHALLENGE 9 SOLUTION
BY JACOB THOMPSON (@IMPOSECOST)

Challenge 9: encryptor

Challenge Prompt

Another day, another ransomware infection. The network security team was able to grab the ransomware executable and an important file that's been encrypted. Can you break the encryption so that your company need not become a cryptocurrency connoisseur?

7-zip password: flare

Solution

The ransomware encrypts files using a per-file ChaCha20 key and uses a per-run RSA key pair to encrypt the per-file keys. The per-run RSA key pair is in turn encrypted using a per-campaign RSA public key baked into the malware, allowing the malware author to decrypt and return the per-run RSA private key in return for payment. But there is a fatal flaw—through a coding mistake, the ransomware accidentally signs the ChaCha20 keys instead of encrypting them, allowing anyone with the per-run public key to verify (decrypt) each file's ChaCha20 key and decrypt a file.

Introduction

The challenge comes with two files—an executable of about 26K, and a file with a .Encrypted extension that is 1101 bytes.

Looking at a hex dump of SuspiciousFile.txt.Encrypted, we see that it consists of some binary data with no apparent pattern, followed by several hex strings. In fact, there are four lines of hex strings, and each hex string consists of 256 hex characters, which decodes to 128 bytes each. An alert reader may notice that 128 bytes is exactly 1024 bits.

```
$ od -t x1z -A x SuspiciousFile.txt.Encrypted
000000 7f 8a fa 63 65 9c 5e f6 9e b9 c3 dc 13 e8 b2 31  >...ce.^........1<
000010 3a 8f e3 6d 94 86 34 21 46 2b 6f e8 ad 30 8d 2a  >:..m..4!F+o..0.*<
000020 79 e8 ea 7b 66 09 d8 d0 58 02 3d 97 14 6b f2 aa  >y..{f...X.=..k..<
000030 60 85 06 48 4d 97 0e 71 ea 82 06 35 ba 4b fc 51  >`..HM..q...5.K.Q<
000040 8f 06 e4 ad 69 2b e6 25 5b 39 66 31 38 37 37 36  >....i+.%[9f18776<
000050 62 64 33 65 37 38 38 33 35 62 35 65 61 32 34 32  >bd3e78835b5ea242<
000060 35 39 37 30 36 64 38 39 63 62 65 37 62 35 61 37  >59706d89cbe7b5a7<
000070 39 30 31 30 61 66 62 35 32 34 36 30 39 65 66 61  >9010afb524609efa<
000080 64 61 30 34 64 30 64 37 31 31 37 30 61 38 33 63  >da04d0d71170a83c<
000090 38 35 33 35 32 35 38 38 38 63 39 34 32 65 30 64  >853525888c942e0d<
0000a0 64 31 39 38 38 32 35 31 64 66 64 62 33 63 64 38  >d1988251dfdb3cd8<
0000b0 35 65 39 35 63 65 32 32 61 35 37 31 32 66 62 35  >5e95ce22a5712fb5<
0000c0 65 32 33 35 64 63 35 62 36 66 66 61 33 33 31 36  >e235dc5b6ffa3316<
0000d0 62 35 34 31 36 36 63 35 35 64 64 38 34 32 31 30  >b54166c55dd84210<
0000e0 31 62 31 64 37 37 61 34 31 66 64 63 63 30 38 61  >1b1d77a41fdcc08a<
0000f0 34 33 30 31 39 63 32 31 38 61 38 66 38 32 37 34  >43019c218a8f8274<
000100 65 38 31 36 34 62 65 32 65 38 35 37 36 38 30 63  >e8164be2e857680c<
000110 32 62 31 31 35 35 34 62 38 64 35 39 33 63 32 66  >2b11554b8d593c2f<
000120 31 33 61 66 32 37 30 34 65 38 35 38 34 37 66 38  >13af2704e85847f8<
000130 30 61 31 66 63 30 31 62 39 39 30 36 65 32 32 62  >0a1fc01b9906e22b<
000140 61 62 61 32 66 38 32 61 31 0a 64 63 34 32 35 63  >aba2f82a1.dc425c<
000150 37 32 30 34 30 30 65 30 35 61 39 32 65 65 62 36  >720400e05a92eeb6<
000160 38 64 30 33 31 33 63 38 34 61 39 37 38 63 62 63  >8d0313c84a978cbc<
000170 66 34 37 34 37 34 63 62 64 39 36 33 35 65 62 33  >f47474cbd9635eb3<
000180 35 33 61 66 38 36 34 65 61 34 36 32 32 31 35 34  >53af864ea4622154<
000190 36 61 30 66 34 64 30 39 61 61 61 30 38 38 35 31  >6a0f4d09aaa08851<
0001a0 31 33 65 33 31 64 62 35 33 62 35 36 35 63 31 36  >13e31db53b565c16<
0001b0 39 63 33 36 30 36 61 32 34 31 62 35 36 39 39 31  >9c3606a241b56991<
0001c0 32 61 39 62 66 39 35 63 39 31 61 66 62 63 30 34  >2a9bf95c91afbc04<
0001d0 35 32 38 34 33 31 66 64 63 65 65 36 30 34 34 37  >528431fdcee60447<
0001e0 38 31 66 62 63 38 36 32 39 62 30 36 66 39 39 61  >81fbc8629b06f99a<
```

```
0001f0 31 31 62 39 39 63 30 35 38 33 36 65 34 37 36 33  >11b99c05836e4763<
000200 38 62 62 64 30 37 61 32 33 32 63 36 35 38 31 32  >8bbd07a232c65812<
000210 39 61 65 62 30 39 34 64 64 61 66 34 63 33 61 64  >9aeb094ddaf4c3ad<
000220 33 34 35 36 33 65 65 39 32 36 61 38 37 31 32 33  >34563ee926a87123<
000230 62 63 36 36 39 66 37 31 65 62 36 30 39 37 65 37  >bc669f71eb6097e7<
000240 37 63 31 38 38 62 39 62 63 39 0a 38 65 36 37 38  >7c188b9bc9.8e678<
000250 66 30 34 33 63 30 64 38 62 38 64 33 64 66 66 33  >f043c0d8b8d3dff3<
000260 39 62 32 38 63 65 39 39 37 34 66 66 37 64 34 31  >9b28ce9974ff7d41<
000270 36 32 34 37 33 30 38 30 62 35 34 65 65 66 61 61  >62473080b54eefaa<
000280 36 64 65 63 62 38 38 32 37 37 31 37 63 36 62 32  >6decb8827717c6b2<
000290 34 65 64 66 66 66 37 30 36 33 33 37 35 62 36 35  >4edfff7063375b65<
0002a0 38 38 61 63 66 38 65 63 61 33 35 63 32 30 33 33  >88acf8eca35c2033<
0002b0 65 66 38 65 62 65 37 32 31 34 33 36 64 65 36 66  >ef8ebe721436de6f<
0002c0 32 66 36 36 35 36 39 62 30 33 64 66 38 63 35 38  >2f66569b03df8c58<
0002d0 36 31 61 36 38 65 35 37 31 31 38 63 39 66 38 35  >61a68e57118c9f85<
0002e0 34 62 32 65 36 32 63 61 39 38 37 31 66 37 32 30  >4b2e62ca9871f720<
0002f0 37 66 61 66 61 39 36 61 63 65 62 61 31 31 66 66  >7fafa96aceba11ff<
000300 64 33 37 62 36 63 34 64 62 66 39 35 62 32 35 36  >d37b6c4dbf95b256<
000310 31 38 34 39 38 33 62 61 64 34 30 37 63 37 39 37  >184983bad407c797<
000320 33 65 38 34 62 32 33 63 64 32 32 35 37 39 64 64  >3e84b23cd22579dd<
000330 32 35 62 66 34 63 31 61 30 33 37 33 34 64 31 61  >25bf4c1a03734d1a<
000340 37 62 30 64 66 64 63 66 64 34 34 0a 35 61 30 34  >7b0dfdcfd44.5a04<
000350 65 39 35 63 64 30 65 39 62 66 30 63 38 63 64 64  >e95cd0e9bf0c8cdd<
000360 61 32 63 62 62 30 66 35 30 65 37 64 62 38 63 38  >a2cbb0f50e7db8c8<
000370 39 61 66 37 39 31 62 34 65 38 38 66 64 36 35 37  >9af791b4e88fd657<
000380 32 33 37 63 31 62 65 34 65 36 35 39 39 62 63 34  >237c1be4e6599bc4<
000390 63 38 30 66 64 38 31 62 64 62 30 30 37 65 34 33  >c80fd81bdb007e43<
0003a0 37 34 33 30 32 30 61 32 34 35 64 35 66 38 37 64  >743020a245d5f87d<
0003b0 66 31 63 32 33 63 34 64 31 32 39 62 36 35 39 66  >f1c23c4d129b659f<
0003c0 39 30 65 63 65 32 61 35 63 32 32 64 66 31 62 36  >90ece2a5c22df1b6<
0003d0 30 32 37 33 37 34 31 62 66 33 36 39 34 64 64 38  >0273741bf3694dd8<
0003e0 30 39 64 32 63 34 38 35 30 33 30 61 66 64 63 36  >09d2c485030afdc6<
0003f0 32 36 38 34 33 31 62 32 32 38 37 63 35 39 37 32  >268431b2287c5972<
000400 33 39 61 38 65 39 32 32 65 62 33 31 31 37 34 65  >39a8e922eb31174e<
000410 66 63 61 65 34 37 65 61 34 37 31 30 34 62 63 39  >fcae47ea47104bc9<
000420 30 31 63 65 61 30 61 62 62 32 63 63 39 65 66 39  >01cea0abb2cc9ef9<
000430 37 34 64 39 37 34 66 31 33 35 61 62 31 66 34 38  >74d974f135ab1f48<
000440 39 39 39 34 36 34 32 38 31 38 34 63 0a           >99946428184c.<
00044d
```

Now we turn to flareon.exe. It's a x86_64, console-mode executable:

```
$ file flareon.exe
flareon.exe: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
```

Running it, we just get a usage statement. Apparently this ransomware was meant for manual execution by a threat actor who has access to the system, and not for automatic encryption via a phishing attack:

```
C:\>flareon.exe
usage: flareon path [path ...]
```

Let's give it an empty file to work on, so that perhaps we can distinguish between ciphertext and any headers added to a file. It doesn't seem to like that. It doesn't work with a non-empty file either.

```
C:\>copy nul empty.txt
        1 file(s) copied.
```

```

C:\>flareon.exe empty.txt
0 File(s) Encrypted

C:\>echo hello > notempty.txt

C:\>flareon.exe notempty.txt
0 File(s) Encrypted
```

Breaking out IDA and locating main() by backtracking from the call to exit(), we find that the malware will only encrypt files if their names end in .EncryptMe:

Renaming our empty file to end in .EncryptMe and trying again, we get an encrypted file (.Encrypted) and on the desktop get a ransom note:

```
C:\>ren empty.txt empty.EncryptMe

C:\>flareon empty.EncryptMe
empty.EncryptMe
1 File(s) Encrypted

C:\>dir /b
empty.Encrypted
empty.EncryptMe
flareon.exe
notempty.txt
SuspiciousFile.txt.Encrypted
```

empty. Encrypted is 1028 bytes long (four 256-character strings and four LF characters), and interestingly, the information in the ransom notes that needs to be sent to the attacker is just the first three hex strings in the file. Further testing would find that the very last hex string, in contrast, is unique to each encrypted file.

Within main() is a GetProcAddress() call on an unusual symbol. Background research reveals that SystemFunction036 is just an exported symbol for the RtlGenRandom() function (https://docs.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-rtlgenrandom) and further research shows this is Microsoft's cryptographically secure random number generator but in a more convenient form than loading all the cryptographic libraries and cryptographic service providers, etc. (https://en.wikipedia.org/wiki/CryptGenRandom) So long as this function is consistently used where random numbers are needed, the flaw in this FLARE-ON challenge is unlikely to stem from flawed random number generation, and brute force attempts to break the cryptography would not be successful.

Operation

Further reverse engineering of the main() function reveals its overall control flow:

The function calls sub_402560() and analysis indicates this is just a compiler-generated initialization function.

The function obtains a pointer to SystemFunction036 in advapi32.dll and stores this pointer in a global variable. Cross references to this variable, qword_409040 include sub_4022a3(), sub_401a70(), and sub_401d4b(). Given that this is a cryptographically secure random number generator, this raises suspicion that those three functions involve key generation.

If argc is 0 or 1, the malware issues a usage statement and exits. Otherwise, it calls sub_4021d0(), and then loops on the command line arguments.

If a command line argument ends in .EncryptMe, the ransomware opens that file for reading and in binary mode, and if that succeeds, changes the extension to .Encrypted and tries to open such a file for reading, ensuring that it fails (so that it won't overwrite an existing file). The malware opens the .Encrypted file for writing, and then the malware passes both FILE * pointers to sub_4022a3(). The malware closes both files, frees the .Encrypted filename string, and moves on to the next argument. Passing every file to sub_4022a3() means that function might be the per-file encryption function.

After the last argument, the ransomware prints the "%u File(s) Encrypted\n" string, and if at least one file was encrypted, call sub_402390b()—and analysis of that function indicates it is obviously generating and dropping the ransom note on the desktop.

Cryptographic Initialization Function

Review of sub_4021d0() reveals it is indeed a cryptographic initialization function. Complicating analysis is the fact that this ransomware seems to use a handmade big number library. The big number library can handle integers up to 1088 bits long, and they are stored least-significant byte first (little-endian). Here is a mapping of function addresses to functionality. For the inspiration, see https://www.mandiant.com/resources/blog/cryptography-blackmatter-ransomware.

sub_401caf() – generate random, 512-bit possible prime

sub_401a70() – generate random 512-bit odd integer with top two bits set

sub_401941() – compute remainder of big number by a plain integer

sub_401992() – add plain integer to a big number

sub_401df2() – Miller-Rabin primality test

sub_4018a0() – decrement big number

sub_4019b8() – subtract plain integer from big number

sub_401a4f() – shift even big number to the right until it becomes odd (returning the shift count)

sub_401d4b() – generate random big number within a range

sub_4019de() – subtract two big numbers

sub_4016cc() – modular exponentiation by repeated squaring

sub_4015be() – modular multiplication

sub_401a0a() – compare big number to plain integer

So far, sub_4021d0() seems to be generating a 512-bit prime, and in fact repeats this a second time. Given background knowledge of RSA, this is a giveaway that the ransomware is trying to generate p and q which will be multiplied together to compute n, the public key, and the private key will then be found by calculating e^(-1) "mod"(p-1)(q-1) where e is usually 65537. Further review of sub_4021d0() reveals this is indeed the case:

sub_401550() – multiply two big numbers (not modular)

sub_401b46() – modular inverse using Extended Euclidean algorithm sub_4018be() – is big number equal to zero?

sub_401778() – divide two big numbers

sub_4018d1() – modular subtraction

Given observation of these algorithms and knowledge of RSA, we can start labeling the functions and variables in the cryptographic initialization function:

Given a ransomware author's motivation, the overall structure of the function makes sense. It would make sense for an author to generate a per-run RSA key pair, encrypt the private key using the campaign RSA public key hard-coded in the malware, then throw away the per-run private key as it isn't needed any more to encrypt per-file keys which would happen later. But there is a huge mistake here—the bn_modinv() call to compute d by computing 65537^(-1) "mod" ϕ(n) stores its result in the variable... e, overwriting it! Accordingly, the bn_modexp() function which is attempting to encrypt d by taking its value from the var_A0 stack location, is in fact just encrypting stack garbage as further analysis reveals var_A0 is never written anywhere.

Consider the ramifications here—if instead of RSA-encrypting values by raising them to d mod n, allowing the recipient to decrypt the result by raising it to e mod n, the author has effectively swapped these values by correctly calculating d but then overwriting e with the result. Given n and the knowledge that e = 65537, anyone can RSA-decrypt per-file keys written by this ransomware once the data format and symmetric encryption algorithm are identified so that the per-file key can be used. To figure that out, the next function to look at is the per-file encryption function.

Per-File Encryption Function

Analysis of sub_4022a3() reveals it is indeed a per-file encryption function. The overall control flow is that it:

Calls RtlGenRandom() to generate 32 bytes (256 bits) of data.

Calls RtlGenRandom() to generate 12 bytes of data.

Calls sub_4020f0(), which loads the string "expand 32-byte k," which is a dead giveaway for possible ChaCha20

sub_401f10() indeed is the ChaCha20 transformation function

sub_4020f0() makes repeated calls to fread and fwrite on 64 bytes at a time, and is encrypting the buffer with ChaCha20 in between

Calls the modular exponentiation function to encrypt the 32- and 12-byte buffer (which proves to be a ChaCha20 key and nonce) using the per-run RSA private key (due to the mix-up between e and d).

Writes the per-campaign public key (n, as hard-coded in the binary) to the end of the file as ASCII hex characters.

Writes the per-run public key (n, as generated at startup) to the end of the file as ASCII hex characters.

Writes the encrypted per-run private key (which is correctly encrypted unlike the per-file key but is just encrypted stack garbage) to the end of the file as ASCII hex characters.

Writes the encrypted per-file ChaCha20 key and nonce (which was wrongly encrypted using the per-run private key) to the end of the file as ASCII hex characters.

Now we see the meaning of the three strings in the ransom note (also in each file)—the per-campaign key helps the attacker identify the victim and lets the attacker look up the private key that goes along with it. The per-run public key is needed by the decryption tool. The encrypted per-run private key, which should have revealed the per-run d value when decrypted, would have been decrypted and given to the "customer" along with the decryptor in exchange for payment. And the encrypted per-file key would of course be used by the decryptor to ChaCha20-decrypt the file.

Breaking the Encryption

Because the per-file key is an encrypted (or more accurately, signed) message m^d "mod " n, and we know n and e, we can decrypt (or more accurately, verify) the message by raising it to e, mod n. Of course, a real cryptographic library would have padded the message prior to the raw modular exponentiation, and would not sign raw messages, but hashes thereof.

Extracting the values from SuspiciousFile.txt.Encrypted, we have:

```
n =
0xdc425c720400e05a92eeb68d0313c84a978cbcf47474cbd9635eb353af864ea46221546a0f4d09aaa0885113e31db5
3b565c169c3606a241b569912a9bf95c91afbc04528431fdcee6044781fbc8629b06f99a11b99c05836e47638bbd07a2
32c658129aeb094ddaf4c3ad34563ee926a87123bc669f71eb6097e77c188b9bc9,
m =
0x5a04e95cd0e9bf0c8cdda2cbb0f50e7db8c89af791b4e88fd657237c1be4e6599bc4c80fd81bdb007e43743020a245
d5f87df1c23c4d129b659f90ece2a5c22df1b60273741bf3694dd809d2c485030afdc6268431b2287c597239a8e922eb
31174efcae47ea47104bc901cea0abb2cc9ef974d974f135ab1f4899946428184c,
```

and from reviewing the code and from familiarity with RSA, we know that e = 65537.

We don't know the per-campaign private key, and don't need it:

```
$ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex( pow(
0x5a04e95cd0e9bf0c8cdda2cbb0f50e7db8c89af791b4e88fd657237c1be4e6599bc4c80fd81bdb007e43743020a245
d5f87df1c23c4d129b659f90ece2a5c22df1b60273741bf3694dd809d2c485030afdc6268431b2287c597239a8e922eb
31174efcae47ea47104bc901cea0abb2cc9ef974d974f135ab1f4899946428184c, 65537,
0xdc425c720400e05a92eeb68d0313c84a978cbcf47474cbd9635eb353af864ea46221546a0f4d09aaa0885113e31db5
3b565c169c3606a241b569912a9bf95c91afbc04528431fdcee6044781fbc8629b06f99a11b99c05836e47638bbd07a2
32c658129aeb094ddaf4c3ad34563ee926a87123bc669f71eb6097e77c188b9bc9) )
'0x958f924dfe4033c80ffc490200000000989b32381e5715b4a89a87b150a5d528c943a775e7a2240542fc392aa197b
001'
>>>
```

The run of zeros is the initial ChaCha20 count value and, happily, this ChaCha20 implementation is totally compatible with the openssl 1.1.1 command line tool. After adjusting for byte order, and chopping off the 1028-byte block of hex values at the end of the file we decrypt the file and recover the flag:

```
$ dd bs=73 count=1 if=SuspiciousFile.txt.Encrypted | openssl chacha20 -K
01b097a12a39fc420524a2e775a743c928d5a550b1879aa8b415571e38329b98 -iv
000000000249fc0fc83340fe4d928f95
1+0 records in
1+0 records out
73 bytes copied, 0.000125028 s, 584 kB/s
Hello!
```

The flag is:

```
R$A_$16n1n6_15_0pp0$17e_0f_3ncryp710n@flare-on.com
```