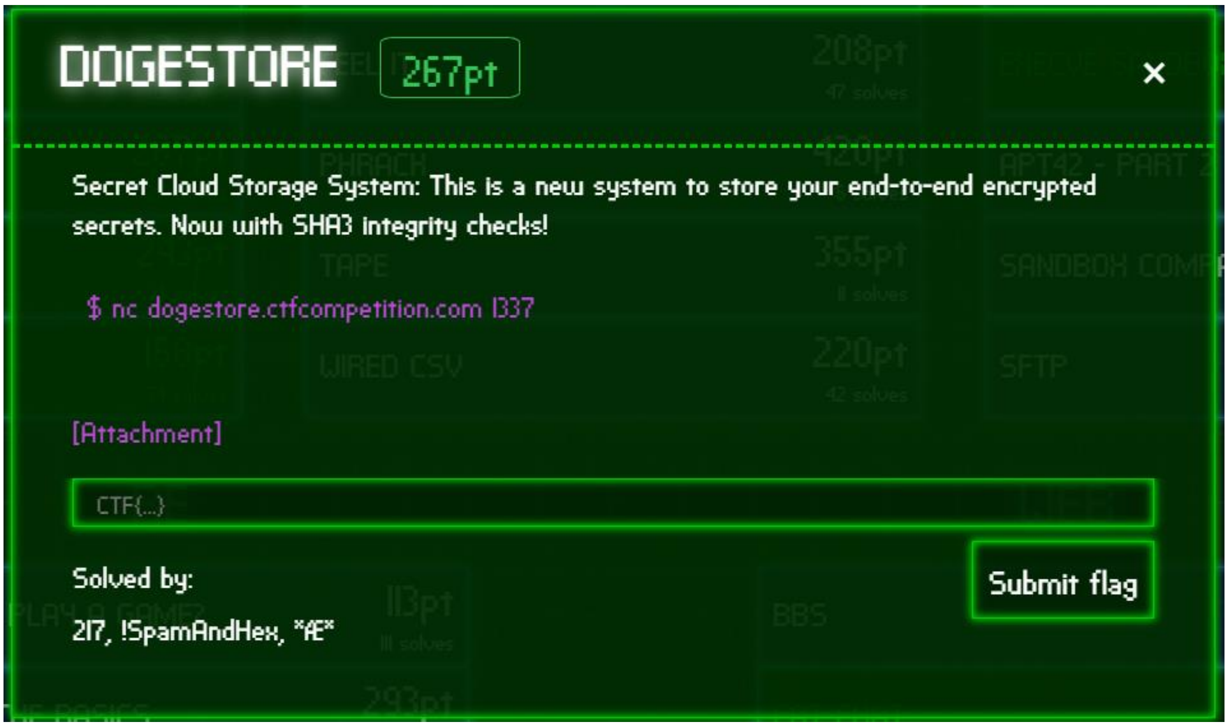
Google CTF 2018

Dogestore Writeup

shediyo @ PseudoRandom



DOGESTORE 267pt

Secret Cloud Storage System: This is a new system to store your end-to-end encrypted secrets. Now with SHA3 integrity checks!

$ nc dogestore.ctfcompetition.com 1337

[Attachment]

CTF{...}

Solved by:
217, !SpamAndHex, *Æ*

Submit flag

This time there is Rust code of an integrity server, and an encrypted flag.

The server receives a block of data B, decrypts it using $AES-CTR$ with a constant (I implicitly guessed) key and IV, decodes the result, and then sends the SHA3 of the decoded data (in base 64, that does not matter for the attack).

The encoding is Run-Length encoding, for each character in the plaintext there are 2 bytes in the encoding – the character, and the "run-length" - extra times the character appears in the plaintext until it is switched by another one.

For example - a plaintext "HHHHELLLLLOOO" will be RLE encoded as H,3,E,0,L,4,O,2.

The oracle - send a block B of length 110 and get

$$SHA3\big(\ RLE^{-1}\big(AES-CTR-DEC_{K,IV}(B\ )\big)\ \big).$$

From that I also understand that the given data is

$$AES-CTR-ENC_{K,IV}(RLE(flag\,))$$ for unknown K, IV.

In AES-CTR a key stream is made using the IV and the key K, not depending on the plaintext P in the following way:

$$KS = AES_K(IV), AES_K(IV+1), AES_K(IV+2), \ldots$$

Then the plaintext P is just XOR-ed to the bytes of the keystream KS.

I denote the given data G which is the XOR of F (the flag data notation) and the keystream KS.

G_1	G_2	G_3	G_4	.	.	.	.	.	G_{10}

$$=$$

F_1	L_1	F_2	L_2	.	.	.	.	.	L_5

$$\oplus$$

KS_1	KS_2	KS_3	KS_4	.	.	.	.	.	KS_{10}

After each flag byte F_i there is a length byte L_i (RLE encoding).

The goal is finding $F_1, L_1, \ldots, F_{55}, L_{55}$.

Finding the flag characters bytes

The first thing I noticed is that if I XOR the given data with any byte stream I want then the RLE data will be the flag XOR-ed with those same bytes

$G_1 \oplus X_1$	$G_2 \oplus X_2$	$G_3 \oplus X_3$	$G_4 \oplus X_4$

$$=$$

$F_1 \oplus X_1$	$L_1 \oplus X_2$	$F_2 \oplus X_3$	$L_2 \oplus X_4$

$$\oplus$$

KS_1	KS_2	KS_3	KS_4

That is, I can change the flag bytes and the length bytes and get decoded data of different characters and lengths.

I'm receiving only the $SHA3$ of the data so the best I can hope for in this case is getting information regarding the flag bytes through ambiguous data interpretation, leading to equal/different $SHA3$ answers.

An interesting relevant property of RLE encoding is that there are several options for encoding a string "inefficiently", for example the string 'CCC' can be encoded as C,2 but also C,0,C,1 or C,1,C,0 or C,0,C,0,C,0.

This property can be used for forging different length bytes with the same sum and same characters to get the same decoded data and so the same $SHA3$.

I denote $X_{1,2} = F_1 \oplus F_2$, and I assume (for now) that $L_1 = L_2 = 0$.

Notice that whether I XOR the given data with $0,1,X_{1,2},0,0 \dots 0$ or $0,0,X_{1,2},1,0 \dots 0$ - I get the same data after RLE decoding:

G_1	$G_2 \oplus 1$	$G_3 \oplus X_{1,2}$	G_4

$$=$$

F_1	1	F_1	0

$$\oplus$$

KS_1	KS_2	KS_3	KS_4

G_1	G_2	$G_3 \oplus X_{1,2}$	$G_4 \oplus 1$

$$=$$

F_1	0	F_1	1

$$\oplus$$

KS_1	KS_2	KS_3	KS_4

Both will be decoded as F_1, F_1, F_1 concatenated with the remaining flag and the $SHA3$ will be the same.

Moreover, XOR of G_3 with a value different from $X_{1,2}$ will decode as different data because there will be different characters with different lengths each time.

Using that I can use the oracle to efficiently find the XOR of two adjacent flag bytes:

```
for byte in range(256):
        first_send_data = xor_in_index(local_enc_data[:], chr(1),
                                        change_index - 1)
        first_send_data = xor_in_index(first_send_data, chr(byte),
                                        change_index)
        sha3_data_first = get_oracle_answer(first_send_data)

        second_send_data = xor_in_index(local_enc_data[:], chr(1),
                                         change_index + 1)
        second_send_data = xor_in_index(second_send_data, chr(byte),
                                         change_index)
        sha3_data_second = get_oracle_answer(second_send_data)

        if sha3_data_second == sha3_data_first:
                print('Found xor of bytes ' + str(byte))
```

This technique follows the assumption that the flag is RLE encoded with each character appearing once ($L_1 = L_2 =..= L_{55} = 0$) but actually it is not the case.

The logic fails for example if $L_0 = 0, L_1 = 1$, then we get $L_0 \oplus 1 = 1$ but $L_1 \oplus 1 = 0$, the length sum is different, because the bit is set (added) at the first XOR operation and cleared (subtracted) at the second operation.

Fortunately, a practical assumption was that the flag letters do not appear more than 0x80 times, so we can XOR with 0x80 setting the bit in both cases.

This way I got $F_i \oplus F_{i+1}$ for each (valid) i:

[14, 14, 14, 14, 12, 12, 12, 12, 12, 23, 18, 32, 32, 2, 23, 18, 61, 40, 18, 5, 5, 18, 23, 23, 23, 7, 23, 18, 61, 55, 19, 26, 26, 13, 13, 16, 22, 6, 21, 15, 11,5, 2, 7, 29, 46, 60, 18, 23, 7, 23, 18, 61, 113]

I tried all possible values for F_1 in the range of [0,255] and checked whether the result string contains the 'CTF{' flag indicator, and indeed:

HFHFHDHDHDSAaACTF{SADASDSDCTF{L_E_R_OY_JENKINS}ASDCTF{

The substring 'CTF{L_E_R_OY_JENKINS}' looks like a valid flag but it isn't, and indeed I had to find the length bytes for the whole flag.

<u>Finding the length bytes</u>

Knowing all the flag bytes, I can set them to be equal on the next oracle requests.

I also assumed that the '{' character has length 0, and so I have to deal with the case that only the second length is unknown (once I know it – I can reset it to 0 again and continue with the same assumption for the next length byte).

If we XOR the lengths with a single-set-bit byte Q (0x1,0x2,0x4..,0x80):

$$\begin{array}{|c|c|c|c|}\hline G_1 & G_2 \oplus Q & G_3 & G_4 \\ \hline \end{array}$$
$$=$$
$$\begin{array}{|c|c|c|c|}\hline A & Q & A & L_2 \\ \hline \end{array}$$
$$\oplus$$
$$\begin{array}{|c|c|c|c|}\hline KS_1 & KS_2 & KS_3 & KS_4 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|c|}\hline G_1 & G_2 & G_3 & G_4 \oplus Q \\ \hline \end{array}$$
$$=$$
$$\begin{array}{|c|c|c|c|}\hline A & 0 & A & L_2 \oplus Q \\ \hline \end{array}$$
$$\oplus$$
$$\begin{array}{|c|c|c|c|}\hline KS_1 & KS_2 & KS_3 & KS_4 \\ \hline \end{array}$$

If the set bit in Q is not set in L_2 - I get $L_2 \oplus Q = L_2 + Q$ and so the data is equal and I get the same $SHA3$ from the oracle for both cases.

If the set bit in Q is set in L_2 - I get $L_2 \oplus Q \neq L_2 + Q$ and so the data is different and I get a different $SHA3$ answers.

This way I found all length bytes bit-by-bit:

```
eq_byte, final_xor = value_bytes_xors[value_index], 0
for check_bit in range(8):
        size_change = 2 ** check_bit

        first_send_data = xor_in_index(local_enc_data[:],
                              chr(size_change), change_index - 1)
        first_send_data = xor_in_index(first_send_data,
                              chr(eq_byte), change_index)
        sha3_data_first = get_oracle_answer(first_send_data)

        second_send_data = xor_in_index(local_enc_data[:],
                              chr(size_change), change_index + 1)
        second_send_data = xor_in_index(second_send_data,
                              chr(eq_byte), change_index)
        sha3_data_second = get_oracle_answer(second_send_data)

        if sha3_data_second != sha3_data_first:
                final_xor = final_xor ^ size_change
```

The complete flag:

CTF{LLLLLLLLL___EEEEE____RRRRRRRRRR_OYYYYYYYYYY_JEEEEEEE ENKKKINNSSS}