# pwnaccelerator BLOG

**security and maybe more**

# Hunting For Vulnerabilities in Signal - Part 3

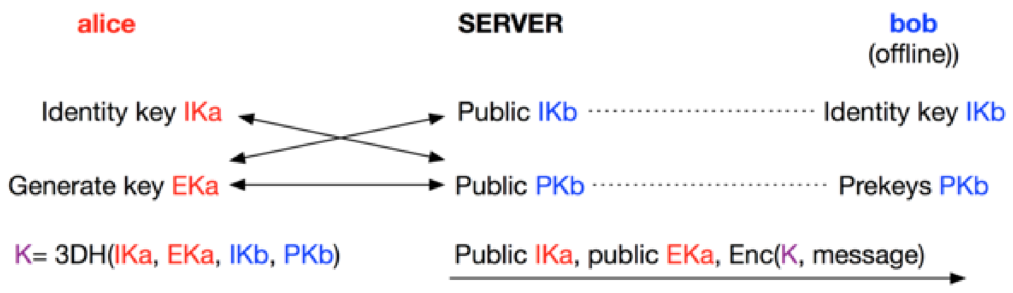
May 1, 2017

It's been a while since the original articles about vulnerabilities in Signal that were published together with @veorq on this blog. Following this we were honored to present this research as well as new findings at great conferences all over the world such as TROOPERS17, INFILTRATE, and HITBSECCONF2017.

We had some new things to present that sparked some discussion. An example is the lack of key validation done on public keys in Signal. This is discussed in-depth by @veorq in his blogpost.

Here I would like to talk about another really interesting finding that relates to the Signal protocol itself: **Message replay in a compromised server scenario.**

## X3DH Key agreement



Signal uses a variant of the Diffie–Hellman key exchange called the X3DH key agreement protocol also called "Extended Triple Diffie-Hellman". It is especially suited for use in asynchronous communication scenarios where one party may be offline during key exchange.

## Key-agreement overview

We will not cover the full X3DH protocol here, but focus on the parts that are interesting for a message replay instead:

1. When Alice wants to send a message in Signal to Bob, Alice will ask the Signal backend server for (public) cryptographic keys of Bob.
2. Bob may be offline at this time, therefore he already told the Signal backend his public identity key (long-term), a signed prekey, and several one-time prekeys. These values are stored on the Signal servers and may be requested by any other Signal user.
3. When Alice receives Bob's keys, she creates her own ephemeral key-pair (base-key). Then she will use the identity key, signed prekey, ephemeral-key (base-key), and one-time prekey to derive a shared secret using ECDH and encrypt a message with it.
4. Alice will send this message to Bob together with her identity key, the id of Bob's prekey, Bob' s signed prekey, and her ephemeral base-key.
5. Bob receives the encrypted message together with Alice's public values. He now has all the values (especially Alice's base-key) to derive the same shared-secret, and can decrypt Alice's message.

Alice and Bob keep track of their sessions in a session store. Signal allows to have more than one session in parallel.

## Reuse of keys

Obviously this process could be replayed since Bob already published his ephemeral key as a pre-key. It is fixed and since Bob is offline he cannot change it when Alice wants to talk to him.

This problem is somewhat known to the Signal vendor and discussed in the X3DH specification:

> 4.2. PROTOCOL REPLAY If Alice's initial message doesn't use a one-time prekey, it may be replayed to Bob and he will accept it. This could cause Bob to think Alice had sent him the same message (or messages) repeatedly. To mitigate this, a post-X3DH protocol may wish to quickly negotiate a new encryption key for Alice based on fresh random input from Bob. This is the typical behavior of Diffie-Hellman based ratcheting protocols [5]. Bob could attempt other mitigations, such as maintaining a blacklist of observed messages, or replacing old signed prekeys more rapidly. Analyzing these mitigations is beyond the scope of this document.
>
> 4.3. REPLAY AND KEY REUSE Another consequence of the replays discussed in the previous section is that a successfully replayed initial message would cause Bob to derive the same SK in different protocol runs. For this reason, any post-X3DH protocol MUST randomize the encryption key before Bob sends encrypted data. For example, Bob could use a DH-based ratcheting protocol to combine SK with a freshly generated DH output to get a randomized encryption key [5]. Failure to randomize Bob's encryption key may cause catastrophic key reuse.

It basically says replay is possible in X3DH if no one-time prekey is used in the initial message or no other mitigations are in place. This has also been researched by Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet in their paper.

Signal tries to do things right:

1. Signal always uses a one-time prekey.
2. Signal tries to remember the established sessions and the ephemeral values used by them. It will discard key exchanges if it has already seen the values before.
3. Additionally a session is only added to the session store if the key exchange messages are accompanied by an encrypted message and the decryption is successful.

## Sessions and the session store

Problem solved right? Not quite:

1. Even though Signal uses one-time prekeys that should be removed as soon as they are used, one of them is special. It is called the "last resort" prekey and is never removed as we can see in the code of Signal.
2. Signal will only remember a limited amount of previously seen sessions and their ephemeral keys, this limit is currently 40.
3. Not all parts of a Signal message are integrity checked.
4. A session will be persisted only if it is accompanied by an encrypted message that can be decrypted successfully. Yet this message can belong to **any** other session, there is no requirement that it actually belongs to the newly created session.

## Signal protocol implementation

Looking into a reference implementation of Signal (libsignal-protocol-java) there is some interesting code in file *java/src/main/java/org/whispersystems/libsignal/SessionBuilder.java* to be found:

```
private Optional<Integer> processV3(SessionRecord sessionRecord, PreKeySignalMessage message)
    throws UntrustedIdentityException, InvalidKeyIdException, InvalidKeyException
{
 ...
 if (message.getPreKeyId().isPresent() && message.getPreKeyId().get() != Medium.MAX_VALUE) {
    return message.getPreKeyId();
  } else {
    return Optional.absent();
  }
}
```

The method *SessionBuilder::processV3* returns the id of the prekey that was used, **except** if it is the last resort prekey.

The return value of method *SessionBuilder::processV3* is used to determine the prekey to be removed in method *SessionCipher::decrypt* (calls *processV3* internally) defined in file *java/src/main/java/org/whispersystems/libsignal/SessionCipher.java*:

```
public byte[] decrypt(PreKeySignalMessage ciphertext, DecryptionCallback callback)
    throws DuplicateMessageException, LegacyMessageException, InvalidMessageException,
           InvalidKeyIdException, InvalidKeyException, UntrustedIdentityException
{
    ...
    Optional<Integer> unsignedPreKeyId = sessionBuilder.process(sessionRecord, ciphertext);
    ...
    if (unsignedPreKeyId.isPresent()) {
      preKeyStore.removePreKey(unsignedPreKeyId.get());
    }
```

So if the id of the "one-time" prekey is MAX_VALUE (0xFFFFFF), the key is never removed from the prekey store and can be reused.

## Demonstrating a working replay attack

Let’s put theory aside and create a practical demo of this replay attack.

## Compromised server scenario

So what is the actual scenario here? The juice of Signal is that it should provide end-to-end security by employing cryptography. This means you don’t have to trust the Signal backend (in case you trust it, TLS should be enough for you).

So we will assume just that: *Mallory stole the TLS certificate for textsecure-service.whispersystems.org (WHOIS says it’s running on Amazon EC2!)*.

She can now happily MITM any traffic from Alice to Bob even when they are using unmodified Signal clients. It does not matter if they also checked their “Safety Numbers” (remember kids, safety first!).

## Defeating the mitigations

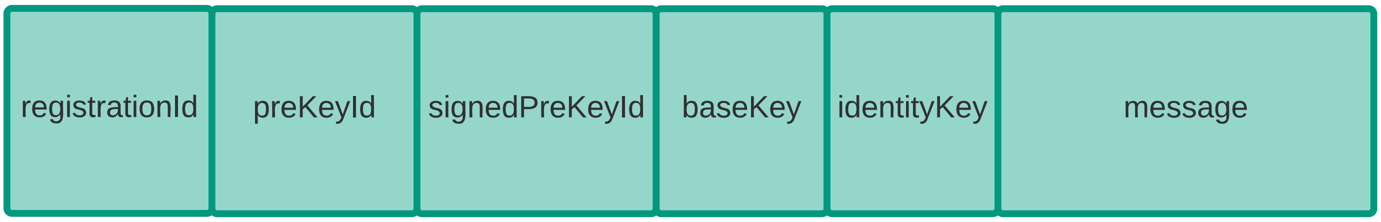
In order to conduct a message replay we have to defeat the mitigations of Signal mentioned above. We also use the limitations of the protocol to our advantage. Let’s look at that in more details:

### A. Forcing usage of the last-resort prekey

As seen above there is one key that is never removed from the prekey-store. As a MITM Mallory can force Alice to use this key over and over again. When Alice request the prekeys for Bob, Mallory will just delete all the other prekeys and only give the last resort key with id 0xFFFFFF to Alice. Alice has to believe this is the only key and will happily use it for all sessions with Bob.

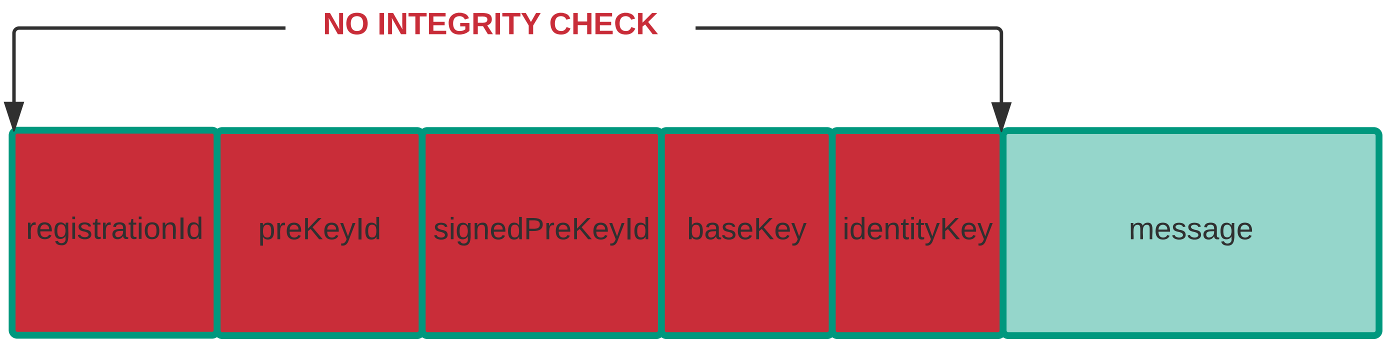
### B. Tampering key exchange messages

We mentioned that not all parts of a Signal message are integrity checked. Let’s have a look on how such an initial PreKeyMessage looks like:



As you can see the message includes an encrypted part *“message”* as well as key exchange values, ids of the prekeys used, Alice’s public identity, and the registration id of the corresponding Signal user.

Interestingly integrity checking via HMAC is only done on the encrypted part:



What does this mean for Mallory? Being a MITM she can change the *baseKey* value. This will result in a key exchange being conducted that results in a fake session. Mallory cannot really use this session since she does not have the private identity key and therefore does not know the shared secret. But the ability to create *“fake sessions”* alone comes in handy in the next steps.

### C. Persisting fake sessions that resulted from tampered key exchanges

Only if a key exchange (PreKeyMessage) is accompanied by an encrypted message that can be decrypted (and especially has a valid HMAC) successfully, a new session is persisted. As stated above, the encrypted message can belong to **any** other session, there is no requirement that it actually belongs to the newly created session.

So Mallory being a MITM just does the following: After the initial message she adds a her own fake key exchange to all encrypted messages sent from Alice to Bob. With each message a new session is created because the encrypted message is valid, and there is no integrity checking on the base keys and id values.

### D. Purging the session state

As mentioned, Signal will not create a new session if it already knows the base-key. Therefore we have to purge a session before it can be replayed. Combining the two previous steps, Mallory can force a certain session to be purged from the session store if:

- She forces the last resort key to be used for the target session (see first step).
- She modifies 40 messages to add her own key exchanges and create 40 fake sessions.
- The encrypted messages are encrypted for another session than the target session (because the target session is promoted and not purged otherwise) – this is for example the case if Alice or Bob hit the “Reset Session” button in Signal at any time.

### Replaying previous messages

After conducting the above steps, Mallory has now forced the target session to be purged on Bob’s side. Alice and Bob may communicate happily and will not see any warnings.

It is now time for Mallory to replay the initial message sent from Alice to Bob (Mallory saved this of course after forcing the usage of the last-resort key). Since the last-resort prekey was used session-creation succeeds, the same shared secret will be created, and the initial message will be decrypted successfully. This applies not just to the first message but also the following messages sent by Alice up to the next Diffie-Hellman ratchet step. Most probably individual messages can be omitted (but we did not test that).

The following video shows a replay attack conducted against a current Signal app (as of 2017-04-14) running on Android (sorry for the bad quality, I will probably create a better one if there is demand):

To conduct the attack and create a MITM condition, a single modification was done to add a CA certificate into the APK file. I did not have access to the Signal servers and did not expect Moxie to give us the keys to conduct our research. To determine how difficult it is to get a certificate from an Amazon EC2 instance is left as an exercise to the reader. Messages were sent from a third-party command-line client for ease of use. This has no impact on the attack carried out against the original Signal app. The message modification is conducted by a dedicated MITM tool written in Python. It will be released publicly so others can also do research on the Signal protocol.

## FAQ

### What is the impact?

- An attacker can replay the first messages of a session. The applies to all messages sent until the first Diffie-Hellman ratchet step, not only the first message. Depending on the type of communication severity may vary. For human to human text-only communication the impact is limited, but in machine-to-machine or group messaging contexts this property of the Signal Protocol might enable further attacks.

### Is this a vulnerability?

- IMHO - yes! Cryptographic communication protocols should not allow replay of messages. This is commonly a basic communication-protocol-property and there are explicit measures in the Signal protocol that try to prevent this such as message counters. Also it is explicitly mentioned in the X3DH spec.

### Does this require modifications to the original Signal app?

- No, the attack is purely MITM. The original app does not need to be modified in any way. In the demo we intercepted TLS with a rouge CA certificate for testing purposes. A malicious attacker does not need to do this if he stole the original certificate or controls the backend.

### Are all Signal implementations affected?

- Yes this is not bound to a specific platform. We did test on Android for ease of use, but the message replay is courtesy of the Signal Protocol and present in all reference implementations we checked. All applications that want to be compatible with the original Signal apps need to have partially unsigned envelopes.

### Is this a bug in the implementation or the protocol?

- Hard to say since the specification is not complete in all aspects. Section 4.2 of the X3DH spec clearly states that replay is possible *if there is no one-time prekey* used. The last-resort prekey is treated by Signal (as released by OWS) as a one-time prekey, yet it can be used several times. This is sort of a contradiction. Also not integrity checking key exchange messages enables this attack and potentially others.

### How can this be fixed?

- Short-term: The whole message envelope should be integrity checked, not just the encrypted part.
- Long-term: The concept of last-resort prekeys should be deprecated. A more secure fallback mechanism for key-exhaustion is recommended.

### Does key rotation mitigate a message replay?

- No since there is still a substantial time window for a replay attack until the key is actually rotated.
- Also a MITM attacker may even prevent the key rotation from succeeding, e.g. by causing an upload of new keys by Bob to fail.

### Did Open Whisper Systems acknowledge this as a vulnerability?

- **Update 2017-05-03**: Moxie Marlinspike asked us to clarify about the disclosure of this message replay. We did not talk with Moxie beforehand and he did not give any technical input to this post and did not acknowledge anything beforehand. The rationale behind disclosure is that the underlying issues are already known, discussed, and documented (see the linked X3DH spec, GitHub issue, blogpost, and papers). What has been shown here is a practical proof-of-concept for such a replay attack in a specific scenario and the limitations of the protocol. Apparently the last resort keys are currently being phased out, which is a good thing.
- ~~Not yet, there was no public statement to our research regarding the message replay. The potential for a replay was discussed in a public github issue by others and the usage of last resort keys criticised. Still it was deemed as **WONTFIX**. Maybe this will be reconsidered given a practical attack has now been demonstrated publicly.~~

### pwnaccelerator BLOG

- pwnaccelerator BLOG

security and maybe more