Authentication weaknesses in GCM

Niels Ferguson

2005-05-20

Abstract

We show two weaknesses in the the authentication functionality of GCM when it is used with a short authentication tag. The first weakness raises the probability of a successful forgery significantly. The second weakness reveals the authentication key if the attacker manages to create successful forgeries.

1 Introduction

GCM is an authenticated-encryption block cipher mode proposed by David McGrew and John Viega [3, 4]. NIST is currently considering standardizing additional authenticated-encryption modes of operation, and GCM is one of the candidates.

GCM encrypts the message using the block cipher in counter mode, and computes the authentication tag using a polynomial over GF(2^{128}).

There are many situations in which short authentication tags are used. This is why many authenticated encryption modes, including GCM, allow a range of tag sizes. A system designer who uses a 32-bit authentication tag can reasonably assume that the probability of a successful forgery is close to 2^{-32}. As we will show, GCM does not meet this expectation.

In section 2 we give some critique on the engineering aspects of the GCM design. The rest of the paper describes the weaknesses in the authentication function of GCM when it is used with short tags.

2 General comments on GCM

Before we go into the cryptanalysis of GCM we will give some general comments on the engineering aspects of GCM. We evaluate GCM as a general-purpose cipher mode that could be used anywhere data is being encrypted and authenticated.

A general-purpose cipher mode should adapt to the widely different circumstances in which ciphers are used. From an engineering perspective, a mode that has acceptable behavior for almost all situations is preferable to a mode that is very good in some situations and very bad in others.

2.1 Message size

GCM limits the message size to $2^{36} - 64$ bytes. Though this might seem sufficient for now, it is a significant limitation. Today 64 GB is a very large message, but there are already situations in which a message of that size is reasonable. For example, it is entirely reasonable for a researcher who works with large data sets to encrypt and transfer a 100 GB file to an associate. Though there are few people today who work with large data sets like this, they will become very common as data rates increase. Cryptographic primitives are used for several decades, and should be designed with the future in mind. Encrypting a 100 GB file in 2015 might be as common as encrypting a 100 MB file today, or a 100 kB file in 1995. We feel that the message size restriction is unnecessarily burdensome for a general-purpose cipher mode.

2.2 Bit sizes

GCM allows data strings to be of arbitrary bit length. Though nice in theory, this is a disadvantage in practice. The odd bit lengths are a significant additional burden on the implementor and tester, and the odd bit length are never used by any application. Most implementors will restrict the implementation to handle only byte strings, but then the implementation cannot claim to implement full GCM. In the end, there will be a need for two sets of test vectors and two certification standards: one for byte-length implementations and one for bit-length implementations. This is all extra work for a large number of people for no practical gain. We prefer a mode that restricts itself to byte strings.

2.3 Performance and table size

As with any cryptographic function, performance is an important aspect. GCM implementations can improve the performance of the authentication function by using large pre-computed tables that contain multiples of H. As we will argue below, in the most common situations large-table implementations cannot be used, and performance comparisons should concentrate on implementations that use small tables. The GCM authors give performance numbers for various table sizes, including an implementation that uses a 256-byte table. Unfortunately, their results are for a relatively rare CPU: the Motorola G4. Brian Gladman has published GCM performance numbers for x86 platforms, but unfortunately not for an implementation with small tables [1]. A useful performance comparison with other modes would require actual performance data for small tables on the most common desktop CPUs.

Using large tables to speed up the computation adds a significant extra cost. First of all, these tables have to be pre-computed for each new key. A proper performance comparison takes the pre-computation time into account. How fast can GCM encrypt a 1 kB message with a new key? How fast can it encrypt a 100 kB message, or a 10 MB message?

For applications that use a large number of keys, the total size of all the tables becomes a problem. For example, a wireless access point could be handling 1000 keys at any one time. If it uses a 64 kB table for each key, it needs 64 MB additional memory. This is unacceptable in small embedded devices. Recomputing the table for every message or using smaller tables has a significant performance penalty. These issues need to be taken into account when doing performance comparisons with other authenticated encryption modes.

Using large tables can also be a problem in many other situations. For example: suppose GCM is implemented as part of a general library that provides a single abstract interface to a number of authenticated encryption algorithms. An application uses the abstract API and the choice of algorithm is made by a policy setting that can be changed. The application does not care, or even know, whether it is using AES-GCM, AES-CCM, 3DES-CBC-HMAC-MD5, or a new algorithm not yet invented when the application was written. There are many engineering and economic advantages to such an implementation, and they are becoming more common. Designs like this are being driven by the different national requirements for cryptographic implementations, as well as the desire to allow third-party hardware acceleration of the cryptographic computations without modifying existing applications.

A GCM implementation in such a library is given a new key. How large should it make the pre-computed table? The application is not going to provide any information about the intended use of the key, as that is not part of the abstract API. If the library uses a large table, it will consume unacceptably large amounts of memory when an application uses a large number of keys. The library almost certainly will have to restrict itself to using a small table.

The library could adaptively change the size of the table, creating a small table initially and then a larger table when the key is used a lot. When a table has not been used for a while the large table can be deleted to free up memory for other tables. Such constructions are possible, but add significant complexity (and therefore development and testing costs) to the implementation. Even worse, it produces a system with very unpredictable performance. When the load on the system increases the throughput can suddenly drop when the library can no longer maintain enough large tables in memory. The slowdown increases the number of keys that are actively in use, which makes the problem worse. Even when the load drops back to the previous level, the throughput might not recover because the loss of throughput has increased the number of keys in use.

In our experience, such adaptive implementation techniques do not work well. There are always situations in which the adaptive technique fails to implement a reasonable solution, which results in an unexpected performance drop that is hard to explain to a customer.

Another disadvantage is the sheer amount of confidential data these tables generate. The tables contain key material, and have to be protected just like other key material. In some situations this means the tables have to be kept in main memory and cannot be swapped to a swap file. (Swap file security can be complicated, and using a swap file for keying material is forbidden by certain certification standards.) Using non-swappable memory exacerbates the memory problems introduced by using large pre-computed tables.

For these reasons we believe that the most widely used software implementations (which are inevitably in the form of a generic library) will use a small table of 256 bytes. Therefore, performance comparisons of GCM to other modes should concentrate on GCM implementations that use a 256 byte table. Performance numbers for implementations with larger tables are less relevant as those large tables are only useable in certain specific settings.

2.4 CTR mode collisions

GCM can use any size IV, but it prefers a 96-bit IV. As long as the IV is 96 bits there is never a collision in the CTR mode encryption.

For any other size of IV GCM uses the GHASH function to generate a pseudo-random 128-bit starting value Y_0 for the CTR mode encryption. This introduces the risk of collisions. After about 2^{64} blocks of data have been processed we can expect a collision on the counter values. Unfortunately, GCM is rather fragile, as a collision between two messages tends to reveal the authentication key H.

If we assume that all plaintext is known, then each GCM-encrypted message reveals a sequence of key stream blocks of the form $E(K, i), E(K, i + 1), \ldots$. Suppose one of these blocks is ever used to encrypt the authentication tag of another message. The attacker can now decrypt the GHASH intermediate result in the authentication computation. This leads to a simple polynomial equation for the authentication key H which the attacker can solve. Thus, a single collision leads to a loss of the authentication key, and a loss of all authentication security.

This is an example of a general phenomenon: block cipher modes lose some security once the amount of data processed gets close to the birthday bound. However, the severity of the security loss is different for different modes. Fragility, like the complete loss of authentication security, is not a desirable property.

2.5 Length of the authentication tag

The GCM specifications [4] state that the length of the authentication tag T must be fixed for a single key, but that is not repeated in the NIST submission [3].

Applications must be very careful not to allow different tag length with the same key as this leads to a loss of security. This restriction should be made very clear to users of GCM.

3 Linear behavior of the GCM authentication function

The GCM authentication function has three inputs: the authentication key H, the additional authentication data, and the ciphertext. Our attack does not use the additional authentication data, so we ignore it for the rest of the paper.

The first part of the authentication function consists of converting the ciphertext to a sequence of blocks which we will call $C_1, \ldots, C_n$ where n is the total number of blocks used. Blocks $C_2, \ldots, C_n$ are the ciphertext, and C_1 encodes the length of the ciphertext.[1]

The authentication function can now be written as

$$T := K_0 + \sum_{i=1}^{n} C_i H^i$$

in GF(2^{128}), where K_0 is a block of key stream generated by the block cipher. The authentication tag consists of the leading bits of T.

We are going to analyze the authentication function in terms of linear operations on bits.

[1] Note that we number the C_i blocks in reverse order from the numbering used in the GCM specifications.

3.1 Representations of GF(2^{128})

There are many ways to represent the field GF(2^{128}). We use several of them.

- The abstract element of GF(2^{128}) is written as a single capital letter: e.g. X.
- The 127-degree polynomial over GF(2) that corresponds to X is denoted as Poly(X).
- The 128 coefficients of the polynomial Poly(X) are individual bits which we write as $X_0, X_1, \ldots, X_{127}$.
- The vector of length 128 over GF(2) which is made up from the polynomial coefficients is written as $\overline{X}$. We have $\overline{X} = (X_0 \cdots X_{127})^T$.

Readers who do not understand the relations between these notations can find the details in any textbook on discrete mathematics.

3.2 Multiplication by a constant is linear

The function

$$X \mapsto C \cdot X$$

in GF(2^{128}) for some constant field element C is a linear function when viewed in terms of bit vectors. In other words, for each C there is a 128×128 matrix $\boldsymbol{M_C}$ over GF(2) such that

$$\overline{C \cdot X} = \boldsymbol{M_C}\overline{X}$$

for all X. It is easy to see that the coefficients of $\boldsymbol{M_C}$ are linear functions of the bits of $\overline{C}$.

3.3 Squaring is linear

The function

$$X \mapsto X^2$$

in GF(2^{128}) is linear. This is due to the fact that GF(2^{128}) is a field of characteristic 2, which implies that

$$\forall_{A,B} : (A + B)^2 = A^2 + B^2$$

(There is an extra term $2AB$ but $2AB = AB \oplus AB = 0$ so this term disappears.)

Thus, there is a fixed matrix $\boldsymbol{M_S}$ such that

$$\overline{X^2} = \boldsymbol{M_S}\overline{X}$$

for all X. Note that $\boldsymbol{M_S}$ does not depend on anything except the chosen field representation, which is public.

3.4 Error polynomials

We want to change the ciphertext of a GCM message without the change being detected by the recipient. If the original ciphertext consists of the blocks C_i and the modified ciphertext of the blocks C'_i, then we want

$$\sum_{i=1}^{n} C_i H^i = \sum_{i=1}^{n} C'_i H^i$$

or at least we need agreement in the leading bits that form the actual authentication tag. This can be restated by saying that we need to have the leading bits of $\sum_i (C_i - C'_i)H^i$ to be zero. We call this difference polynomial the error polynomial, and for the rest of the paper we are looking for coefficients E_i such that the leading bits of

$$\sum_i E_i H^i$$

are zero. We restrict ourselves to only have nonzero coefficients when i is a power of two. In other words, we look for coefficients D_i such that the leading bits of

$$E := \sum_i D_i H^{2^i}$$

are zero, where $D_i := E_{2^i}$. We should point out that the structure of GCM imposes significant restrictions in choosing D_0 as the D_0 block is used to encode the length of the message and additional authentication data. There are, however, no restrictions on any of the other D_i values, although the message length we need grows exponentially with the number of D_i coefficients that we use.

Let's look at the error polynomial in terms of bit vectors. We get

$$\overline{E} := \sum_i \boldsymbol{M_{D_i}} (\boldsymbol{M_S})^i \overline{H}$$

where $\overline{E}$ is the result of the error polynomial. Observe that $(\boldsymbol{M_S})^i$ is a fixed known matrix for each value of i. If we define

$$\boldsymbol{A_D} := \sum_i \boldsymbol{M_{D_i}} (\boldsymbol{M_S})^i$$

then we have $\overline{E} = \boldsymbol{A_D} \cdot \overline{H}$. The matrix $\boldsymbol{A_D}$ is a 128×128 matrix over GF(2).

Matrix $\boldsymbol{M_S}$ is fixed, and the coefficients of each $\boldsymbol{M_{D_i}}$ are linear combinations of the bits of the corresponding D_i. Thus, the coefficients of each $\boldsymbol{M_{D_i}}(\boldsymbol{M_S})^i$ are a linear combination of the bits of the corresponding D_i, and the coefficients of $\boldsymbol{A_D}$ are linear combinations of the bits of the D_i's.

4 Increasing the forgery probability

It is now easy to force bits of the error polynomial to zero. Write equations setting each of the bits in a single row of $\boldsymbol{A_D}$ to zero. Each equation imposes a single linear constraint on the choice we have for the bits of the D_i values. To force a single result bit to zero we have to create 128 linear

constraints. If we have n different D_i coefficients to choose, we have $128 \cdot n$ free variables and we can force $n - 1$ bits of the result to zero.[2]

Let's assume that GCM is used with a 32-bit authentication tag. We furthermore assume that there is a known (properly authenticated) message of 2^{17} blocks (about 2 MB).

Due to the details of GCM, the block corresponding to D_0 of our error polynomial encodes the length of the message. In our first attack we won't change the length of the message, so we have $D_0 = 0$.

The blocks $D_1, \ldots, D_{17}$ can be freely chosen in any message forgery attempt. With 17 times 128 free variables we can find nonzero solutions that zero out 16 rows of the A_D matrix as described above. This, in turn, ensures us that the first 16 bits of the authentication tag will not change if we apply the differences D_i to the ciphertext. With only 16 effective authentication bits left, we have a 2^{-16} chance of a successful forgery. This is a much higher chance than one would reasonably expect from a 32-bit authentication code.

5 Recovering the authentication key

After 2^{16} forgery attempts we can expect a successful forgery. A successful forgery reveals the fact that the authentication tag wasn't changed, which is the same as saying that the first 32 bits of $\overline{E}$ were unchanged. We already knew the first 16 bits were zero, but the fact that the other 16 bits are zero gives us 16 linear equations on the bits of H.

With these equations we can write $\overline{H}$ as follows:

$$\overline{H} = \boldsymbol{X}\overline{H'}$$

where $\overline{H'}$ is a 112-bit column vector representing the unknown bits of H, and $\boldsymbol{X}$ is a known 128×112 matrix derived from the linear equations we have for the bits of H. We now have

$$\overline{E} = \boldsymbol{A_D} \cdot \boldsymbol{X} \cdot \overline{H'}$$

and the matrix $\boldsymbol{A_D} \cdot \boldsymbol{X}$ is a 128 by 112 bit matrix. Forcing a complete row to zero only requires 112 free variables, so for the next forgery attempt we can use our 17×128 free variables in the D_i values to zero 19 rows of the matrix $\boldsymbol{A_D} \cdot \boldsymbol{X}$. This improves the forgery probability to 2^{-13}.

This process can obviously be repeated. Every time a forgery is successful it reveals more information about H, which makes the next forgery even easier. Towards the end we can create forgeries where we have forced 31 of the error polynomial bits to zero, at which point the receiver is revealing one bit of information about H with every forgery attempt.

This quickly leads to a situation in which the attacker knows all of H, after which arbitrary forgeries can be created without any chance of detection by the receiver.

[2] If we were to try to force n bits to zero we would have as many variables as equations, and the obvious solution of setting all $D_i = 0$ would be the only one. The all-zero solution is of course not interesting.

5.1 A disastrous usage scenario

There are some situations in which even shorter authentication tags are entirely reasonable. For example, suppose a secure telephone system uses short authenticated and encrypted packets with 48 bytes of payload to transfer the voice data to the other side with low latency. Rather than have a large authentication tag for each packet it is reasonable to use an 8-bit authentication tag, with an expected forgery rate of 2^{-8}. This is acceptable for voice systems, as an attacker that can forge less than 0.5% of the packets cannot effectively forge the voice channel because the result would be mostly silence due to bad packets. Let's also assume that the communication protocol returns an indication of packet failures (e.g. for quality control, error logging, vocoder switching, or any other reason).

If the designer chooses GCM things fall apart very quickly. The attacker has two D_i blocks to play with, and can force one bit of the error polynomial result to zero. After 128 forgery attempts he succeeds once and learns 7 bits of H. He can now force 2 bits of to zero, and learns the next 6 bits after 64 attempts. All in all it takes, on average, less than 1000 forgery attempts to recover H, after which the attacker can forge all the packets he wants. If each packet contains 20 ms of sound (a reasonable value) the attacker could try 5 forgeries per second. This would introduce a 10% packet loss rate, which results in a voice channel that is noticeably flawed but still quite useable. After about three minutes the attacker knows H and can forge all subsequent packets.

6 Further improvements and future work

There are small improvements that can be made to this attack. The block that corresponds to D_0 of the error polynomial encodes the message length. If the length of the message is not a multiple of 16, then the attacker can extend the message length by appending zero bytes to the ciphertext. This changes only the length encoding in D_0. By introducing a nonzero D_0, the all-zero solution is no longer possible when we solve for suitable D_i values. This means that the attacker can zero out k bits of the tag using only k D_i's, rather than the $k + 1$ we had before. As the efficiency of the attack is dominated by finding the first successful forgery, this doubles the efficiency of the attack.

We have only given a very simple analysis of how to proceed with the attack when linear equations on the bits of H are known. There might be better ways of using this knowledge.

Every failure in a forgery attack provides partial knowledge about H. The attacker knows that at least one of remaining bits in E was nonzero. Using this information is much harder, but it could lead to further improvements.

Our attack uses only linear analysis. It might be fruitful to analyze GCM in terms of non-linear functions in the bits of H. The coefficients of H^i are in general d'th degree functions of the bits of H, where d is the Hamming weight of i. For example, allowing quadratic functions on the bits of H in our attack would increase the number of E_i coefficients that can be used in the attack for a given message length. The whole analysis then becomes far more complex, and it remains to be seen whether this leads to an improved attack.

7 Fixing GCM

The weakness of the authentication function of GCM is not fundamental to GHASH. It is well known that an n'th degree polynomial can have n roots, so the collision probability for GHASH when applied to a 2^k block message is around $2^{-(128-k)}$, which will be acceptable in many situations.

The problem is in the way the GCM truncates the GHASH result. The linear structure of GHASH allows us to force individual bits of the truncated result to zero. GCM encrypts the GHASH result by xorring it with a block of key stream, which does not prevent manipulation of the individual output bits.

CWC (a competing mode) solves this very same problem by encrypting the hash result with the block cipher before xorring it with a block of the key stream [2]. As far as we can see this construction would resolve the GCM short-tag authentication weakness.

We must point out that we have not done any cryptanalysis or security proofs for this modified version of GCM. We strongly urge people not to use this version until cryptanalytical review and security proofs have been completed.

8 GCM's proof of security

Our attack does not provide a counterexample to GCM's proof of security [4]. Rather, our attack is exactly on the boundary of what the proof claims the security of GCM is.

From the reactions we have gotten to our results it seems that most people in the field had not realized that this type of attack was allowed by the security proof. This suggests that the cryptographic community could do a better job documenting and explaining exactly what each security proof promises. (This should not be taken as a criticism of the GCM authors. Their claims are in fact easier to read than most. But there is still ample room for improvement.)

9 Recommendations

Based on these weaknesses, our recommendations are:

- Do not use GCM. Consider using one of the other authenticated encryption modes, such as CWC, OCB, or CCM.
- If other considerations dictate the use of GCM, use it only with a 128-bit tag.

10 Conclusion

GCM has two authentication weaknesses.

The first weakness is that an n-bit tag provides only $n - k$ bits of authentication security when messages are 2^k blocks long. Competing modes do not have this problem, or have it only when $n = 128$, in which case the practical effect is minimal.

The second weakness is that a successful forgery immediately reveals information about the authentication key. This weakness exacerbates the consequences of the first one, and leads to a complete loss of authentication security.

11 Acknowledgements

I would like to thank David Wagner, Doug Whiting, Yoshi Kohno, Josh Benaloh, David McGrew, John Viega, Dan Shumow, and Denise Ferguson for their time, insightful comments, and support.

References

[1] Brian Gladman. AES and combined encryption/authentication modes. `http://fp.gladman.plus.com/AES/index.htm`.

[2] Tadayoshi Kohno, John Viega, and Doug Whiting. CWC: A high-performance conventional authenticated encryption mode. `http://eprint.iacr.org/2003/106/`.

[3] David A. McGrew and John Viega. The Galois/Counter Mode of operation (GCM). `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf`.

[4] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation (full version). `http://eprint.iacr.org/2004/193`.