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

Hanno Böck[*] Aaron Zauner[†‡] Sean Devlin[§]
Juraj Somorovsky[¶] Philipp Jovanovic[‖]

July 25, 2016

Abstract

We investigate nonce reuse issues with the GCM block cipher mode as used in TLS and focus in particular on AES-GCM, the most widely deployed variant. With an Internet-wide scan we identified 184 HTTPS servers repeating nonces, which fully breaks the authenticity of the connections. Affected servers include large corporations, financial institutions, and a credit card company. We present a proof of concept of our attack allowing to violate the authenticity of affected HTTPS connections which in turn can be utilized to inject seemingly valid content into encrypted sessions. Furthermore, we discovered over 70,000 HTTPS servers using random nonces, which puts them at risk of nonce reuse, in the unlikely case that large amounts of data are sent via the same session.

1 Introduction

The Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM), or short: AES-GCM [25, 6], is currently the most widely used cipher for symmetric (authenticated) encryption in the TLS protocol [4]. This came as a consequence of the exposure of various weaknesses in many alternative symmetric TLS ciphers during the past few years. The CBC mode was affected by a whole series of attacks, including BEAST [5] (affecting TLS 1.0), Lucky Thirteen [1] (affecting all versions, based on timing side-channels and the older Vaudenay attack), POODLE [26] (only affecting SSLv3) and POODLE-TLS [23] (implementation bugs). All those attacks did not exploit weaknesses of CBC per se, but took advantage of the particular way how

[*] https://hboeck.de, hanno@hboeck.de
[†] SBA Research gGmbH, azauner@sba-research.org
[‡] lambda: resilient.systems, azet@azet.org
[§] Independent, seanpatrickdevlin@gmail.com
[¶] Horst Görtz Institute for IT Security, Ruhr University Bochum, juraj.somorovsky@rub.de
[‖] École Polytechnique Fédérale de Lausanne (EPFL), philipp.jovanovic@epfl.ch

CBC was deployed in TLS (implicit IVs, lack of strict padding and MAC-then-Pad-then-Encrypt). Before TLS 1.2, the RC4 stream cipher was the only alternative to CBC-based ciphers but it had been long known for its weaknesses [9] and eventually came under attack in TLS [2] as well. The attacks against CBC-based ciphers can be mitigated by careful implementations, however it has been shown that these mitigations are extremely difficult to implement correctly [34]. It is not possible to mitigate the weaknesses in RC4. The cryptographic community concluded that both CBC and RC4 should be avoided and later even prohibited use of RC4 in TLS [29] entirely.

With AES-GCM the TLS standard provides only one widely available alternative to CBC and RC4. Technically, there are other options, such as CCM and GCM in combination with block ciphers like Camellia, but all of them lack widespread support. The OCB mode [21], which by many is considered superior to GCM, suffered from patenting issues for a very long time. Those problems prevented its wide deployment and only got resolved recently [36]. Another promising alternative is the ChaCha20 stream cipher in combination with Poly1305 as authenticator which was released as an RFC [27] in 2015 and a specification as a TLS cipher mode will soon be available as an RFC [22].

Despite currently being the most popular TLS cipher, AES-GCM is not well received by the cryptographic community. Niels Ferguson described potential attacks on GCM with short authentication tags [8], Antoine Joux published a critical comment during the standardization process of GCM [19], and several other cryptographers recently described GCM as "fragile" [28, 12].

Many stream cipher-based crypto algorithms take, next to a secret key and a message, a so-called nonce as an additional input. Varying the nonce allows to generate multiple distinct cipher streams under the same secret key and thus multiple messages can be encrypted safely. If, however, the same nonce-key pair is ever repeated for different messages, an attacker is able to learn the XOR of the plaintext messages by XORing the corresponding ciphertexts. This works since the used cipher streams are identical (due to the same nonce-key pair) and therefore cancel each other out. In summary this leads to a violation of the confidentiality of the affected messages.

Internally, GCM uses a counter mode-like construction and thus suffers from the above issues as well. In the nonce reuse scenario, however, there is yet another vulnerability on which we focus in this work: the "forbidden attack" by Joux [19] exploits nonce reuse to reconstruct the authentication key. This then leads to efficient forgery attacks enabling the creation of seemingly valid ciphertexts without knowledge of the secret master key. In this paper we first show that several TLS implementations are vulnerable to nonce reuse attacks since they use repeated nonces after a few server messages. Second, we present results from our Internet wide scan that identified more than 70,000 potentially vulnerable servers. These servers generate nonces at random, which makes the nonce reuse attacks possible after sending a large amount of TLS records. For example, a server is vulnerable with a 40% probability after sending 2^{32} TLS records (see Table 1). Our results motivate for the standardization of algorithms that resist nonce misuse and the publication of errata on IETF documents with insufficiently secure nonce generation methods.

2 Background

In the following we briefly recap AES-GCM [25, 6] and its application in TLS [4]. We note that the AES-GCM specification allows for different initialization vector or authentication tag lengths. We only concentrate on the version of AES-GCM as it is used in TLS. The description applies to AES-128 and AES-256 since both have equal input/output lengths.

2.1 AES-GCM

AES-GCM [6] is a block-cipher mode of operation which provides authenticated encryption with associated data (AEAD). It uses counter mode to encrypt plaintexts. The resulting ciphertext is authenticated using a hash function called GHASH which is based on a computation over the Galois field GF(2^{128}).

We use the following notation:

$a \parallel b$	Concatenation of strings a and b.
0^s	String consisting of s zero bits.
P_i	The i-th plaintext block.
C_i	The i-th ciphertext block.
A_i	The i-th block of additional authenticated data.
IV	Initialization vector consisting of 96 bits (12 bytes).
cnt	4-byte long counter value.
J_i	The i-th counter block, computed using concatenation of the IV and the counter value cnt, $cnt = (i+1) \mod 2^{32}$, to achieve 128 bits. $J_0 = IV \parallel 0^{31} \parallel 1$.
$Enc_k(X)$	AES encryption of block X, with symmetric key k.
$Gmul_H(X)$	Multiplication $H \cdot X$ in Galois Field GF(2^{128}), with the irreducible polynomial $f = 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128}$.
T	Authentication tag.
$len(X)$	Bit-length of string X, represented by 64 bits.

The AES-GCM encryption process of a message consisting of n blocks works as follows (see Fig. 1):

1. The encryptor generates a 96-bit long initialization vector IV.
2. She generates 128-bit long counter blocks J_i, where $J_i = IV \parallel cnt$ and $cnt = (i+1) \bmod 2^{32}$, for $i \in \{0, \ldots, n\}$.
3. She computes the i-th ciphertext block as follows: $C_i = Enc_k(J_i) \oplus P_i$. Note that the length of the last ciphertext block C_n is equal to the length of the last plaintext block P_n.

In order to generate the authentication tag T, the encryptor computes a GHASH over the additional authenticated data and the ciphertext:

1. The encryptor generates the hash key $H = Enc_k(0^{128})$.
2. Starting with $X_0 = 0$, she computes Galois field multiplications over the additional authenticated data consisting of m blocks (note that the last block is padded with zeros to achieve a 128-bit block length):

$$X_i = Gmul_H(X_{i-1} \oplus A_i), \text{ for } i \in \{1, \ldots, m\} .$$

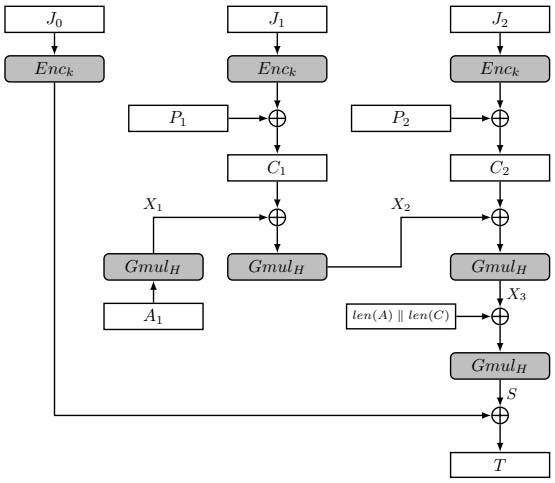

Figure 1: AES-GCM encryption computed using two plaintext blocks and one block of additional authenticated data [25].

3. She executes Galois field multiplications over n ciphertext blocks:

$$X_{i+m} = Gmul_H(X_{i+m-1} \oplus C_i), \text{ for } i \in \{1, \ldots, n\} .$$

4. She executes the last multiplication using the bit-lengths of A and C:

$$S = Gmul_H(X_{m+n} \oplus (len(A) \parallel len(C))) .$$

5. Finally, the encryptor computes the authentication tag:

$$T = S \oplus Enc_k(J_0) .$$

The final output of this function is the ciphertext concatenated with authentication tag: $C \parallel T$. We refer to [6] for more details.

2.2 AES-GCM in TLS

TLS may use AES-GCM to encrypt and authenticate data in the record layer. In TLS the maximum record size is on the order of 2^{14}. Encryption keys and further key material are derived during the TLS handshake phase [4]. The TLS handshake procedure is not relevant to our attack. It is only important to know that the output of a TLS handshake is a `master_secret`, which is used to derive further key material, including the `server_write_IV` and `client_write_IV`.

An input into AES-GCM encryption is a plaintext and a 12-byte long initialization vector IV. According to TLS, IV is constructed as follows:

- **Salt** (4 bytes) is derived during the TLS handshake and its value is equal to `server_write_IV` / `client_write_IV`. This is also called the implicit part of the IV.
- **Nonce** (8 bytes): A TLS peer must generate an eight byte nonce, also called the explicit part of the IV. It is up to the implementation to make sure the nonce is unique.

The initialization vector is then used to create a 16-byte long counter value. The counter value is incremented by 1 with each new ciphertext block, as described in the previous section.

3 The Forbidden Attack

In his comments to NIST Joux [19] described an attack against GCM if nonces are reused. This attack allows an attacker to learn the authentication key and forge messages. Because the uniqueness of nonces is typically a ground rule for cryptanalysis, Joux called his attack the "forbidden attack". Nevertheless, it highlights an important failure mode in real-world implementations.

3.1 Overview

Joux's attack takes advantage of the underlying mathematical structure of the GHASH primitive. Specifically, the computation of the tag T can be viewed as the evaluation of the following polynomial g at the authentication key H:

$$\begin{aligned} g(X) = A_1 X^{m+n+1} + \cdots + A_m X^{n+2} \\ + C_1 X^{n+1} + \cdots + C_n X^2 + LX + Enc_k(J_0) \end{aligned}$$

Where L is a block encoding the lengths of A and C, and $Enc_k(J_0)$ is a secret nonce-derived value. Note that all values other than $Enc_k(J_0)$ are known to the attacker. Evaluating at H, we have:

$$g(H) = T$$

To understand Joux's attack, consider the case where two messages are encrypted under the same nonce. For simplicity, let us suppose two messages each with a single block of ciphertext and no blocks of additional authenticated data. We have:

$$g_1(X) = C_{1,1} X^2 + L_1 X + Enc_k(J_0)$$

$$g_2(X) = C_{2,1} X^2 + L_2 X + Enc_k(J_0)$$

Recall that addition in the field is equivalent to `XOR`. Knowing that $g_1(H) = T_1$ and $g_2(H) = T_2$, we modify each polynomial by adding the known tag. We now have

$$g_1'(X) = C_{1,1} X^2 + L_1 X + Enc_k(J_0) + T_1$$

$$g_2'(X) = C_{2,1} X^2 + L_2 X + Enc_k(J_0) + T_2$$

with $g_1'(H) = g_2'(H) = 0$. Note that since $Enc_k(J_0)$ is a nonce-derived value, it is common to both polynomials. Adding these polynomials, we obtain

$$g_{1+2}'(X) = (C_{1,1} + C_{2,1}) X^2 + (L_1 + L_2) X + T_1 + T_2$$

which is fully known to the attacker and satisfies $g_{1+2}'(H) = 0$. Since H is a root of g_{1+2}', we can factor the polynomial to recover a short list of candidates for the authentication key. Although the list of candidates may be as long as the degree of the polynomial, in practice it is usually relatively short. If further nonce reuse is detected, additional polynomials sharing a common root in the authentication key can be constructed. Factoring these polynomials and finding the common root yields the correct value for H.

This results in catastrophic failure of authenticity, even if a nonce is only re-used a single time and enables us to carry out a practical forgery attack against HTTPS as described in Section 6.

Table 1: Nonce collision probability p after 2^n nonces of 64 bit length.

n	p	n	p
22	0.000000	29	0.007782
23	0.000002	30	0.030767
24	0.000008	31	0.117503
25	0.000031	32	0.393469
26	0.000122	33	0.864665
27	0.000488	34	0.999665
28	0.001951	35	1.000000

3.2 Nonce Generation in TLS

In TLS, GCM requires a 96-bit nonce where 32 bits are derived along with session key material and remain static for the duration of the session and the other 64 bits are transmitted explicitly in each record. As highlighted in Section 3.1, nonce uniqueness is essential for GCM's security. However, the TLS specification does not provide clear guidelines to developers how to choose the 64-bit explicit nonce [24].

The easiest secure way is a counter. Given that the nonce value is 64-bit long, a repeating nonce will only happen after 2^{64} TLS records. It is not realistic that many encryptions happen over a single TLS connection. There are two counter variants in widespread use: Some implementations start the counter with zero and increment from there, others start with a random value and increment from it (e.g., OpenSSL). Both variants are equally secure. Another option would be a negative counter, but we have not observed this in practice.

3.3 Duplicate Nonces

Faulty implementations may send duplicate nonces, e.g. by always sending the same value as a nonce or repeating a nonce for two encryption operations. A single repeated nonce is usually enough to fully recover the connection's authentication key. In such faulty implementations, authenticity is lost and an attacker is able to manipulate TLS-protected content.

3.4 Random Nonces

A less clear risk is present if an implementor chooses to use random values as a nonce. If only a few TLS records are encrypted with the same key, then a random nonce does not pose a risk. However, if a large number of records is encrypted with the same key, the risk may become relevant. If choosing nonces at random after 2^{28} encryptions the probability of a nonce collision will be around 0,2 % due to the birthday paradox. After 2^{33} encryptions the probability will be more than 80 % (see Table 1).

The size of a TLS record is determined by many factors, therefore it is not trivial to calculate the exact amount of data necessary to generate a nonce duplication with an implementation with random nonces. It is however most likely in the area of Terabytes. There are probably few scenarios in which this is a problem. VPN networks may use the same connection for such a large number of TLS records. Also in an attack scenario where an attacker can control Javascript and the victim has a very fast Internet connection such an attack might be possible. However this

requires an HTTPS server that allows an unlimited number of requests over a single connection. Common HTTP server implementations usually limit the number of Keep Alive requests that can be sent over one connection, but this limit can be disabled.

We conclude that an attack on an implementation using random nonces is unlikely, but it cannot be definitely excluded. For safety reasons random nonces should be avoided and a counter should be used.

4 Internet-wide Survey (HTTPS)

Our evaluation of Internet connected devices has been split into multiple sub-tasks: an initial discovery scan, followed by vulnerability scans on discovered target devices with different parameters. The scans stretched a time span of approximately 18 days. In this section we describe the methods used for discovery and analysis of Internet connected devices during our evaluation.

All tooling is available for review and testing via this project's GitHub repository. Concrete information on used compilers, libraries, applications, and operating system is provided in Appendix A.

4.1 Host Discovery

We performed an IPv4 discovery scan using `masscan`[1] for TCP port 443 (HTTPS) starting on new-years eve 2016. With appropriate rate-limiting this scan took about two days and resulted in $48,406,453$ distinct IP addresses serving on TCP port 443. We limit `masscan`-runs to 75,000 pps (packets per second) to reduce strain on upstream carrier equipment and monitoring as agreed upon with our upstream ISP.

We honor blacklisting requests while conducting scans and thereafter, thus keep excluding CIDR blocks during renewed Internet-wide scans. As of May 4th 2016 our "blacklist" consists of a total of 558,608 IPv4 addresses.[2]

4.2 Vulnerability Testing

Between the 4th and the 17th of January 2016 we performed two vulnerability scans on the randomized set of all previously collected IPv4 addresses serving HTTPS. A patch to OpenSSL 1.0.2e and externally parallelized C-program were used to collect first 10 and then 100 nonces from the TLS handshake of these targets.

We have limited control over the number of connections a server uses. Our test tool was not ideal in this regard. Due to the HTTP Keep-Alive feature it is possible to send multiple HTTP requests over one connection. After reading content from the server we sent a second HTTP request without knowing whether the server would send more data. An ideal test tool would first read any content coming from the server, however to reliably do so one would have to fully implement HTTP, which is not trivial. Our initial scan tool required a patched OpenSSL version. We later created a tool that uses an OpenSSL callback, thus avoiding to patch OpenSSL itself, however in our experience this later tool turned out to be more error prone.

[1] `masscan` is an open-source project under AGPLv3 license available from https://github.com/robertdavidgraham/masscan.

[2] Our "blacklist" is not publicly available nor shared due to obvious privacy implications. We therefore refrain from detailing on size and count of excluded net-ranges.

5 Vulnerable Devices

We found 184 devices that used a duplicate nonce. The behavior of these devices was mixed. 66 devices were using the value `0100000003001741` twice and then continued with a randomly chosen value and a counter starting from that value. Four further devices showed a similar behavior, but with other starting values (`010000000100c289`, `0100055f03010240` and `010000000080c0eb` twice). 84 devices used a random value for the first encryption and subsequently zero values. 23 devices simply always used zero. All of these devices can be practically attacked.

Given these different behaviors we assume we have at least four different kinds of faulty implementations.

We tried to contact the owners of affected devices, but ran into significant difficulties. Most of our contact attempts were not answered at all. The affected parties include several web pages of the credit card company VISA, the polish banking association ZBP (Zwizek Banków Polskich), and Deutsche Börse (German stock exchange).

With the help of CERT.at we were able to establish a contact with one affected device owner. We were able to determine that some devices were load balancers produced by Radware. We contacted Radware and explained the issue to them. The affected devices were using code or hardware from Cavium. Radware has since fixed the issue and published a security advisory [30]. The Radware device was using the "nonce" `0100000003001741` twice and then a counter starting from value chosen at random.

5.1 Missing Return Value Check in OpenSSL

Several further devices we observed were sending two values `0100000003001741` and similar other values. These look like uninitialized memory. We do not have a definitive answer why these devices behave like this, but we have some further observations. One device owner mentioned that the device is using a modified version of OpenSSL 1.0.1j. Given that we learned from Radware that their devices internally use a Cavium chip, we believe they use a modified OpenSSL version in combination with a hardware accelerator chip from Cavium. Thus, we checked whether we could find a plausible way how OpenSSL could generate this behavior.

The code that generates the first nonce value in OpenSSL calls the random number generation function `RAND_bytes()` to get eight random bytes. This is done in the function `aes_gcm_ctrl()` in `e_aes.c`. The error code of `RAND_bytes` is checked and `aes_gcm_ctrl()` returns 0 in case of an error. In `t1_enc.c a` call to `EVP_CIPHER_CTX_ctrl()` happens, which maps to `aes_gcm_ctrl()`. The return value is not checked here.

This means that in case the random number generator returns an error the code continues with uninitialized memory in the IV. It is possible that the devices showing this behavior use a hardware random number generator that is malfunctioning.

We simulated a failing random number generator by returning an error code if eight bytes were requested from this function. This deliberately broken OpenSSL variant sent the value `010000a60000012c` as a nonce, which is uninitialized memory. However unlike with the devices we observed in the wild we were unable to connect to that broken OpenSSL version. Current OpenSSL versions properly check the return value of the function `RAND_bytes()`, which was added by OpenSSL developer Matt Caswell in

February 2015.[3]

5.2 LFSR

We found a significant number of devices that, according to their Server HTTP header, were produced by Check Point. At first their nonce value looked random. However after contacting Check Point we learned that their TLS implementation is using a Linear Feedback Shift Register (LFSR) for the nonce generation. The LFSR is generated by the following polynomial:

$$X^{64} + X^{63} + X^{61} + X^{60} + 1 \,.$$

The chosen LFSR has maximal period of $2^{64} - 1$, i.e. if the LFSR is initialized with an arbitrary non-zero nonce then only after $2^{64} - 1$ updates the values start to repeat. This implementation therefore has the same security properties as a counter. We considered this in our analysis and excluded nonces using this LFSR. While this approach is unusual, there is no security risk associated with it. It can however be used to fingerprint devices.

There may be other implementations that have a randomly-looking nonce, but are in fact generated using an LFSR-like algorithm. However we are not aware of such devices at this time and welcome feedback from device manufacturers and software developers.

5.3 Random Nonces

After filtering the results for the LFSR used by Check Point there were approximately 70,000 devices left that had a random-looking nonce. Based on the title tags and HTTP headers we tried to identify the devices.

Based on the HTTP "Server" header around 7,700 devices were Lotus Domino installations. We disclosed the issue to IBM, the vendor of Lotus Domino. They confirmed the vulnerability and published an update [16].

Based on the title tag 19,120 hosts were devices by the company Sangfor, a chinese vendor of network equipment. We disclosed the issue to Sangfor, but never received any reply to our contact attempts.

Due to a contact of one affected server operator we were able to identify it as an A10 load balancer (model AX1030, OS version 2.7.2-P5). We disclosed the issue to A10 and they confirmed and fixed it [35].

A significant number of affected hosts identified themselves as Microsoft IIS in different versions in the "Server" header. The most common identification string was "Microsoft-IIS/7.5", which we found 9,633 times. We are unable to explain this. In our tests Windows/IIS installations did not show any suspicious behavior. We contacted Microsoft and they informed us that all versions of SChannel use a counter as a nonce. The most likely reason for this finding is that these hosts all had their TLS termination offloaded to a load balancer or firewall.

Based on our findings we must assume that there are potentially vulnerable implementations we were unable to identify.

6 A Practical Attack on Browser HTTPS

We implemented an attack to inject malicious content into browser-based HTTPS sessions. Our attack takes advantage of servers that repeat GCM nonces either by random chance or due to implementation errors.

[3] https://github.com/openssl/openssl/commit/eadf70d2c885e3e4e943091eabfec1e73d4f3883

Let C, S, and M denote the browser client, a vulnerable web server, and our man-in-the-middle (MitM) attacker, respectively. For the purposes of our attack, assume M controls the local network: they may observe all of C's traffic and modify or drop messages. We assume also that S exposes a faulty TLS implementation that will repeat nonces reliably.

The attack proceeds as follows:

1. M coerces C into loading attacker-controlled content. This can be done either via a phishing attack or by injecting malicious content into unauthenticated HTTP traffic.
2. M serves C HTML or JavaScript to initiate an HTTPS session with S. M observes the handshake to verify that a GCM cipher suite is negotiated. If not, M aborts the attack.
3. After S changes cipher suites following a successful handshake, M begins recording all server-sent traffic. In particular, M notes the following in each record:
   - The sequence number, a simple incrementing counter.
   - The record header comprising the first five bytes of the record.
   - The explicit nonce part comprising the first eight bytes of the record fragment.
   - The authentication tag comprising the last 16 bytes of the record fragment.
   - The ciphertext comprising the remaining bytes of the record fragment.
4. M serves C content to poll S at a short interval and continues to record the responses in a lookup table indexed by explicit nonce part.
5. When S repeats a nonce, M builds a polynomial derived from the relevant pair of records. In GCM under TLS, the AAD for a record includes both the record header and the sequence number. The ciphertext and tag are as described above.
6. M factors this polynomial to find a short list of candidates for the authentication key. M takes the set intersection of this list and the previous list of candidates. (In the event that this is the first collision, this list is the set of all possible keys.)
7. If more than one candidate for the authentication key remains, M returns to step 4. Otherwise, M serves content redirecting C to a static endpoint of the vulnerable application S.
8. M intercepts the response from S. Since the target is a static endpoint, it is trivial for M to inject malicious content into the response by `XOR`ing the known payload against the tail of the ciphertext and then `XOR`ing the malicious content against the same.

Additional considerations:

- If S does not repeat nonces more than once in a given session, M may attempt to hijack the connection even if there are multiple candidates for the authentication key. M can simply guess at one of the candidates and attempt to tamper with a server response before redirecting. If the guess is incorrect, C and S will simply renegotiate a new session and M can try again.

- If the application served by S does not contain any static HTML endpoints, M may choose instead to target static resources such as CSS or JavaScript. This will additionally require M to tamper with the HTTP response headers to change the content type to `text/html`.

A proof-of-concept exploit for this attack is provided on GitHub.[4]

7 Further Observations

In this section we discuss further observations made by analysis of cryptographic protocols and libraries.

7.1 Encryption Oracle with Zero-Length Inputs

The AES-GCM specification allows one to encrypt arbitrary data of a length up to $2^{39} - 256$ bits [25]. It is also possible to encrypt zero-length messages with zero-length additional authentication data. In that case, the output of the last $Gmul$ function becomes zero: $S = 0$ (see Fig. 1). This implicates that the authentication tag is equal to the encryption of the first counter block $T = Enc_k(J_0)$.

This property of AES-GCM has no direct security consequences. If a victim encrypts zero-length data with AES-GCM, the attacker learns neither the secret key k nor the hash key H. However, if the attacker can force the victim to encrypt zero-length data, he is able to obtain valuable plaintext/ciphertext block pairs. Thus, he can use the victim as an encryption oracle to encrypt random messages. If the ciphertext receiver supports different modes of operations – e.g., CBC (Cipher Block Chaining) – the attacker is then able to construct valid messages encrypted with victim's secret key k.

We stress that this is not a valid TLS scenario. First, in TLS the additional authentication data is non-zero [4]. It consists of a sequence number, message type, TLS version and message length. Second, it is not possible to use the same symmetric key for different algorithms in TLS. However, this property of AES-GCM could become exploitable in specific scenarios where the same symmetric keys can be used for different modes of operations [17]. Potential examples include XML Encryption [7], JSON Web Encryption [18], or PKCS#11 [11].

7.2 Analysis of Cryptographic Libraries

We analyzed usage of AES-GCM nonces in cryptographic libraries: Botan 1.11.28, BouncyCastle Java 1.54, MatrixSSL 3-7-2b, SunJCE 1.8, and OpenSSL 1.0.2g. None of the TLS servers provided by these libraries was vulnerable to nonce-reuse attacks. The first four libraries set the nonce value to zero and increment the nonce value with each new record. OpenSSL behaves differently and sets the first nonce value to a random 8-byte string. Further nonce values are constructed by incrementing this random string.

We furthermore investigated the usage of the counter value (cnt) in these libraries. According to the standard, if the number of blocks in one ciphertext is larger than 2^{32}, a modulo reduction should applied: $cnt = i \mod 2^{32}$ [25]. This is misleading because the standard prescribes usage of at most 2^{32} blocks, and it is known that repeating counter values is insecure in AES counter mode of operation. Interestingly, the Botan library did not

[4] https://github.com/nonce-disrespect/nonce-disrespect/

perform the modulo reduction and in a case the number of blocks in one ciphertext was larger than 2^{32}, the counter overflowed and the last byte of the nonce value was increased as well. To the best of our knowledge, this does not influence the security of Botan. Botan developers fixed this issue in version 1.11.30 [3]. We observed a similar behavior in the MatrixSSL library.

8 Conclusions and Recommendations

Our results show that there is a significant risk of GCM getting implemented in an insecure way. This risk gets elevated by the fact that the TLS specification gives developers little guidance on how to implement GCM securely. More resilient approaches are possible though and we outline two solutions briefly below.

The TLS drafts for Chacha20-Poly1305 [22] (which has been submitted to IESG for publication recently) and AES-OCB [36] both specify methods to generate nonces from record sequence numbers and shared secrets in a deterministic way. This construction prevents that implementations choose their own (potentially insecure) nonce generation methods, saves some bandwidth since an explicit transmission of the nonce is not necessary anymore, and assures that erroneous implementations are non-interoperable. In particular, this avoids the risk of developers choosing random or, even worse, constant nonces. TLS 1.3 [31] enforces a similar approach for all of its AEAD cipher suites.

The other alternative is to rely on cryptographic algorithms that inherently resist nonce-misuse, i.e. such ciphers uphold their security guarantees, even if a nonce-key pair is reused for different messages. The price for this property however is that these ciphers are inherently "offline", meaning that two passes over the data are necessary in order to perform authenticated encryption. These algorithms usually operate in a MAC-Then-Encrypt-like manner where first message and associated data are processed to produce the authentication tag and then the latter is used as the nonce for the encryption algorithm. Such nonces are also often known as *synthetic IVs* (SIV), a term first coined by Rogaway and Shrimpton [32] in 2006. Examples of nonce misuse-resistant algorithms include AES-SIV [14], AES-GCM-SIV [13], AEZ [15], HS1-SIV [20], and MRO from the MEM-AEAD [10] cipher family.

In general future protocols should rely on algorithms and constructions that reduce the risk of implementation errors as much as possible. Both of the options presented above are viable approaches to protect against nonce-misuse. In this specific case ambigious wording and human-error seemed to be the reason for a rather serious attack against TLS and in particular HTTPS. One of the authors has prepared an errata[5] to [33] which is currently being discussed within the TLS working group.[6] It is our hope that future protocol design decisions take human error, common implementation and software bugs into account and ensure that appropriate, distinct and clear discussion will be added to the security considerations section of documents published within the IETF Security Area.

[5] https://www.rfc-editor.org/errata_search.php?rfc=5288&eid=4694

[6] https://mailarchive.ietf.org/arch/search/?email_list=tls&gbt=1&index=pV7IzE5XmgUytI5OemV-vqjzPqE

9 Acknowledgements

We thank Adam Langley from Google for having the initial idea of nonce reuse issues and for helping with vendor contacts, Aaron Kaplan from the Austrian CERT.at for helping us to contact server operators and device manufacturers, and Kenny Paterson and Eric Rescorla for providing very useful, early feedback during Real World Cryptography 2016 and later on a draft version of this paper. We further thank Yoav Nir from Check Point for helping us finding a vendor of an affected load balancer, and Tibor Jager for helpful discussions on AES-GCM and cross-cipher attacks.

This research was funded by the Austrian Research Promotion Agency (FFG) through the grant P846028 (TLSiP) and the COMET K1 program, and by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1).

References

[1] Al Fardan, N. J., and Paterson, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 526–540. Author version available at http://www.isg.rhul.ac.uk/tls/TLStiming.pdf.

[2] AlFardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. On the security of RC4 in TLS. In *22nd USENIX Security Symposium* (2013), pp. 305–320. Author version available at http://www.isg.rhul.ac.uk/tls/RC4biases.pdf.

[3] Botan. Release notes, version 1.11.30, 2016. https://botan.randombit.net/news.html [Accessed 2016-07-19].

[4] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. https://tools.ietf.org/html/rfc4346.

[5] Duong, T., and Rizzo, J. Here come the XOR ninjas. *Unpublished manuscript* (2011). http://netifera.com/research/beast/beast_DRAFT_0621.pdf.

[6] Dworkin, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Nov. 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[7] Eastlake, D., Reagle, J., Hirsch, F., Roessler, T., Imamura, T., Dillaway, B., Simon, E., Yiu, K., and Nyström, M. XML Encryption Syntax and Processing 1.1. *W3C Candidate Recommendation* (2012). http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018.

[8] Ferguson, N. Authentication weaknesses in GCM. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/CWC-GCM/Ferguson2.pdf.

[9] Fluhrer, S. R., Mantin, I., and Shamir, A. Weaknesses in the key scheduling algorithm of RC4. In *SAC 2001* (Aug. 2001), S. Vaudenay and A. M. Youssef, Eds., vol. 2259 of *LNCS*, Springer, Heidelberg, pp. 1–24.

[10] Granger, R., Jovanovic, P., Mennink, B., and Neves, S. Improved Masking for Tweakable Blockciphers with Applications to Authenticated Encryption. In *Advances in Cryptology – EUROCRYPT 2016* (2016), M. Fischlin and J.-S. Coron, Eds., vol. 9665 of *LNCS*, Springer Berlin Heidelberg, pp. 263–293.

[11] Griffin, R., and Fenwick, V. Pkcs #11 cryptographic token interface, version 2.40, oasis specification.

[12] Gueron, S., and Krasnov, V. The fragility of aes-gcm authentication algorithm. In *Information Technology: New Generations (ITNG), 2014 11th International Conference on* (April 2014), pp. 333–337. Preprint available at https://eprint.iacr.org/2013/157.pdf.

[13] Gueron, S., and Lindell, Y. GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. https://eprint.iacr.org/2015/102.pdf.

[14] Harkins, D. Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). RFC 5297 (Informational), Oct. 2008. https://tools.ietf.org/html/rfc5297.

[15] Hoang, V. T., Krovetz, T., and Rogaway, P. Robust authenticated-encryption AEZ and the problem that it solves. In *EUROCRYPT 2015, Part I* (Apr. 2015), E. Oswald and M. Fischlin, Eds., vol. 9056 of *LNCS*, Springer, Heidelberg, pp. 15–44.

[16] IBM. Security Bulletin: Vulnerability in IBM Domino Web Server TLS AES GCM Nonce Generation. https://www-01.ibm.com/support/docview.wss?uid=swg21979604 [Accessed 2016-05-13].

[17] Jager, T., Paterson, K. G., and Somorovsky, J. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. NDSS 2013. https://www.nds.rub.de/research/publications/backwards-compatibility/.

[18] Jones, M., and Hildebrand, J. JSON Web Encryption (JWE) – draft-ietf-jose-json-web-encryption-06, May 2015. https://tools.ietf.org/html/rfc7516.

[19] Joux, A. Authentication failures in NIST version of GCM. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/800-38_Series-Drafts/GCM/Joux_comments.pdf.

[20] Krovetz, T. HS1-SIV v1. CAESAR Round 1 submission, 2014.

[21] Krovetz, T., and Rogaway, P. The software performance of authenticated-encryption modes. In *Fast Software Encryption — FSE 2011* (Berlin, Heidelberg, 2011), A. Joux, Ed., Springer Berlin Heidelberg, pp. 306–327.

[22] Langely, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and Josefsson, S. ChaCha20-Poly1305 cipher suites for transport layer security (TLS). Internet-Draft, Dec. 2015. https://tools.ietf.org/html/draft-ietf-tls-chacha20-poly1305-04.

[23] Langley, A. The POODLE bites again. https://www.imperialviolet.org/2014/12/08/poodleagain.html [Accessed 2016-05-13].

[24] Langley, A. TLS symmetric crypto. https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html [Accessed 2016-05-13].

[25] McGrew, D., and Viega, J. The Galois/Counter Mode of Operation (GCM), May 2005. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf.

[26] Möller, B., Duong, T., and Kotowicz, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014. https://www.openssl.org/~bodo/ssl-poodle.pdf [Accessed 2016-05-13].

[27] Nir, Y., and Langley, A. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational), May 2015. https://tools.ietf.org/html/rfc7539.

[28] Paterson, K. G. Countering cryptographic subversion, 2015. https://hyperelliptic.org/PSC/slides/paterson-PSC.pdf [Accessed 2016-05-13].

[29] Popov, A. Prohibiting RC4 Cipher Suites. RFC 7465 (Informational), Feb. 2015. https://tools.ietf.org/html/rfc7465.

[30] Radware. Sa18456: Security advisory explicit initialization vector for aes-gcm cipher. https://kb.radware.com/Questions/SecurityAdvisory/Public/Security-Advisory-Explicit-Initialization-Vector-f [Accessed 2016-05-13].

[31] 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.

[32] Rogaway, P., and Shrimpton, T. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. Cryptology ePrint Archive, Report 2006/221, 2006. http://eprint.iacr.org/2006/221.

[33] Salowey, J., Choudhury, A., and McGrew, D. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), Aug. 2008. https://tools.ietf.org/html/rfc5288.

[34] Somorovsky, J. Curious padding oracle in openssl (cve-2016-2107), 2016. http://web-in-security.blogspot.de/2016/05/curious-padding-oracle-in-openssl-cve.html.

[35] Tzvetanov, K. A10 CVE-2016-0270 GCM nonce vulnerability. https://files.a10networks.com/vadc/cve-2016-0270-gcm-nonce-vulnerability/ [Accessed 2016-07-11].

[36] Zauner, A. AES-OCB (Offset Codebook Mode) Ciphersuites for Transport Layer Security (TLS). Internet-Draft, Apr. 2016. https://tools.ietf.org/html/draft-zauner-tls-aes-ocb-04.

A Applications, Operating Systems, Compilers & Library Versions Used in our Tests

```
$ uname -a
Linux scan.sba-research.org 3.13.0-83-generic #127-Ubuntu SMP Fri Mar 11
↪ 00:25:37 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux

$ lsb_release -a
No LSB modules are available.
Distributor ID:Ubuntu
Description:Ubuntu 14.04.4 LTS
Release:14.04
Codename:trusty

$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.7) stable release version 2.19,
↪ by Roland McGrath et al.
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
```

```
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.8.2.
Compiled on a Linux 3.13.11 system on 2016-02-16.
Available extensions:
crypt Availabledd-on version 2.1 by Michael Glad and others
GNU Libidn by Simon Josefsson
Native POSIX Threads Library by Ulrich Drepper et al
BIND-8.2.3-alT5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/eglibc/+bugs>.

$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ ./masscan --version

Masscan version 1.0.3 ( https://github.com/robertdavidgraham/masscan )
Compiled on: May 20 2015 15:17:28
Compiler: gcc 4.8.2
OS: Linux
CPU: unknown (64 bits)
GIT version: 1.0.3-95-gb395f18

openssl-1.0.2e-patched$ ./apps/openssl version -a
WARNING: can't open config file: /usr/local/ssl/openssl.cnf
OpenSSL 1.0.2e 3 Dec 2015
built on: reproducible build, date unspecified
platform: linux-x86_64
options:  bn(64,64) rc4(16x,int) des(idx,cisc,16,int) idea(int)
↳ blowfish(idx)
compiler: gcc -I. -I.. -I../include  -DOPENSSL_THREADS -D_REENTRANT
↳-DDSO_DLFCN -DHAVE_DLFCN_H -Wa,--noexecstack -m64 -DL_ENDIAN -O3 -Wall
↳-DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5
↳-DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM
↳-DAES_ASM -DVPAES_ASM -DBSAES_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
↳-DECP_NISTZ256_ASM
OPENSSLDIR: "/usr/local/ssl"
```