# pwnaccelerator BLOG

**security and maybe more**

# Hunting For Vulnerabilities in Signal - Part 1

Sep 15, 2016

At Blackhat 2016 Jean-Philippe Aumasson and Markus Vervier were a bit bored and decided to take a peek at the Signal source code. This actually evolved into a longer hunt for bugs in the high profile messenger recommended by Snowden. Since two of the bugs for the Java reference implementation of Signal have been publicly fixed after our disclosure, we think we should give a little description about what we found.

We checked common pitfalls of Java, Objective-C and C/C++ code and common attack vectors, by reviewing the Signal code base (Signal protocol libraries, mobile clients, service). We also reviewed the general architecture of Signal and its attack surface. We found several issues of varying impact, about which we will blog in the coming months, starting with this post.

## MAC Validation Bypass For Attachments

Signal attachments are encrypted and authenticated in order to prevent a third party from reading or changing them. This also includes the Signal maintainers. The encrypt-then-MAC approach used by Signal is the most secure way to do this, compared to MAC-then-encrypt (as found in TLS) or encrypt-and-MAC (as found in SSH).

If a message is sent with an attachment, this attachment is downloaded separately from a server located on AWS, such as https://whispersystems-textsecure-attachments.s3.amazonaws.com/. Attachments are encrypted (by the sender) using AES-128-CBC with PKCS7 padding, and authenticated with HMAC-SHA-256, also using a 128-bit key.

After downloading the attachment via HTTPS and storing it on the Android storage, the file’s MAC is checked by the following code from the Signal Service, in `libsignal-service-java/java/src/main/java/org/whispersystems/signalservice/api/crypto/AttachmentCipherInputStream.java`:

```
private void verifyMac(File file, Mac mac) throws FileNotFoundException, InvalidMacException {
  try {
    FileInputStream fin           = new FileInputStream(file);
    int             remainingData = (int) file.length() - mac.getMacLength();
    byte[]          buffer        = new byte[4096];

    while (remainingData > 0) {
      int read = fin.read(buffer, 0, Math.min(buffer.length, remainingData));
      mac.update(buffer, 0, read);
      remainingData -= read;
    }

    byte[] ourMac   = mac.doFinal();
    byte[] theirMac = new byte[mac.getMacLength()];
    Util.readFully(fin, theirMac);

    if (!Arrays.equals(ourMac, theirMac)) {
      throw new InvalidMacException("MAC doesn't match!");
    }
  } catch (IOException e1) {
    throw new InvalidMacException(e1);
  }
}
```

As seen above `remainingData` is of type `int` and calculated from the length of the file subtracted by the MAC length. Since `file.length()` will return a value of type `long` and files may be larger than `Integer.MAX_VALUE`, `remainingData` will wrap around.

Unlike C(++), Java is “memory safe” and therefore this will not lead to any classical memory corruption condition. Instead we can use this overflow to subvert the program logic. Now if the file is of size *4GB + 1 byte + X*, the value will wrap around and `remainingData` will be set to *X*.

Here is the trick: Signal stores all attachments on AWS S3, fetches them over HTTPS and uses the system certificate store to check the server’s cert (note that the Signal server on S3 serves a wildcard certificate for `*.s3.amazonaws.com`). An entity with access to Amazon S3 or having access to any of the CA certificates commonly found in trust stores on Android or other systems, could modify attachments in the following way:

1. Watch for a request to fetch an attachment.
2. Fetch the original attachment of size *X*.
3. Pad the attachment with data of size *4GB + 1byte*, resulting in a total size of *X + 4GB + 1*.

As described above this will result in X bytes being checked with `verifyMAC()`, for which the original MAC is valid. Therefore we can add arbitrary data to the file without the MAC check failing!

Note that a MITM attacker does not really need to send more than 4GB of data over any network connection: If we use HTTP stream compression with gzip we can create 4GB files which compress down to 4.5MB.

A PoC was created that works as a proxy for the Signal attachment storage server, padding the content as described above:

```
[s@polo tools-markus]$ python2 sap.py --encoding gzip
Serving HTTP on 0.0.0.0 port 8000 ...
opening: https://whispersystems-textsecure-attachments.s3.amazonaws.com/attachments/id1/id2...
* Compressing content...
** Padding content...
** Finished
Compressed Content Length (Raw 49284): 4458483
* Set Content-Length to: 4458483
* Sent header, writing content
* Request finished
```

Looking into Android logcat we now see the following Exception:

```
W/AttachmentDownloadJob(10484): ws.com.google.android.mms.MmsException: java.io.IOException: javax.crypto.BadPaddingException: EVP_C
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.database.AttachmentDatabase.setAttachmentData(AttachmentDataba
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.database.AttachmentDatabase.setAttachmentData(AttachmentDataba
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.database.AttachmentDatabase.insertAttachmentsForPlaceholder(At
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.jobs.AttachmentDownloadJob.retrieveAttachment(AttachmentDownlo
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.jobs.AttachmentDownloadJob.onRun(AttachmentDownloadJob.java:84
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.jobs.MasterSecretJob.onRun(MasterSecretJob.java:18)
W/AttachmentDownloadJob(10484):         at org.whispersystems.jobqueue.JobConsumer.runJob(JobConsumer.java:76)
W/AttachmentDownloadJob(10484):         at org.whispersystems.jobqueue.JobConsumer.run(JobConsumer.java:46)
W/AttachmentDownloadJob(10484): Caused by: java.io.IOException: javax.crypto.BadPaddingException: EVP_CipherFinal_ex
W/AttachmentDownloadJob(10484):         at org.whispersystems.signalservice.api.crypto.AttachmentCipherInputStream.readFinal(Attachm
W/AttachmentDownloadJob(10484):         at org.whispersystems.signalservice.api.crypto.AttachmentCipherInputStream.read(AttachmentCi
W/AttachmentDownloadJob(10484):         at org.whispersystems.signalservice.api.crypto.AttachmentCipherInputStream.read(AttachmentCi
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.util.Util.copy(Util.java:220)
W/AttachmentDownloadJob(10484):         at org.thoughtcrime.securesms.database.AttachmentDatabase.setAttachmentData(AttachmentDataba
W/AttachmentDownloadJob(10484):         ... 7 more
W/AttachmentDownloadJob(10484): Caused by: javax.crypto.BadPaddingException: EVP_CipherFinal_ex
W/AttachmentDownloadJob(10484):         at com.android.org.conscrypt.NativeCrypto.EVP_CipherFinal_ex(Native Method)
W/AttachmentDownloadJob(10484):         at com.android.org.conscrypt.OpenSSLCipher.doFinalInternal(OpenSSLCipher.java:430)
W/AttachmentDownloadJob(10484):         at com.android.org.conscrypt.OpenSSLCipher.engineDoFinal(OpenSSLCipher.java:490)
W/AttachmentDownloadJob(10484):         at javax.crypto.Cipher.doFinal(Cipher.java:1314)
W/AttachmentDownloadJob(10484):         at org.whispersystems.signalservice.api.crypto.AttachmentCipherInputStream.readFinal(Attachm
W/AttachmentDownloadJob(10484):         ... 11 more
```

We just successfully bypassed the MAC check :-)

After checking the MAC, the constructor of the class `AttachmentCipherInputStream` will create an instance of class `javax.crypto.Cipher`:

```
    public AttachmentCipherInputStream(File file, byte[] combinedKeyMaterial)
        throws IOException, InvalidMessageException
    {
  ...
        verifyMac(file, mac);

        byte[] iv = new byte[BLOCK_SIZE];
        readFully(iv);

        this.cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        this.cipher.init(Cipher.DECRYPT_MODE, new SecretKeySpec(parts[0], "AES"), new IvParameterSpec(iv));

        this.done          = false;
        this.totalRead     = 0;
        this.totalDataSize = file.length() - cipher.getBlockSize() - mac.getMacLength();
      } catch (NoSuchAlgorithmException | InvalidKeyException | NoSuchPaddingException | InvalidAlgorithmParameterException e) {
        throw new AssertionError(e);
      } catch (InvalidMacException e) {
        throw new InvalidMessageException(e);
      }
```

Reaching this state we can now make the cipher decrypt the ciphertext of our choice. Besides the obvious problem that such large attachments are decompressed and stored on the Android storage, quickly using up all space, being able to supply arbitrary ciphertext as a suffix may allow padding oracle attacks against AES-CBC with PKCS7 (a.k.a. PKCS5) padding. This would require a communication channel back to the attacker that indicates failed padding or successful decryption. We will discuss if this is really possible and feasible in upcoming writeups.

Open Whisper Systems rapidly fixed this issue after our disclosure, with to this commit.

## Arithmetic Underflow Via Malformed RTP Packets

The `CallAudioManager` class defined in the Signal Android client at `src/org/thoughtcrime/redphone/audio/CallAudioManager.java` is using native code whose source is in `jni/redphone/CallAudioManager.cpp`.

In line 129 of `callAudioManager.cpp`, the following code serves to receive audio data into a buffer of 4096 bytes:

```
   char buffer[4096];

    while(running) {
      RtpPacket *packet = audioReceiver.receive(buffer, sizeof(buffer));
```

The function `receive` receives data into the buffer and will create a new instance of class `RtpPacket` defined in `jni/redphone/RtpPacket.cpp`:

```
RtpPacket* RtpAudioReceiver::receive(char* encodedData, int encodedDataLen) {
  int received = recv(socketFd, encodedData, encodedDataLen, 0);

  if (received == -1) {
    __android_log_print(ANDROID_LOG_WARN, TAG, "recv() failed!");
    return NULL;
  }

  RtpPacket *packet = new RtpPacket(encodedData, received);
```

When `packetLen` is smaller than `sizeof(RtpHeader)` the value of `payloadLen` set in the class constructor will become negative:

```
RtpPacket::RtpPacket(char* packetBuf, int packetLen) {
  packet     = (char*)malloc(packetLen);
  // 1. INTEGER UNDERFLOW
  payloadLen = packetLen - sizeof(RtpHeader);
  memcpy(packet, packetBuf, packetLen);
}
```

This results in `payloadLen` being incorrectly stored as negative value. Offending values to for packetLen are in the range of [0, sizeof(RtpHeader] which is between 0 and 12.

So what does this actually mean? After constructing such an RtpPacket with a short size, the `srtpStream.decrypt` method is called on this packet:

```
    RtpPacket *packet = new RtpPacket(encodedData, received);

    if (srtpStream.decrypt(*packet, sequenceCounter.convertNext(packet->getSequenceNumber())) != 0) {
      __android_log_print(ANDROID_LOG_WARN, TAG, "SRTP decrypt failed!");
      delete packet;
      return NULL;
    }

    return packet;
```

As seen above, `packet->getSequenceNumber()` is called and reads data out of bounds which is passed to `sequenceCounter.convertNext()`. Then `decrypt` is called:

```
int SrtpStream::decrypt(RtpPacket &packet, int64_t logicalSequence) {
  uint8_t iv[AES_BLOCK_SIZE];
  uint8_t ecount[AES_BLOCK_SIZE];
  uint8_t ourMac[SRTP_MAC_SIZE];

  uint32_t num    = 0;
  uint32_t digest = 0;

// 1. OOB READ
  setIv(logicalSequence, packet.getSsrc(), parameters->salt, iv);
  memset(ecount, 0, sizeof(ecount));

// 2. SIGNED TO UNSIGNED CONVERSION RESULTS IN VERY LARGE VALUE
// 3. CHECK PASSED
  if (packet.getPayloadLen() < (SRTP_MAC_SIZE + 1)) {
    __android_log_print(ANDROID_LOG_WARN, TAG, "Packet shorter than MAC!");
    return -1;
  }

// 4. SECOND INTEGER UNDERFLOW
  HMAC(EVP_sha1(), parameters->macKey, SRTP_MAC_KEY_SIZE,
       (uint8_t*)packet.getSerializedPacket(), packet.getSerializedPacketLen() - SRTP_MAC_SIZE, ourMac, &digest);

  if (memcmp(ourMac, packet.getSerializedPacket() + packet.getSerializedPacketLen() - SRTP_MAC_SIZE,
      SRTP_MAC_SIZE) != 0)
  {
    __android_log_print(ANDROID_LOG_WARN, TAG, "MAC comparison failed!");
    return -1;
  }

  packet.setPayloadLen(packet.getPayloadLen() - SRTP_MAC_SIZE);

  AES_ctr128_encrypt((uint8_t*)packet.getPayload(), (uint8_t*)packet.getPayload(),
                     packet.getPayloadLen(), &key, iv, ecount, &num);

  return 0;
}
```

The integer underflow at 4. will cause a negative value to be passed on to the HMAC function:

```
unsigned char *HMAC(const EVP_MD *evp_md, const void *key, int key_len,
                    const unsigned char *d, size_t n, unsigned char *md,
                    unsigned int *md_len)
```

Since the parameter `n` is of type `size_t`, the value will be casted into a very large value near `LONG_MAX` (on 64bit architectures). This will for sure cause an out of bounds read, hitting unmapped memory addresses and therefore crashing the Signal application.

We currently consider this issue as non-exploitable in sense of creating a write primitive because of the second arithmetic underflow happening at 4. If the subsequent crash in HMAC would be prevented, things might be different depending on the creativity and skill. However there is still an impact now, as it is possible to remotely crash Signal.

Also if this code is used in other applications, other code paths might lead to exploitable conditions. Therefore we collected a list of all the consequences of this underflow

Methods `RtpPacket::getPayload()`, `RtpPacket::getSsrc()` and `RtpPacket::getTimestamp()` will return out of bounds memory on the heap behind *packet`:

```
uint16_t RtpPacket::getSequenceNumber() {
  RtpHeader *header = (RtpHeader*)packet;
  return ntohs(header->sequenceNumber);
}

int RtpPacket::getPayloadType() {
  RtpHeader *header = (RtpHeader*)packet;
  return header->flags & 0x7F;
}

uint32_t RtpPacket::getTimestamp() {
  RtpHeader *header = (RtpHeader*)packet;
  return ntohl(header->timestamp);
}

uint32_t RtpPacket::getSsrc() {
  RtpHeader *header = (RtpHeader*)packet;
  return ntohl(header->ssrc);
}
char* RtpPacket::getPayload() {
  return packet + sizeof(RtpHeader);
}
```

Method `RtpPacket::setTimestamp(uint32_t timestamp)` will write data out of bounds on the heap:

```
void RtpPacket::setTimestamp(uint32_t timestamp) {
  RtpHeader *header = (RtpHeader*)packet;
  header->timestamp = htonl(timestamp);
}
```

Method `RtpPacket::getPayloadLen()` will return a large positive number:

```
uint32_t RtpPacket::getPayloadLen() {
  return payloadLen;
}
```

Method `RtpPacket::getSerializedPacketLen()` will actually return the correct size of data received (due to a *reverse integer overflow* again):

```
int RtpPacket::getSerializedPacketLen() {
  return sizeof(RtpHeader) + payloadLen;
}
```

Please check your code in case you are using the RtpPacket class.

This issue has been fixed in the Signal Android client thanks to this commit.

## Timeline

- 2016-08-03 *Start of review*
- 2016-09-13 *Disclosure of initial findings to vendor*
- 2016-09-13 *Vendor releases and publishes two fixes*
- 2016-09-15 *Writeup release*

### pwnaccelerator BLOG

- pwnaccelerator BLOG

security and maybe more