pwnaccelerator BLOG

security and maybe more

Hunting For Vulnerabilities in Signal - Part 2

Sep 19, 2016

We (@marver and @veorq) released information about two vulnerabilities that we discovered in Signal in part 1 of this series of posts about what we found during an informal audit of the Signal source code.

Naturally there were some discussions about the impact and practical implications. We still don't consider any of the vulnerabilities as critical or exceptionally bad. In particular, the MAC bypass seems difficult to leverage in order to get a padding oracle, due to a missing back channel.

However, we would like to show some things that can actually be achieved when exploiting the MAC validation bypass via the AES CBC cipher mode that is used for Signal attachments. Many thanks also to Hanno Böck (@hanno) who gave valuable input about the practical AES-CBC attack described below.

Impact of the MAC Bypass

As explained in the previous post, the bug we found allows you to append exactly 4GB of data to the original attachment (but remember: only about 4MB have to be actually transferred over the network when using gzip compression). This modified attachment has now the size 4GB+X if the original encrypted file has the size X.

So what can you do with this? Many people would probably say "well not so much". This is not really correct, as we will show here, thanks to the use of the malleability provided by the CBC block cipher encryption mode.

This actually gives us control over part of the decrypted data. Technical details are given down below in the last section of this post, but the implications are the following:

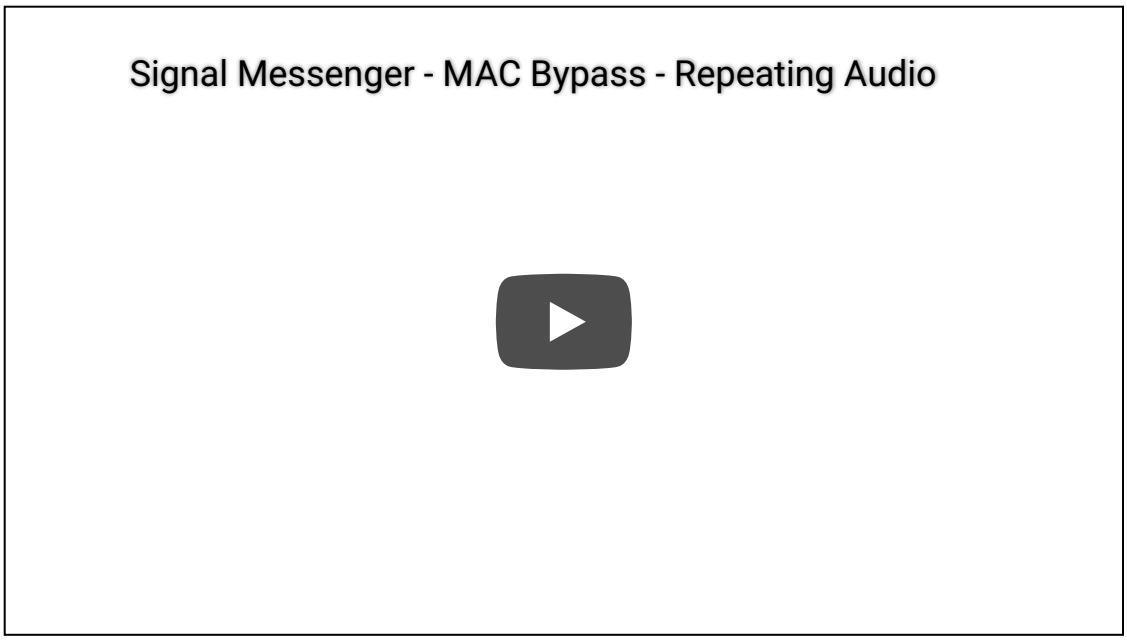
When no plaintext is known

We can do the following:

- It is possible to append the encrypted file to itself: Resulting in the original file being decrypted, then 48 bytes of garbage (MAC and a bad block) and then the rest of the original file again.
- It is possible to append part of the encrypted file to itself: Resulting in the original file being decrypted, then 48 bytes of garbage (MAC and bad block), and then parts of the encrypted file selected by a MITM and in an order selected by a MITM.

So the MITM can actually forge a new file from the parts of the original one. Some parts of this file will be random garbage, but not all. The garbage blocks result from the fact that whenever appending two ciphertext regions together, there will be a 16 byte block of data that decrypts incorrectly. Please see below for a detailed description of why this is.

In case of a robust media codec such as MP3 we can create a little proof of concept of repeating the decrypted files contents resulting in a file that is still playable:

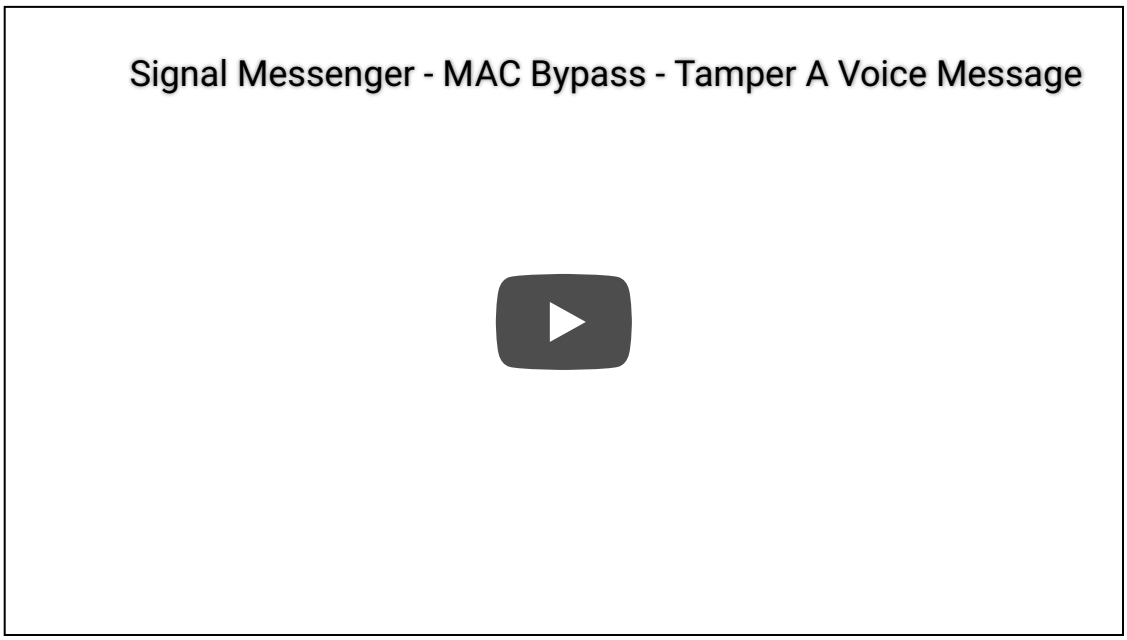


Our tests showed that opus, ogg, and other containers may also be concatenated and are still playable.

When we know more about what was originally sent

If we know something more about the file encrypted, e.g. that it is a text being spoken and when it is spoken, we could even create a new more convincing and playable audio file.

The result of this can be seen in the following video:



If the file being sent is well known by the attacker, he might learn this by correlating file sizes (which will be nearly the same size as the encrypted size) in case of common files.

In a more advanced attack, if an attacker knows the actual plaintext (the data that our encrypted attachment is finally decrypted to), he might even do worse things such as controlling up to every second block of the decrypted data, while the other ones will decrypt to random data. For example one could imagine that under special circumstances, injecting control data into media streams could trigger a bug in the code parsing this stream (stagefright anyone?).

This attack exploits a known property of the CBC mode (Bellovin on IPsec in 1996, Bodo Moller on SSL, Vaudenay), as noted by a comment on the Ars Technica article about our initial publication.

The script used to attack the attachment server can be downloaded here.

Something keep in mind: If the attacker knows just a single block of plaintext in the encrypted attachment being sent, he can start the attack described above by just using this single block. How much would you bet against creative individuals being able to exploit this in some way or the other?

Following now in the following section is an exact description about how to attack AES-CBC.

Exploiting CBC Malleability

In cryptanalysis, *malleability* is the ability to turn some ciphertext into another ciphertext whose matching plaintext is meaningfully related to the original plaintext. The CTR mode, for example, allows trivial malleability: if `C` is the ciphertext of some plaintext `P`, then `C ⊕ M` is a valid ciphertext of `P ⊕ M`, using a same nonce.

With CBC, as used to encrypt Signal attachments, malleability works like this:

We've got a ciphertext `C = C[1] || C[2] || ... || C[n]` where

```
C[1] = E(K, IV ⊕ P[1])
```

given the IV transmitted along with the ciphertext, and

```
C[i] = E(K, C[i—1] ⊕ P[i])
```

for `i > 1`, where `P[i]`s are plaintext blocks and `E` is the encryption function. Graphically, CBC looks like this:

<img src="/images/posts/2016-09-19/cbc.png" width=600>

The ciphertext `C` obtained is followed by a 32-byte MAC tag `T`. We want to extend `C || T` with additional ciphertext data. We don't know the key `K` and we've got no encryption oracle therefore we can't forge a whole new ciphertext for the plaintext of our choice. But we can do something close to it.

The simplest we can do is append `C` and build `C || T || C`. The second `C` won 't decrypt to `P`, however: when decrypting the second occurence of `C[0]`, decryption will produce

```
D(K, C[0]) ⊕ T[1]
```

instead of `D(K, C[0]) ⊕ IV`. Since we can't modify `T` (to ensure MAC validation), we can't control the plaintext block obtained. However, the second occurences of `C[2]`, `C[3]`, …, `C[n]` will be correctly decrypted to their original plaintext blocks.

Likewise, if you create the ciphertext `C || T || C || ... || C`, consisting of the original ciphertext and MAC followed by the ciphertext repeated multiple times, then the `C`s appended will decrypt to `P'` where `P'` is the original plaintext but with an incorrect first block.

We can do better than this. We can add ciphertext material for which we totally control what every other block will decrypt to: create a ciphertext `C'`, where `C'[2] = C[2]`, such that `C'[2]` will decrypt to: `P'[2] = D(K, C'[2]) ⊕ C'[1] = P[2] ⊕ C[1] ⊕ C'[1]` Here I just replaced `D(K, C'[2]) = D(K, C[2])` with `P[2] ⊕ C[1]`, as per the CBC encryption of the original plaintext.

Now note that we've got to choose `C'[1]`, hence we can control the difference between the original plaintext block `P[2]` and the block we want `C'[2]` to decrypt to. That is, if we know the original plaintext block `P[2]`, then we can choose what the new ciphertext data will decrypt to.

Repeating the same trick for every two blocks, you can control one every two blocks of the ciphertext data added, by exploiting degrees of freedom in the preceding block. It suffices to know one plaintext block `P[i]`

from the original message in order to forge a ciphertext where every other block decrypts to the plaintext block we want.

pwnaccelerator BLOG

- pwnaccelerator BLOG

security and maybe more