Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS

Dennis Gankin
EPFL, Switzerland / TUM, Germany

May 11, 2018

Abstract

This paper explains the dangers of nonce reuse in AES-GCM for TLS. It points out the importance of the nonce to guarantee authentication and explains different nonce generation methods. Further, the document focuses on an Internet scan that analyzes how common nonce reuse is in real life. As a conclusion the paper presents possible ways to prevent nonce misuse.

1 Introduction

The Internet's wide use creates the need for a secure communication, which is provided by the TLS and its underlying ciphers like AES-GCM. Though considered secure, AES-GCM's complicated specifications can lead to implementation mistakes and unwanted security flaws. To uphold the AES-GCM security it is crucial to use and create the nonce correctly. That is because nonce reuse eliminates the authentication property in TLS, opening up the possibility for the Forbidden Attack. An Internet scan by [1] in 2016 showed that around 70.000 devices were using insecure nonce generation algorithms. 184 Servers where acutely vulnerable to practical forgery attacks. The findings motivate to use new standards for authenticated encryption.

2 Transport Layer Security

The Transport Layer Security protocol (TLS) secures Internet connections: typically, the HTTP protocol. In TLS the handshake protocol sets up the cryptographic algorithms, validates both parties through digital certificates and generates a symmetric key. Older version of TLS were using RC4 and CBC ciphers for authentication and encryption, which proved to have major security issues. Thus, the currently widely used TLS 1.2 and the newest version 1.3 encrypt with AES-GCM.

2.1 AES-GCM

AES-GCM offers authenticated encryption with associated data (AEAD). It uses the Galois Counter Mode (GCM), an authenticated mode of operation and the Advanced Encryption Standard (AES), a block cipher.

Figure 1 shows how the encryption works. Given the message M, associative data A and initialization vector IV the sender computes a ciphertext C and a corresponding message authentication tag T. (IV, A, C, T) is sent allowing the receiver to verify the authenticity.
AES-GCM encrypts the plaintext using counter mode (CTR) and AES. AES encryption is denoted as E_K. The symmetric key K for AES is created in the TLS handshake protocol. Additionally, the hash function GHASH computes an authentication tag, based on multiplication over the Galois field $GF(2^{128})$:

$$GHASH_L(A, C) = \bigoplus_{i=1}^{\ell} L^{\ell-i+1} \cdot X_i, withX = A||0^*||C||0^*||enc_{64}(|A|)||enc_{64}(|C|)$$

The GHASH key L is computed by encrypting a block of zeros: $L = E_k(0)$. The initialization vector IV is concatenated with four bytes to be used for the counter mode.
The IV should only be used once and in TLS [3] it is composed of four bytes of salt and an eight byte long nonce: $IV = salt_{32}||nonce_{64}$. The nonce generation depends on the sender's implementation but it must ensure the nonce's uniqueness.

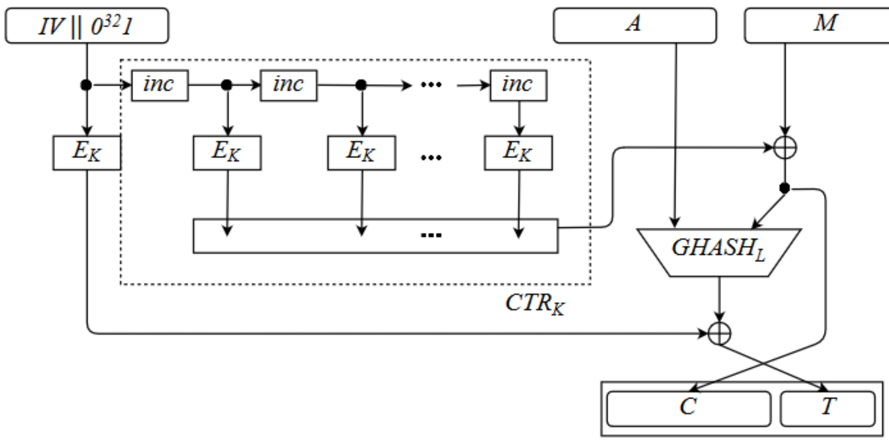

Figure 1: AES-GCM encryption

2.2 Nonce Reuse

A nonce is a number that can only be used once, meaning it should be unique. If this requirement is not met, the security guarantees cannot be met anymore.

Nonce reuse makes a **known plaintext attack** possible. As shown in figure 1 the received ciphertext is: $C = CTR_K(IV) \oplus M$. When knowing the plaintext M, one can retrieve the IV encrypted in counter mode: $CTR_K(IV) = C \oplus M$. If the IV is reused the retrieved $CTR_K(IV)$ will decrypt any other ciphertext with unknown messages.

Furthermore, with nonce reuse an adversary can mount a **forgery attack**, which is referred to as the Forbidden Attack by Joux [2]. The goal in this case is to derive the key used for GHASH. Given two sent messages M, M' that where encrypted with the same nonce one can build the following equation using XOR:

$$0 = \left(\bigoplus_{i=1}^{\ell} L^{\ell-i+1} \cdot (X_i \oplus X_i') \right) \oplus T \oplus T'$$

This polynomial is of degree ℓ and therefore has ℓ roots. Key L is one of those roots. So by factoring the polynomial above we obtain possibilities for L [5]. [1] use the Cantor-Zassenhaus algorithm for factoring with polynomial worst case complexity. After each nonce reuse another polynomial can be build and factored, providing a new set of possible keys. L is a common root of all created polynomials. Hence, after factoring two polynomials an intersection between the polynomial's roots will only contain one element, which is the key L.

Though this does not yield the symmetric AES key, it allows to forge messages where one can at least replace the original A by some forged A'. If additionally an adversary knows $CTR_K(IV)$ from a known plaintext attack he can forge the plaintex, too. In order to do so the attacker computes the new ciphertext $C' = C \oplus M \oplus M'$ and an according authentication tag $T' = T \oplus GHASH_L(A, C) \oplus GHASH_L(A', C')$.

All in all, to retain security nonces must never be reused which places importance on generating nonces correctly.

3 Nonce Generation

The TLS specification does not define a standard nonce generation algorithm. Some methods of nonce creation may lead to security flaws and should be avoided.

3.1 Secure Approaches

Using a **counter** is a simple and good way to safely create nonces. The initial number can start at any arbitrary value. A nonce will only be reused after the whole number range is exhausted. Thus, with an eight byte long nonce a value will only be reused after 2^{64} messages. The cryptographic libraries Botan 1.11.28, BouncyCastle Java 1.5 and SunJCE use that implementation starting with zero while OpenSSL 1.0.2g picks a random number as starting point for the counter.

A **Linear Feedback Shift Register (LFSR)** is a shift register whose input bit is a linear function of its previous state. It's security properties are similar to a counter. That is because it creates a reoccurring sequence of states and the number of states equals the number range. Therefore, the implementation is harder to reconstruct and the next nonce is harder to predict than with a counter.

3.2 Risky Approaches

Generating **random nonces** each time can be insecure if many messages are encrypted in one session using the same AES key. With only a few messages a nonce reuse is improbable but with 2^{33} sent messages the probability of a collision will reach over 86% due to the Birthday Paradox [1]. The Birthday Paradox states that in a set of size N only $O(\sqrt{N})$ samples are needed to find a collision with a constant probability. The collision probability with n uniformly picked samples can be approximated by $Pr[\exists i < j X_i = X_j] \approx 1 - e^{-\frac{n^2}{2N}}$ [4].
To exploit this, one would need to send at least a few terabytes of data [1] over one single connection which makes the attack rather unlikely because servers usually limit time and data over a single connection.

Wrong implementations pose much bigger risk than random nonces. Often mistakes in the code lead to unwanted behavior and possibly repeating nonces. Authors in [1] observed cases where the nonce generation method's return value was not checked which made the script run with uninitialized memory when an error was returned. This and many other mistakes can lead to nonce reuse unwillingly. Concluding, a correct implementation is crucial.

4 Real Life Attacks

In 2016 the authors of [1] conducted a search on the Internet to find out how well AES-GCM was implemented in practice. A patched version of OpenSSL and a C-program gathered nonces from around 48 million devices to check for duplicates.

4.1 Vulnerable Devices

The findings revealed 184 servers that reused nonces. While 70 devices only returned one duplicate before continuing with random nonces, 107 devices used the same number constantly. Those behaviors probably stem from faulty implementations. Some of those devices where owned by companies from the financial sector where security is crucial. Many other devices acted as load balancers, making it hard to find out the owner. The vulnerable parties have been informed and the financial institutions changed their implementations.

On top, approximately 70.000 devices generated seemingly random nonces which as shown above could also pose a low risk. While authors of [1] were not able to find nonce reuse in other cases they assume that there are probably more unnoticed vulnerable implementations.

4.2 A Practical Nonce Reuse Attack

Devices that reuse nonces can be practically exploited in a man in the middle attack.

We assume a server reusing nonces and an attacker controlling the local network. The adversary records all sent traffic from the server to the client, in particular he collects all nonces.
First, the authentication key is derived by factoring as described in subsection 2.2. Next, the adversary redirects the client to a static endpoint on the server to mount a known plaintext attack. That is because on a static site the payload, for instance the html-code, is known beforehand. Now the adversary is able to encrypt and forge any message as shown in subsection 2.2.
As a result, the attacker can inject any javascript into the site while the client seems to have a TLS secured connection to the server, hence trusting the site.

To conclude, if a man in the middle attack is possible, exploiting nonce reuse is simple and allows an adversary to mount a serious forgery attack and decrypt messages. That proves that nonce reuse is very risky in AES-GCM and can lead to fatal exploits.

5 Countermeasures

The findings of the Internet scan proved that without any concrete guidelines in the TLS specifications regarding nonce creation, AES-GCM can be implemented insecurely very easily. To prevent such mistakes one can either create nonces in a deterministic way or use cryptographic algorithms that do not fail under nonce reuse.

5.1 Secure Algorithms

[1] proposed ChaCha20-Poly1305 and AES-OCB for deterministic nonce creation. The nonce is constructed from the record sequence number and the shared secret in the TLS connection. As this information is known to both parties, the nonce does not have to be transmitted with each message. That saves a little bandwidth and excludes the possibility of random or faulty nonce creation. Including the record sequence number in the nonce is similar to using a counter for nonce creation and thus guarantees no nonce reuse.
Cryptographic algorithms that uphold security even with a nonce duplication usually use the "MAC then encrypt" approach. In those methods first the authentication tag is created from the message and then used as a nonce to encrypt the message. In that case nonce reuse is not an issue. As a consequence though, the message has to be processed twice while AES-GCM and other "encrypt then MAC" approaches first encrypt and then derive the authentication tag from the encrypted data.

5.2 Changes in TLS 1.3

The TLS working group published the new TLS 1.3 standard [3] in March 2018 where among other changes they address the AES-GCM insecurities due to nonce misuse. TLS 1.3 forbids all old cipher suites that were proven insecure but it continues to use AES-GCM. Additionally ChaCha20-Poly1305 was also added as an AEAD algorithm.

As AES-GCM stays in the specifications nonce reuse is still an issue. That is why the new cipher specifications explain correct nonce use and generation in more detail than before and thus offer some guidelines for programmers. Still, the nonce generation is not done in a deterministic way as in ChaCha2-Poly1305.

6 Conclusion

Even though AES-GCM is considered secure it depends highly on correct nonce use. Duplications and wrong nonce generation open up the possibility for a man in the middle attack where the adversary can forge messages and even get the plaintext in some cases.
Added guidelines for nonce creation in the new TLS standard try to prevent human errors. An even safer approach would be to eliminate the possibilities of those errors. That is easily done with deterministic nonce creation in ChaCha20-Poly1305 which is also added in TLS 1.3. Such a deterministic nonce creation could also have been added to AES-GCM. Nevertheless, AES-GCM is kept as it is in the new TLS standard. This preserves the possibility of an insecure implementation.

All in all, AES-GCM is a theoretically secure cipher that relies on nonces. Though TLS 1.3 gives a few guidelines for nonce generation, faulty implementations cannot be excluded. That is why ChaCha20-1350 is in most cases a safer option with TLS 1.3.

References

[1] Böck, H., Zauner, A., Devlin, S., Somorovsky, J., and Jovanovic, P. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. https://eprint.iacr.org/2016/475.pdf [Accessed 2018-05-01].

[2] Joux, A. Authentication failures in NIST version of GCM. https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_comments.pdf [Accessed 2018-05-01].

[3] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft, Mar. 2016. https://tools.ietf.org/html/draft-ietf-tls-tls13-12.

[4] Vaudenay, S. Cryptography and security lecture notes. https://moodle.epfl.ch/pluginfile.php/1633943/mod_resource/content/4/lecturenotes2017.pdf [Accessed 2018-05-10].

[5] Vaudenay, S., and Vizár, D. Under pressure: Security of caesar candidates beyond their guarantees. Cryptology ePrint Archive, Report 2017/1147, 2017. https://eprint.iacr.org/2017/1147.