Invisible Salamanders in AES-GCM-SIV

By **Sophie Schmieg < https://keymaterial.net/author/sophieschmieg/>**

September 7, 2020 < https://keymaterial.net/2020/09/07/invisible-salamanders-in-aes-gcm-siv/>

By now, many people have run across the **Invisible Salamander < https://eprint.iacr.org/2019/016.pdf>** paper about the interesting property of AES-GCM, that allows an attacker to construct a ciphertext that will decrypt with a valid tag under two different keys, provided both keys are known to the attacker. On some level, finding properties like this isn't too surprising: AES-GCM was designed to be an AEAD, and nowhere in the AEAD definition does it state anything about what attackers with access to the keys can do, since the usual assumption is that attackers don't have that access, since any Alice-Bob-Message model would be meaningless in that scenario.

What is interesting to me is that this property comes up more often than one would think, I ran across it several times now during my work reviewing cryptographic designs, it's far from an obscure property for real world systems. The general situation these systems have in common is that they involve three parties: Alice, Bob, and Trent. Trent is a trusted third party for Bob, who is allowed to read messages and scan them, with details like when and why depending on the crypto system in question. While Trent and Bob agree on the ciphertext—say because Trent hands Bob the ciphertext or because Alice presents Trent's signature on it—Alice has the option of giving Trent and Bob different keys. The challenge for Alice is to come up with a ciphertext that has a valid authentication tag and still decrypts to different messages for Trent and Bob.

Mitigations

Before I dive deeper into how to construct invisible salamanders for AES-GCM and AES-GCM-SIV, a few words on how to defend against these problems. The easiest option here is to add a hash of the key to the ciphertext. This technically violates indistinguishability, as the identity of the key is leaked, i.e. an attacker now knows which key was used for the message. If indistinguishability is necessary, using the IV as

a salt for the hash works well, constructions like `HMAC-SHA-2(key=IV, message=key)` (i.e. aka HKDF-expand) work well here, as long as attention is paid on whether or not this key hash can be used in any other context. In general, it shouldn't because the key already should only be used for AES-GCM/AES-GCM-SIV, but real world systems sometimes have weird properties.

Constructing Salamanders

With the mitigation out of the way, onto the fun part: Constructing the messages. In order to understand why and how these attacks work, we first have to talk about $\mathbb{F}_{2^{128}}$ and the way AES-GCM and AES-GCM-SIV use this field to construct their respective tags. As a finite field $\mathbb{F}_{2^{128}}$ supports addition, multiplication, and division, following the usual field axioms. The field has characteristic 2, which means addition is just the xor operator, and subtraction is the exact same operation as addition. Multiplication and division is somewhat more complicated and not in scope for this article, it suffices to say that multiplication can be implemented with a very fast algorithm if the hardware supports certain instruction sets (carryless multiplication). The division algorithm uses the Euclidean algorithm and will at most take 256 multiplications in a naive implementation, so while slower than the other operations, it will still be extremely fast. I will use $+$ for the addition operation and $\cdot$ for the multiplication operation. The most important caveat is to not confuse these operations with integer arithmetic.

AES-GCM

Next, on to **AES-GCM < https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>** . This AEAD is a relatively straightforward implementation of an AEAD that uses a UHF based MAC for authentication. Our IV is 12 byte long, we use a 4 byte counter and CTR mode to encrypt the message. The slightly odd feature is that we start the counter at 2, for reasons we will see later. For authentication, we first derive an authentication key H by encrypting the zero block (This is why we don't start the counter at zero, otherwise the zero IV would be invalid). Now, using the ciphertext blocks, additional data blocks (both padded with zeros as needed for the last block), and adding a special length block containing the size of the additional data and the ciphertext, we get a collection of blocks, all of which I will refer to as C_i. To compute the tag, we now compute the polynomial

$$GHASH(H, C, T) = C_0 \cdot H^{n+1} + C_1 \cdot H^n + \cdots + C_{n-1} \cdot H^2 + C_n \cdot H +$$

The constant term, T is the encrypted counter block associated to the counter variable of 1 (Which is why we started at 2 for the CTR mode). Remember that in characteristic 2 $+$ is xor, so we could equivalently say that we compute the polynomial without the constant term and then encrypt it with CTR mode as the first block.

Now, how do we get two different plaintexts to agree on both ciphertext and tag, we first choose two keys and produce the corresponding keystreams, choosing the plaintexts so that the ciphertexts agree (If you want two plaintext that make sense, this part is the hardest step, you first brute force the first few bytes in order to be valid in one format and a comment opening statement in the other, so that you can switch which parts of the ciphertext will appear as valid plaintext and which parts appear as commented out). We leave one ciphertext block open for now, as a sacrificial block that we will modify in order to make the tags turn out to be the same. Next derive the corresponding authentication keys H_1 and H_2 and our constant terms T_1, T_2. This means, we have C_i fixed, except for a specific index, say j, and can now solve

$$GHASH(H_1, C, T_1) = GHASH(H_2, C, T_2)$$

$$\sum_{i=0}^{n} C_i \cdot H_1^{n+1-i} + T_1 = \sum_{i=0}^{n} C_i \cdot H_2^{n+1-i} + T_2$$

$$C_j \cdot \left(H_1^{n+1-j} + H_2^{n+1-j}\right) = \sum_{\substack{i=0 \\ i \neq j}}^{n} C_i \cdot \left(H_1^{n+1-i} + H_2^{n+1-i}\right) + T_1 + T_2$$

$$C_j = \left(H_1^{n+1-j} + H_2^{n+1-j}\right)^{-1} \cdot \left(\sum_{\substack{i=0 \\ i \neq j}}^{n} C_i \cdot \left(H_1^{n+1-i} + H_2^{n+1-i}\right) + T_1 + T_2 \right)$$

by solving for the sacrificial block C_j.

AES-GCM-SIV

So far so good, but, what about **AES-GCM-SIV < https://tools.ietf.org /html/rfc8452>** ? GCM is famous for having many weird properties that make it extremely fragile, like leaking the authentication key on a single IV reuse, or allowing for insecure tags smaller than 128 bits. In many ways, AES-GCM-SIV is how AES-GCM should look like for real world applications, much more robust against IV reuse, only revealing the damaging properties of an UHF with a reused IV if both IV and tag

are the same. This is accomplished through using the tag as a synthetic IV, meaning the tag is computed over the plaintext, and then used as IV for CTR mode to encrypt. Even though this kind of SIV construction uses MAC-then-Encrypt, they are secure against the usual downsides due to CTR mode always succeeding in constant time, independent of the plaintext. This means the receiver can decrypt the message and validate the tag without revealing information about the plaintext in case of an invalid tag. The library needs to take care that the plaintext is properly discarded and not exposed to the user in case the tag does not validate.

The actual IV for AES-GCM-SIV is used primarily derive a per message key. This means that if the IV of two messages is different, both encryption and authentication keys will be unrelated and can not be used to infer things about each other.

All in all AES-GCM-SIV works like this:

- $H, K_E = \mathrm{KDF}(K, IV)$
- $T = \mathrm{AES}(K_E, P_0 \cdot H^{n+1} + \cdots + P_n \cdot H)$
- $C = \mathrm{AES\text{-}CTR}(K_E, IV = T)$

where the plaintext blocks P_i again contain additional data and length, and some extra hardening and efficiency tricks having been stripped for clarity.

Our previous approach of first creating the ciphertext and then balancing things out to get the tags to agree clearly cannot work here anymore. The keystream, and therefore the ciphertext, depend on the tag, so if we want to have any chance of finding a salamander, we have to fix the tag before we do any calculation at all. So after having chosen T, we decrypt it under each of our keys to get the result of our polynomial $S_i = \mathrm{AES}^{-1}(K_{E,i}, T)$. What we are left with is finding plaintexts P_1, P_2 such that

$$S_i = \sum_{j=0}^{n} P_{j,i} H_i^{n+1-j}$$

which gives us a system of two linear equations with $2n$ unknowns. But this isn't all constraints we need to satisfy, since we still need to encrypt these plaintexts once we have the tag balanced. Here, we are lucky that everything is over characteristic 2: The CTR encryption is just an addition of the plaintext and the encrypted counter block $C_i = \mathrm{AES}(K_E, CB_i) + P_i$. To say that two plaintexts result in the same ciphertext under two different keys is just fulfilling the equation

$$\mathrm{AES}(K_{E,1}, CB_{j,1}) + P_{j,1} = \mathrm{AES}(K_{E,2}, CB_{j,2}) + P_{j,2}$$

.

This, like our two equations for the tag, is a linear equation. So in the end, for a plaintext that has a size of n blocks, we get $n + 2$ linear equations with $2n$ variables. This means, in almost all cases, we can construct an invisible salamander with only adding two sacrificial blocks, with the same caveat that the two plaintexts need to be partially brute forced.

Test Code

I've put this to the test and have written code to produce **AES-GCM < https://github.com/sophieschmieg/fun-with-gcm/blob/master/FunWithGcm.java>** (Java) and **AES-GCM-SIV < https://github.com/sophieschmieg/fun-with-gcm/blob/master/fun_with_gcm_siv.cc>** (C++) salamanders.

 Up ↑