NSUCRYPTO 2020: AES-GCM

Ioan Dragomir[1], Gabriel Tulba-Lecu[2], and Mircea-Costin Preoteasa[3]

[1] `ioandr@gomir.pw` – Technical University of Cluj-Napoca
[2] `gabi_tulba_lecu@yahoo.com` – Polytechnic Univeristy of Bucharest
[3] `mircea_costin84@yahoo.com` – Polytechnic Univeristy of Bucharest

Table of Contents

1 Problem summary

This problem revolves around the AES-GCM-256 encryption scheme, asking the solver to perform several exploits that arise from the misuse of it, we shall point out that all the attacks are due to nonce reuse:

Question 1: Given several encrypted messages and the plain-text for one of them, decrypt at least one other message.

Question 2 Given two messages that use the same key and the same nonce, create a message forgery.

Question 3 Keep the construction from **Q2**, with the exception that the Additional Authenticated Data (**AAD**) has now 8 unknown bytes $X = f(K)$, for some deterministic function f and the key K.

Question 4 Given two messages that use the same key and the same nonce, create a message forgery, such that, the decrypted plain text shall be human readable.

2 Solution

2.1 The structure of AES-GCM

We will start by analyzing a propriety of the GCM mode of operation:

$$C_i = P_i \oplus E_K(cnt_i) \quad \forall i \in [1, \ldots, n]$$

Where, cnt_i is the i^{th} value derived from the IV and n is the number of blocks in the plain text.

From this, we can see that GCM acts very similar to a stream cipher. It derives from an IV, a stream, and proceeds XOR-ing the plain text, with the generated stream. Therefore, if two messages are encrypted under the same IV and K, they will be XOR-ed with the same stream, so a two time pad attack can be applied.

2.2 Solution for Question 1

The first task provided the following information:

1. 8 encrypted messages.

2. The plain text for the first message = "Hello, Bob! How's everything?"

And asked for the decryption of another plain text. We will begin by performing some reconnaissance on the encrypted messages given. Using a parser written in Python [A], we will separate the various parts of them (encoded in hex):

Message Number	Header	IV	Authentication Tag	Encrypted Payload
0	923488fa5bda1b88	963ea616e2aeaf3e9b675ecf	b5607a45f8255c9874cfc87bb70c32f8	67aada75fa114093bb135d0e21cc73f45f65e8390aca31da97d68ab436
1	fe2a5e45f5d25913	eb9255003717c7dcea8dd805	9e50e2a4f4b2974271301be0e62bede9	ab4f3df8273b70e3c97153417224e84709af
2	9d501a5290c259f7	c5e80ee2151bafafda16475a	4a674244d125713ab6fb12ab10649103	a64276c20d541eb67628aac285d4e791e6fe950c3c3886e246
3	89323cbdda2edd8d	14616e6cedda18beefccdd82	036e0fc5e8b4ed98e6fb5ab7ea751226	9e8e99fb580e6feb98fc36d4ea49285466
4	b09c69d1ea5d3d10	fa1adf74f5f48c01887821e3	48ae7dfd21f43a6e2ee1adb00f38e20c	12a4ec16c1262c680e1c0e506f85ce3083e24dad5345f0e4a2ffa5
5	42e76d8f00a5ed7a	963ea616e2aeaf3e9b675ecf	29b5ebab8685b8bb4a0bec18d8f5d50c	63a6d87afa510ef184100e45458335e31674b861
6	f54c87dd01d3d9a9	963ea616e2aeaf3e9b675ecf	6609632220fd69c2fe4c5fc1f5f1f13b	61a0c56df45107b8b551155d49c224b65820ff210ed468c390cb8db529e69c44ed3e7dde41dc20fd85
7	26cfefbc3df253ea	c5e80ee2151bafafda16475a	f115d23cd40ef8adee22f862a79c5d04	a10d3bda0b424daf6667add6cbdae1d0e9e6d95b

From the table we observe that messages 5 and 6 share their IV with message 0, for which we know the plain text. Furthermore, message 5 has a smaller length than that of message 0. From this, we can apply the idea presented in the previous section:

$$\begin{aligned}
&C_0 = P_0 \oplus \text{stream}(IV, K) \implies \\
&\text{stream}(IV, K) = P_0 \oplus C_0 \\
&C_5 = P_5 \oplus \text{stream}(IV, K) \implies \\
&P_5 = C_5 \oplus P_0 \oplus C_0
\end{aligned}$$

Since we know C_5, P_0, C_0, we can decrypt message 5. This message turns out to be: **Lincoln Park, 10:15.**. As a side note, we can only partially decrypt message 6, because we don't know all of the stream that it was XOR-ed with. The partially decrypted message 6 is: **Nostalgia is a eternal motif**.

2.3 Preforming message forgery

In this section we will present an attack that will solve the remaining 3 questions. This attack fully breaks AES-GCM's integrity, when a nonce is reused, being able to compute authenticated tags for an arbitrary cipher text. Keep in mind that all the following operations are over the field $GF(2^{128})$.

Let's start by making some notations:

$$\begin{aligned}
&A = Header|IV = A_1|A_2\\
&C = C_1|C_2|\dots|C_n\\
&L = len(A)|len(C)\\
&S = E_K(cnt_0)\\
&H = E_K(0)
\end{aligned}$$

Now we can express the authentication tag as the following sum:

$$tag = A_1 \cdot H^{n+3} + A_2 \cdot H^{n+2} + C_1 \cdot H^{n+1} + \dots + C_n \cdot H^2 + L \cdot H + S$$

Does this look familiar? Well, it should, it is a polynomial, evaluated at point H.

Let's continue by looking at what we don't know. We do not know S and H. Since S is the constant term of the polynomial and it depends only on IV and K, we can get read of it if we have a nonce reuse. Let P_1 and P_2 be polynomials for the tag of two distinct messages, that use the same IV and K:

$$\begin{aligned}
&P_1(X) = A_1 \cdot X^{n+3} + A_2 \cdot X^{n+2} + C_1 \cdot X^{n+1} + \dots + C_n \cdot X^2 + L \cdot X + S\\
&P_2(X) = A'_1 \cdot X^{n+3} + A'_2 \cdot X^{n+2} + C'_1 \cdot X^{n+1} + \dots + C'_n \cdot X^2 + L' \cdot X + S\\
&P(X) = P_1(X) + P_2(X) + tag_1 + tag_2 = (A_1 + A'_1) \cdot X^{n+3} + (A_2 + A'_2) \cdot X^{n+2} +\\
&(C_1 + C'_1) \cdot X^{n+1} + \dots + (C_n + C'_n) \cdot X^2 + (L + L') \cdot X + tag_1 + tag_2
\end{aligned}$$

Since $P(H) = 0$, then H is a root of the polynomial so we can recover it by finding the roots of $P(X)$. After finding H, we can find S from the following relation:

$$S = P1(H) + tag_1$$

Now that we know how to find both H and S, that means we can compute tags for arbitrary cipher texts. We can now move to applying this attack and solving the remaining questions.

2.4 Solution for Questions 2 and 4

In order to find a solution for Question 2, we simply have to apply the algorithm described in the previous section. As we have mentioned above, being able to forge authentication tags for an arbitrary cipher text, allows us to modify any bit from a message. Furthermore, since we want to also answer the Question 4, we will make the following observation.

$$\text{Human readable text has ASCII codes } \in [32, \ldots, 127]$$

This means that, for a human readable byte if we only flip the least significant 5 bits, the newly obtained byte will still be human readable.

Thus we have an answer for Question 4: If we only modify the least significant 5 bits of a byte in the cipher text, then the plain text will still be human readable. In our implementation of the attack, as a proof of concept we only changed the least significant bit of every 4th byte. The new cipher text(including the header, IV and tag) we obtained is:

```
fff6ab48fa2d6c404aec77d7184536520e4e2
04b2f84d173ba889601957cb45420f7bffb99
50620360da94bfe55da98194e06a233ad34db
aa9c54b8fdd03f96c0a808e32aedafcd8b213
5389a080b9831058738abc9cea54fcb3945cb
9abb4973be2f4e769ca5e78899b4a8b5c0195
df1aba3b55b8a12acdb434e3b95225f4d94d0
80509ee8a543670a9a5f80a0c716c0dafcef9
d72dcb9ec42015c530bbbe11b4a8e2cb8a42f
60f1aead93aa2ddd113f4e106243dc2c4b421
ec5f01aa805f8b0ae5eca6f5f30939482f04f
986c812588b7cd93b50c21ade66512cbe86af
63d6aed783cc750416d06eda8c8426269eaf3
ab307b2fa100cac880ffe6b73131d91acea36
6a65d1905cb3680b1ee69b5cc215522775672
6ce5242f851d4aa2ddc19a9fff9594edb1791
```

2.5 Solution for Question 3

The difference between Question 3 and Question 2 is the Additional Authenticated Data. Question 3's AAD has 8 unknown bytes derived from the key, appended to the AAD of Question 2. This could be a problem but it fortunately isn't. The organisers have chosen the two messages that have the same nonce such that they have the same length too!

This means that the tag polynomials $P_1(X)$ and $P_2(X)$ have the same degree and the newly added value $X = f(K)$ in the AAD is on the same position so it will cancel itself in the context of a nonce reuse, also this value is dependent only on the key K. Thus, we can think of this new added value as another part of the constant term S, and hence the attack from Question 2 can be applied here as well. So after adding 8 "0" bytes (any value would work here since the values will cancel each other) to the original AAD, we can apply the exact same algorithm as in Question 2. The new cipher text(including the header, IV and tag) we obtained is:

```
047b8c4ede1735befe4deb8b8e8212a00bddd1ec21c731

2b93dc64db60ce9454827c3bb7fac7b490d89bec777dff

ca91744251e2e6288c586172ad5c3ea9481d17644a5e6a

9f86219482bc1f898f1c592e40d16d9446b9f343319d78

5d7e2ac34be60fbbddc9943df5bc1cd5867ff7557982f2

d1960a8d5ed293bffb48cc8b99609684dc211d3d537b1a

51b6173c7cc036096663e193acacc56b3a8cb0a066544a

b69820a1799545459a369b71aa9870b4595ccc6856e37a

4aa6c94fc6a1309d0a8c547ce9b2ccd8769037f5adc2c0

65a9405f396c00b6adea76faa8280c064bf9ad1eb0c883

27522448a6f002541566f30e690bfcd5d90a0ac81f64b8

9bafda35aa70f985051fd2d1917377d5e64021db9ad463

e6d6c910e4aedacf54d768a14de2ebd2296c08697d04f3

f52d600a72ea96bb25d4201fac9b058b6bbd086253dc52

81eaba723d49a0bafb58f4eaf31c143502d2b80532bfc1

1e736db95c5f78a08e21e0
```

2.6 Implementation

We have implemented this attack in C++, using the libraries GMP and NTL. The source code for the attack (**utils.cpp, utils.h & main.cpp**) has been attached to this solution. Also some scripts written in Python3 for testing the correctness of our solution have been attached.

A A

```
import binascii


def xor(a, b):
    o = b''
    for x, y in zip(a, b):
        o += bytes([(x ^ y)])
    return o


def split_message(msg):
    header = msg[:8]
    IV = msg[8:20]
    tag = msg[-16:]
    enc = msg[20:-16]
    return (header, IV, enc, tag)


def read_messages():
    msgs = []
    for i in range(8):
        msg = open("{}.message".format(i), 'rb').read()
        msgs.append(split_message(msg))
    return msgs


def print_message(msg):
    header, IV, enc, tag = msg
    print('header: {} IV: {} tag: {} enc: {}'.format(binascii.hexlify(heade
                                                     binascii.hexlify(tag),


msgs = read_messages()
for msg in msgs:
    print_message(msg)

known_IV = binascii.unhexlify('963ea616e2aeaf3e9b675ecf')
msg_0 = b"Hello, Bob! How's everything?"
enc_0 = binascii.unhexlify('67aada75fa114093bb135d0e21cc73f45f65e8390aca31da97d6
known_stream = xor(msg_0, enc_0)

for msg in msgs:
    header, IV, enc, tag = msg
    if IV == known_IV:
        print(xor(enc, known_stream))
```