The fragility of AES-GCM authentication algorithm

Shay Gueron[1,2], Vlad Krasnov[2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Corporation, Israel Development Center, Haifa, Israel

March 15, 2013

Abstract. A new implementation of the GHASH function has been recently committed to a Git version of OpenSSL, to speed up AES-GCM. We identified a bug in that implementation, and made sure it was quickly fixed before trickling into an official OpenSSL trunk. Here, we use this (already fixed) bug as a real example that demonstrates the fragility of AES-GCM's authentication algorithm (GHASH). One might expect that incorrect MAC tag generation would only cause legitimate message-tag pairs to fail authentication (which is already a serious problem). However, since GHASH is a "polynomial evaluation" MAC, the bug can be exploited for actual message forgery.

Keywords: AES-GCM, GHASH, polynomial evaluation MAC, message forgery, OpenSSL.

1 Introduction

AES-GCM (Galois Counter Mode; [1]) is considered to be a most efficient NIST standard Authenticated Encryption scheme. Its software implementation on modern processors is an important optimization target, and various improvements have been introduced in the last few years (e.g., [2], [4]).

Recently (February 2013 [8]), a new implementation of the function '*gcm_ghash_clmul*' was been committed (by OpenSSL Development Team member A. Polyakov) to the "Git" version of OpenSSL [7], and was awaiting interception in the next revision (*1.0.2*) of this library.

Since [8] was committed and also passed all of the built-in OpenSSL tests, it is fair to assume if it had not been intercepted, it would have appeared in the next OpenSSL version (*1.0.2*). We uncovered a bug in [8], and to stop it from appearing in an official OpenSSL version, we notified OpenSSL (March 5, 2013); the bug was fixed immediately (March 6; [9]). Therefore, the current situation allows us to analyze a real vulnerability (not just theoretical), but without pointing to a real exposure. We emphasize the following: although the message forgery vulnerability that we report here can be reproduced from the references and examples we provide, the actual security threat is *already removed* (even from the development version).

As we show here, the cryptographic construction that underlies the GHASH algorithm can extend the effect of an implementation bug (at least the discussed one)

beyond an obvious "interoperability problem", where legitimate message-tag pairs fail authentication. It can actually be exploited for message forgery.

Interestingly, the implementation in [8] passed all of the OpenSSL built-in OpenSSL test vectors (at that time; we added a new test vector to this battery [10]). Therefore, this incident shows yet another alarming example for the fact that passing a handful of test vectors is not enough to validate a cryptographic implementation.

The paper is organized as follows. Section 2 discusses some preliminaries. Section 3 describes the bug and its root cause. Section 4 demonstrates how an adversary can exploit the bug and create a message forgery. In Section 5, we analyze a few aspects of AES-GCM code optimizations, and point to a more efficient AES-GCM implementation than the one offered by the OpenSSL Git version [8] (and [9]).

2 Preliminaries

AES-GCM [1] combines AES-CTR mode for the encryption, and the GHASH algorithm for the authentication. The GHASH algorithm is a special form of the Carter-Wegman polynomial evaluation MAC. Each *16*-bytes block of the authenticated data is multiplied by a different power of the hash key (*H*), where computations occur in some specific binary finite field that we denote here by $GF\_GCM(2^{128})$ (subtleties related to this field and its operations are discussed in [4]).

The data that is authenticated by AES-GCM is the ciphertext, and optionally some additional clear-text (called Additional Authenticated Data and denoted *AAD* hereafter). In the final steps of the GHASH computations, the lengths of the ciphertext and the *AAD* are concatenated, formatted as a *16*-bytes block, and consumed into the GHASH computations (to prevent length extension attacks). Figure 1 illustrate the AES-GCM algorithm.

For efficiency, it is possible to defer the reduction modulo the $GF\_GCM(2^{128})$ polynomial (*P* hereafter), by aggregating the cumulative contribution of several blocks of the authenticated data, and reducing the result modulo *P* only then. Figure 2 illustrates this method (details can be found in [2], [3], [4]).

$AES\text{-}GCM_K(IV, P, A)$:

1. Let $H = AES_K(0^{128})$
2. Define J_0, as follows:
   If $bit\_len(IV)=96$, then $J_0=IV||0^{31}||1$
   If $bit\_len(IV)\neq 96$, then let $s=128\cdot\lceil bit\_len(IV)/128\rceil - bit\_len(IV)$,
   and let $J_0=GHASH_H(IV||0^{s+64}||[bit\_len(IV)]_{64})$
3. Let $C = GCTR_K(inc_{32}(J_0), P)$
4. Let $u = 128\cdot\lceil bit\_len(C)/128\rceil - bit\_len(C)$
   Let $v = 128\cdot\lceil bit\_len(A)/128\rceil - bit\_len(A)$
5. Define a block, S, as follows:
   $S=GHASH_H(A||0^v||C||0^u||[bit\_len(A)]_{64}||[bit\_len(C)]_{64})$
6. Let $T = GCTR_K(J_0,S)$
7. Return (C,T)

$GHASH_H(DATA)$ (the bit length of DATA is assumed to be divisible by 128)

1. Let $N = bit\_len(DATA)/128$
2. Let $DATA = M_1||M_2||\ldots||M_N$ (the M_j's are 16-bytes blocks)
3. Return $((\ldots((((M_1\times H)+M_2)\times H)+M_3)\times H\ldots)+M_N)\times H =$
   $M_1\times H^N + M_2\times H^{N-1} + \ldots + M_N\times H$

(operations (× and +) are in $GF\_GCM(2^{128})$)

Fig. 1. Top panel: AES-GCM flow. Bottom panel: GHASH calculation (for a buffer whose byte length is divisible by *16*).

The standard iterative GHASH computation

$$Y_i = [(X_i + Y_{i-1}) \bullet H] \bmod P$$

Deeper iteration to aggregate the effect of multiple blocks (here shown up to 4):

$$\begin{aligned} Y_i &= [(X_i + Y_{i-1}) \bullet H] \bmod P \\ &= [(X_i \bullet H) + (Y_{i-1} \bullet H)] \bmod P \\ &= [(X_i \bullet H) + (X_{i-1} + Y_{i-2}) \bullet H^2] \bmod P \\ &= [(X_i \bullet H) + (X_{i-1} \bullet H^2) + (X_{i-2} + Y_{i-3}) \bullet H^3] \bmod P \\ &= [(X_i \bullet H) + (X_{i-1} \bullet H^2) + (X_{i-2} \bullet H^3) + (X_{i-3} + Y_{i-4}) \bullet H^4] \bmod P \end{aligned}$$

(The symbol $\bullet$ represents carry-less multiplication)

Fig. 2. GHASH optimization: deferring the reduction modulo *P* step, by aggregating (via carry-less multiplications) the cumulative contribution of several data blocks.

3 The bug in the OpenSSL Git version

The bug in the GHASH implementation in [8] is found in the '*gcm_ghash_clmul*' function, which implements GHASH using the *PCLMULQDQ* instruction (giving performance benefits on processors that support this instruction). It manifests itself when the underlying function '*gcm_ghash_clmul'* is called with a '*len*' parameter satisfying *len > 64* and *len* mod *64 = 16* ('*len*' represents the byte length of the "GHASHED" buffer, and is assumed here to be divisible by *16*).

The code in [8] uses four block aggregation (see Figure 2), so the main loop of *gcm_ghash_clmul* accumulates the contribution of groups of four *16*-bytes blocks in each iteration. When the main loop terminates, the function needs to handle the 'tail' – i.e., the remaining *R = len/16* mod *4* blocks (*R = 1, 2* or *3*; for *R=0*, there is no tail).

The erroneous implementation mishandles the case where $R = 1$ and $len > 64$ as follows. Instead of GHASHING a single block, the code GHASHES two additional blocks, and the result is an incorrect authentication tag. In fact, the result is not even well defined, because the second (non-existent) block that is mistakenly incorporated into the tag's computations is consumed from a location beyond the boundaries of the buffer that holds the message.

In general, while generating the authentication tag of a message, the GHASH function may be called several times, to consume the message incrementally. For example, a typical usage would call GHASH once for the *AAD*, a second time for the ciphertext, and another final call for the padding block (that encodes the lengths). However, implementations may also call the GHASH more times, for a message that arrives in pieces. In the OpenSSL Git implementation [8], any single call to the GHASH function, with the *len* parameter as described above, corrupts the authentication tag for the entire message, and open the door for possible forgery.

We describe the situation as follows. Consider an "initial" authentication tag value *T*, and its incremental "update", accommodating a message (*M*). Let the message *M* consist of $N > 4$ blocks M_i, $1 \le i \le N$, with $N \bmod 4 = 1$. Suppose that *M* is placed in some memory buffer of $16N$ bytes, and the adjacent (in memory) *16*-bytes block is *J*. The (correct) updated authentication tag is

$$\overline{T} = \left((T + M_1) \cdot H^N\right) + (M_2 \cdot H^{N-1}) \ldots + (M_N \cdot H^1)$$

but instead, OpenSSL Git version code [8] computes

$$\bar{T}` = \left((T + M_1) \cdot H^{N+1}\right) + (M_2 \cdot H^N) \ldots + (M_N \cdot H^2) + (J \cdot H^1)$$

In case $J=0$, we get:

$$\bar{T}` = \left[\left((T + M_1) \cdot H^N\right) + (M_2 \cdot H^{N-1}) \ldots + (M_N \cdot H^1)\right] \cdot H^1 = \bar{T} \cdot H^1$$

In Section 4, we show how to use the above equalities for message forgery.

3.1 Test vectors

The OpenSSL toolkit has a self-testing capability, where different cipher-suites are tested against a fixed set of Known Answers Test Vectors. For AES-GCM, OpenSSL included the following set of *19* test vectors: *18* vectors which are taken from the NIST GCM spec [6] (with up to *64* bytes of Plaintext, and up to *20* bytes of *AAD*), and a 19^{th} one, testing the case where the message that consists of only *128* bytes of *AAD*. None of these tests covers the case where the bug is manifested, and this is why the bug was not detected by OpenSSL's self-testing utility.

To fix this situation, we provided, together with the bug report, a test vector traps the problem in [8], and it was added to OpenSSL's AES-GCM test vectors list [10].

This misfortunate failure shows that validating a cryptographic implementation requires more than checking a handful of Known Answers Tests.

4 Exploiting the bug for message forgery

We provide here three examples that illustrate how the bug in [8] can be exploited for message forgery. These examples show a possible "man in the middle" attack in the following scenarios: 1) a message sent with a tag that is computed by the incorrect implementation can be forged, and pass authentication by a system that has a correct implementation; 2) a message sent with a correct tag can be forged, and pass authentication by a system that uses the incorrect implementation.

The first type of attack requires that a) the adversary knows the *16*-bytes value (*J*) that is found (in memory) after buffer that holds the authenticated data (when the GHASH computations are carried out); b) first block of the authenticated data is zero. The second type of attack requires the receiver to store (in memory) the incoming buffer in such a way that it is followed by a zero block (before the call to GHASH). These situations are conceivable.

4.1 Example 1

Consider a message *M* that consists of *80* bytes of *AAD*, where the leading *16* bytes are zero. The authentication tag is computed by the erroneous implementation [8], and is validated authenticated by some correct implementation. The adversary intercepts the original message and replaces it by a new message that shifts the *AAD* to the left by *16* bytes, while injecting zeroes from the right. The adversary submits the original (erroneously computed) authentication tag. The erroneous implementation would accept the {*M*, tag} pair as valid. Figure 4 shows an example.

```
The encryption key:
        [3da6c536d6295579c0959a7043efb503]
The IV (12 bytes):
        [2b926197d34e091ef722db94]
The AAD (80 bytes):
        [00000000000000000000000000000000]
        [00010203040506070809 0a0b0c0d0e0f]
        [10111213141516171819 1a1b1c1d1e1f]
        [20212223242526272829 2a2b2c2d2e2f]
        [30313233343536373839 3a3b3c3d3e3f]
The correct authentication tag (16 bytes):
        [69dd586555ce3fcc89663801a71d957b]
The erroneous authentication tag produced by OpenSSL Git [8](16 bytes):
        [8ba1670773352621a84abd476042f928]
The forged message (80 bytes):
        [00010203040506070809 0a0b0c0d0e0f]
        [10111213141516171819 1a1b1c1d1e1f]
        [20212223242526272829 2a2b2c2d2e2f]
        [30313233343536373839 3a3b3c3d3e3f]
        [00000000000000000000000000000000]
The (correct) authentication tag of the forged AAD(16 bytes):
        [8ba1670773352621a84abd476042f928]
```

Fig. 3. Forgery Example 1.

4.2 Example 2

Consider a message *M* that consists of *32* bytes *AAD,* where the leading *16* bytes are zero, and additional *80* bytes of plaintext. The message is encrypted, and authentication tag is computed by the erroneous implementation [8]. It is decrypted and verified by a correct implementation. The adversary intercepts the original message {*AAD*, *C*, tag} and replaces it by a new message {*AAD*`,*C*`, tag}, that shifts {*AAD*, *C*} to the left by *16* bytes, while injecting zeroes from the right. The adversary uses the original authentication tag. The erroneous implementation would accept {*AAD*`, *C*`, tag} as valid. Figure 5 shows an example.

```
The encryption key:
        [843ffcf5d2b72694d19ed01d01249412]
The IV (12 bytes):
        [dbcca32ebf9b804617c3aa9e]
The AAD (32 bytes):
        [00000000000000000000000000000000]
        [101112131415161718191a1b1c1d1e1f]
The Plaintext (80 bytes):
        [000102030405060708090a0b0c0d0e0f]
        [101112131415161718191a1b1c1d1e1f]
        [202122232425262728292a2b2c2d2e2f]
        [303132333435363738393a3b3c3d3e3f]
        [404142434445464748494a4b4c4d4e4f]
The Ciphertext (80 bytes):
        [6268c6fa2a80b2d137467f092f657ac0]
        [4d89be2beaa623d61b5a868c8f03ff95]
        [d3dcee23ad2f1ab3a6c80eaf4b140eb0]
        [5de3457f0fbc111a6b43d0763aa422a3]
        [013cf1dc37fe417d1fbfc449b75d4cc5]
The correct authentication tag (16 bytes):
        [3b629ccfbc1119b7319e1dce2cd6fd6d]
The erroneous authentication tag produced by OpenSSL Git [8] (16 bytes):
        [6d0e162733eecd46fa16390d9e3e4e42]
The forged AAD (32 bytes):
        [101112131415161718191a1b1c1d1e1f]
        [6268c6fa2a80b2d137467f092f657ac0]
The forged Ciphertext (80 bytes):
        [4d89be2beaa623d61b5a868c8f03ff95]
        [d3dcee23ad2f1ab3a6c80eaf4b140eb0]
        [5de3457f0fbc111a6b43d0763aa422a3]
        [013cf1dc37fe417d1fbfc449b75d4cc5]
        [00000000000000000000000000000000]
The decrypted message (80 bytes):
        [2fe07ad2c42397002415f38eac6b8b5a]
        [8e44421b539c2f72a58b9238d80aef3a]
        [ae1e897f86b62d8ee5a2f4f25d9d023c]
        [6cee86900c7766504cc52e04b1c45059]
        [417db39f73bb073a57f68e02fb10028a]
The correct authentication tag of the forged message(16 bytes):
        [6d0e162733eecd46fa16390d9e3e4e42]
```

Fig. 4. Forgery Example 2.

4.3 Example 3

Consider a message *M* that consists of *128* bytes *AAD*, where there is a zero block, beginning with byte *96*. The authentication tag is computed by either a correct implementation or by the erroneous implementation, and is authenticated by the erroneous implementation. The adversary intercepts the original message and replaces it by a new message that: shifts the first *112* bytes of the *AAD*, to the right, by *16* bytes, while injecting zeroes from the left, and discarding spilled over data from the right. The adversary uses the original final *16* bytes of the message, and the original authentication tag. The erroneous implementation would accept the {*M*, tag} pair as valid. Figure 6 shows an example.

```
The encryption key:
        [84d5733dc8b6f9184dcb9eba2f2cb9f0]
The IV (12 bytes):
        [35d319a903b6f43adbe915a8]
The AAD (128 bytes):
        [000102030405060708090a0b0c0d0e0f]
        [101112131415161718191a1b1c1d1e1f]
        [202122232425262728292a2b2c2d2e2f]
        [303132333435363738393a3b3c3d3e3f]
        [404142434445464748494a4b4c4d4e4f]
        [505152535455565758595a5b5c5d5e5f]
        [00000000000000000000000000000000]
        [707172737475767778797a7b7c7d7e7f]
The correct authentication tag (16 bytes):
        [ed1b32c63ee51ea90320235df0b93cdc]
The correct authentication tag produced by OpenSSL, when hashed as a
whole (16 bytes):
        [ed1b32c63ee51ea90320235df0b93cdc]
The erroneous authentication tag produced by OpenSSL Gi [8], when
hashing in incremental updates of 32,80 and 16 bytes:
        [850fc587a698e3be0da023a7e9bb902f]
The forged AAD (128 bytes):
        [00000000000000000000000000000000]
        [000102030405060708090a0b0c0d0e0f]
        [101112131415161718191a1b1c1d1e1f]
        [202122232425262728292a2b2c2d2e2f]
        [303132333435363738393a3b3c3d3e3f]
        [404142434445464748494a4b4c4d4e4f]
        [505152535455565758595a5b5c5d5e5f]
        [707172737475767778797a7b7c7d7e7f]
The erroneous authentication tag produced by OpenSSL Git [8], when
hashing in incremental updates of 32,80 and 16 bytes:
        [ed1b32c63ee51ea90320235df0b93cdc]
```

Fig. 5. Forgery Example 3.

4.4 A comment on message forgery for polynomial evaluation MAC's

The above forgeries and the associated examples used the GHASH algorithm's specifics. However, the general concept is not unique to GHASH, as we explain.

Consider any MAC algorithm that operates as follows: a) using a polynomial evaluation over some finite field as an Almost Universal Hash Function; b) encrypting the polynomial value to create a MAC tag. The presence of a bug such as the one we discussed here, would lead to analogous forgeries in such MAC constructions.

5 Analysis of the software AES-GCM implementation

We are not sure what triggered the change (in February 2013) in OpenSSL's AES-GCM implementation (which, prior to that, was last modified in May 2010). We only point out that the OpenSSL patch [5] we posted just a few months before that change

(October 2012) had already indicated the option achieving better performance than the one in OpenSSL 1.0.1c (the latest at the time [5] was posted).

The patch [5] targets high performance AES-GCM for processors that have AES-NI and PCLMULQDQ instructions (e.g., 2nd and 3rd Generation Intel® Core™ processors, as well as on AMD Bulldozer). Performance is gained by interleaving the CTR mode that parallelizes encryption of *8* blocks, with GHASH that aggregates *8* blocks (before reduction). This combination avoids the latency effect of the AES-NI (*8*-cycles in the 2nd and 3rd Generation Intel Core processors), while enjoying their *1*-cycle throughput.

Regarding the aggregation level, it is unclear (to us) why for a long time (since May 2010), OpenSSL developers insisted on implementing only two blocks aggregation, and suddenly chose to increase aggregation level to four, while performance is still measured on the same processors. Perhaps (though this is not stated) they were inspired by the demonstrated method in [5] (as recommended in [2]) that have aggregation level of up to eight blocks.

In addition, [5] uses a new reduction algorithm (developer in [4]). It is especially suitable for processors that have a fast PCLMULQDQ implementation, such as the coming Haswell processor [11], while being only slightly slower than the best possible performance achievable on the 2nd and the 3rd Generation Intel® Core™ processors. In other words, this offers a well-balanced optimization for existing and future architectures.

Since GHASH is mostly used in the AES-GCM context, we argue that fusing the encryption and the authentication into a single function is natural (and has performance benefits as shown in [5]). OpenSSL's implementation [8] (and [9]) still keeps the CTR encryption, separate from the GHASH computations, although in other cases this library does interleave two algorithms that are bundled together in their typical usage (e.g., in the function '*aesni_cbc_sha1_enc*' that implements CBC-SHA1). We could not find a performance advantage in separating the encryption from the authentication (at least not for the message sizes that we examined). Certainly, no advantage is manifested when comparing the performance offered by [8] to that offered by (the previously posted) [5], on current architectures. Furthermore, as could be verified later in 2013, when the new architecture appears, the performance gap between [5] and [8] will increase.

6 Discussion

OpenSSL is a most popular open source SSL/TLS suite, used in a variety of applications. As an open source project, it allows the community to help with finding bugs and security issues, such as the one we described here. Therefore, the situation described here allows us to discuss a *real* vulnerability that can be reproduced from [8], but without causing security exposure, since it is already fixed [9], [10].

The bug that exposes the AES-GCM to message forgery is an interesting illustration of the fragility of polynomial evaluation MAC algorithms in general, and the GHASH in particular.

Finally, we note that the implementation in [5] is faster than [8] on the current processors, and the gap is going to significantly increase in the soon-to-come

processors. Since the main advantage in AES-GCM is performance, we encourage serious consideration of the performance elements demonstrated in [5], including, among other items: balancing performance over processor generations, interleaving encryption and GHASHING, deep aggregation (we suggest 8 levels), and fast reduction modulo the GCM polynomial for processors with fast PCLMULQDQ.

References

1. M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication," Federal Information Processing Standard Publication FIPS 800-38D, April 2006 http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf
2. S. Gueron, M. E. Kounavis. "Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode (Rev. 2)," Intel Software Network, 2010, http://software.intel.com/en-us/articles/carry-less-multiplication-and-its-usage-for-computing-the-gcm-mode/
3. S. Gueron, M. E. Kounavis, "Efficient Implementation of the Galois Counter Mode Using a Carry-less Multiplier and a Fast Reduction Algorithm," Information Processing Letters 110: pp. 549-553, July 2010.
4. S. Gueron, "Fast GHASH computations for speeding up AES-GCM," manuscript (*to be published*).
5. S. Gueron, V. Krasnov, "[PATCH] Efficient implementation of AES-GCM, using Intel's AES-NI, PCLMULQDQ instruction, and the Advanced Vector Extension (AVX)", October 2012, http://rt.openssl.org/Ticket/Display.html?id=2900
6. D. McGrew, J. Viega, "The Galois/Counter Mode of Operation (GCM)," Submission to NIST Modes of Operation Process, January 2004, http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
7. OpenSSL GitWeb, http://git.openssl.org/gitweb/
8. OpenSSL GitWeb, http://git.openssl.org/gitweb/?p=openssl.git;a=commit;h=273a808180e8bff35fb5113f022f8c7c966ab8d1
9. OpenSSL GitWeb, http://git.openssl.org/gitweb/?p=openssl.git;a=commit;h=28997596f2782d7b09203ac4f01e1f6bf97663cd
10. OpenSSL GitWeb, http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=ca303d333bb3ff61a946f92b2569ee98ae18c3cb
11. A. L. Shimpi, "Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel", May 2012, http://www.anandtech.com/show/6355/intels-haswell-architecture/8