Help, my Signal has bad Device!

Breaking the Signal Messenger's Post-Compromise Security through a Malicious Device

Jan Wichelmann ✉, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth

University of Lübeck, Germany
{j.wichelmann,s.berndt,c.pott,thomas.eisenbarth}@uni-luebeck.de

Abstract. In response to ongoing discussions about data usage by companies and governments, and its implications for privacy, there is a growing demand for secure communication techniques. While during their advent, most messenger apps focused on features rather than security, this has changed in the recent years: Since then, many have adapted end-to-end encryption as a standard feature. One of the most popular solutions is the Signal messenger, which aims to guarantee forward secrecy (i. e. security of previous communications in case of leakage of long-term secrets) and future secrecy (i. e. security of future communications in case of leakage of short-term secrets). If every user uses exactly one device, it is known that Signal achieves forward secrecy and even post-compromise security (i. e. security of future communications in case of leakage of *long*-term secrets). But the Signal protocol also allows for the use of multiple devices via the Sesame protocol. This multi-device setting is typically ignored in the security analysis of Signal.
In this work, we discuss the security of the Signal messenger in this multi-device setting. We show that the current implementation of the device registration allows an attacker to register an own, malicious device, which gives them unrestricted access to all future communication of their victim, and even allows full impersonation. This directly shows that the current Signal implementation does not guarantee post-compromise security. We discuss several countermeasures, both simple ones aiming to increase detectability of our attack, as well as a broader approach that seeks to solve the root issue, namely the weak device registration flow.

1 Introduction

Messenger apps like Whatsapp, WeChat or Telegram have become a cornerstone of person-to-person communication in the past decade. To meet users demand for privacy and to protect their right for freedom of expression, many messengers now employ end-to-end encryption (E2EE) to ensure message privacy. E2EE also ensures that operators cannot pry on users communication and thus poses new challenges to government surveillance. With the popularity and better protection of communication, governments and their police forces fear *going blind* and try to regain access via jurisdiction and/or improved technical capabilities. For

example, Russia banned Telegram for several years, due to its use of E2EE [23]. But political will to push regulation and/or improve technical capabilities also exist in the US [19] and the EU [8].

One messenger and its same-named secure communication protocol stands out: Signal. The Signal protocol has received great scrutiny by the crypto community and is widely accepted as providing a very high level of security. In fact, Whatsapp adopted the Signal protocol to restore user trust after being bought by Facebook in 2014. One of the features that make the Signal protocol special is its *future secrecy* property, which—in addition to protecting all communication completed *before* a breach of local credentials—also provides security guarantees in case the *short-term keys* of a system were leaked [25]. Furthermore, the specification states that the protocol achieves some sort of security against passive attackers that were able to compromise one of the parties, but not against active attackers [27]. This is a weaker notion than that of *post-compromise security*, which also protects all communication if the *long-term keys* are leaked [16]. Post-compromise security seems particularly desirable in a world where governments invest heavily in the ability to intercept messenger communication. But it can also restore trust in cases where long-term secrets have been compromised due to a malware infection, leakage of backups, or legal reasons [17,12]. For most messengers, even a short-term compromise results in insecure subsequent communication if long-term keys could be leaked (see e. g. the comparison in [18]). Most of the messengers that restore security toward a broad class of adversaries even with compromised secrets are based on the Signal protocol. Furthermore, it was shown that Signal does indeed guarantee the stronger notion of post-compromise security, in the one-device-per-user use case [9,15].

Multi-device support is handled by the Sesame sub-protocol in Signal. Sesame adds a new level of complexity to the protocol, which is often not reflected in current cryptographic analysis [9,15]. Unlike other parts of the Signal protocol, the Sesame specification is less precise and leaves a lot of freedom to the actual implementation of the protocol. Whether Signal achieves post-compromise security in the general case of users having multiple devices is thus not as clear. In [16], the authors state that TextSecure — the predecessor of Signal — might not achieve post-compromise security due to its implementation of the handling of multiple devices. The authors of [15] state that the post-compromise security of Signal depends on subtle details related to device state reset and the handling of multiple devices, but that Signal could achieve some form of it.

Just recently, the question whether implementations of the Signal protocol do have post-compromise security was answered in [18]: The authors argue that the Signal *protocol* does guarantee post-compromise security, but several prominent implementations either do not guarantee it at all (e. g. WhatsApp and Facebook Secret Conversations) or only partially (e. g. Signal messenger), due to their problematic handling of desynchronization scenarios. More concretely, the authors clone a device and later try to use this clone. Whenever the clone sends a message, the receiving party just displays a message “Bad encrypted message”. Similarly, whenever this clone receives a message, only the message

"Bad encrypted message" is displayed. In both cases, the sending party does not receive any notification about this. This behavior means that the cloned device can impersonate the original device, but can not be used to send or receive messages, making it rather useless.

In this work, we present an attack on the post-compromise security of the Signal messenger that allows to stealthily register a new device via the Sesame protocol. In contrast to the attack in [18], this new device can *send and receive messages* without raising any "Bad encrypted message" errors. Our attack thus shows that the Signal messenger does not guarantee post-compromise security at all in the multi-device setting.

1.1 Our Contribution

This work analyzes the Sesame protocol as it is implemented in the current version of the Signal messenger. As many parts of Sesame are not specified out, we reverse-engineer specific implementation details of Sesame in the Signal messenger. With those gaps in the Sesame specification filled, we analyze the *post-compromise security* property of the Signal protocol, which indeed holds in the single-device per user scenario [15]. However, we show that the current implementation of the Signal messenger, due to unfortunate choices in the Sesame realization, undermines the post-compromise security and may ease interception of messenger communication. We further point out how simple changes in the realization of Sesame can be used to close these existing gaps. In summary, we

- give an overview of the Signal protocol suite, and discuss its security in case one of a user's devices gets temporarily compromised;
- highlight security-critical steps that have been declared implementation details and thus were left out from the protocol specification;
- show that in the current implementation, an attacker can fully break post-compromise security by leaking only two long-term secrets, and using these to register a new device;
- discuss several mitigations to help users detect our attack, and to fix the underlying issue in order to allow secure registration of devices.

1.2 Responsible Disclosure

We disclosed our findings to the Signal organization on October 20, 2020, and received an answer on October 28, 2020. In summary, they state that they do not treat a compromise of long-term secrets as part of their adversarial model. Therefore, they do not currently plan to mitigate the described attack or implement one of the proposed countermeasures.

2 Background

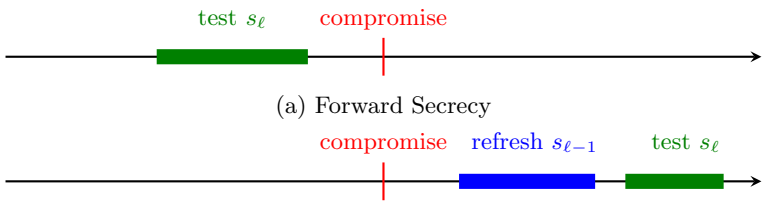
We always denote key-pairs by capital letters. For a key-pair IK, we denote the secret key by $\mathrm{sec}(IK)$ and the public key by $\mathrm{pub}(IK)$. Symmetric keys are denoted by lowercase letters, e.g. sk.

Nowadays, instant messaging is omnipresent and such messengers often even replace the use of e-mail in companies (examples include Slack [4], Microsoft Teams [2], or Webex Teams [6]). Over the last decades, many different cryptographic protocols for secure messaging were developed. Due to the rapid technical development, many new features were added to the applications implementing these protocols, but their security guarantees were rarely updated as well. Two common features leading to security problems are *group communication* and *multi-device communication* (see also the discussion in [13] for important differences between these scenarios). In the case of group communication, multiple users want to communicate in a group. Furthermore, these groups are typically dynamic, i. e. the users in a group can and do change relatively often. To circumvent the arising problems with group communication, the *Messaging Layer Security (MLS)* protocol was introduced and is currently in the standardization process [5,14]. In the case of multi-device communication, two users want to communicate, but each of them may use different devices (such as a laptop, a smartphone, and a tablet). Furthermore, users typically want to register new devices, transfer old messages, and have a synchronized status on all of their devices. To the best of our knowledge, there is no proposal for a unified handling of multiple devices.

In this work, we only consider the multi-device setting and the problems arising in this scenario. For simplicity, we focus on two-user communication, i. e., two users A (or Alice) and B (or Bob) communicate, but each of the users owns several devices.

2.1 Post-Compromise Security

Modern cryptographic protocols aim to achieve different security guarantees, depending on their use case. One of those guarantees is the security in case the long-term keys of a party are leaked. Two important notions dealing with this are *forward secrecy* and *post-compromise security*: Forward secrecy (typically achieved by the use of ephemeral keys) guarantees that *previous* communication is still confidential, even if the long-term keys of the parties are leaked (see Fig. 1a).



(a) Forward Secrecy

(b) Post-compromise Security

Fig. 1: Schematic representation of forward secrecy and post-compromise security.

In contrast, post-compromise security guarantees that leakage of the long-term keys does not break the confidentiality of *future* communication (see Fig. 1b). Clearly, the general goal of universal post-compromise security is not achievable. If A and B have not communicated before, an attacker knowing the long-term identity key $\mathrm{sec}(IK_A)$ of A can perfectly impersonate A and is thus able to perform a man-in-the-middle attack, breaking the confidentiality of the communication between A and B. But, only slightly weaker guarantees are still possible: If A and B have already communicated before, they might have agreed on an (ephemeral) key $EK_{A,B}$ during this session. Whenever A and B now want to resume their communication, A uses both $\mathrm{sec}(IK_A)$ and $EK_{A,B}$ to authenticate themselves. Clearly, having only access to the long-term key $\mathrm{sec}(IK_A)$ is thus not sufficient to break the confidentiality of the communication in this scenario.

In [16], Cohn-Gordon, Cremers, and Garratt formalized this above intuition both about the impossibility of universal post-compromise security, but also on the possibility of slightly weaker versions. Informally, they show that even if all but one exchange of messages before the secure session are compromised, post-compromise security can still be achieved.

Note that post-compromise security can be useful for a wide range of situations, not only for a complete breach of a device: For example, an old backup containing the long-term keys might have been leaked (see e. g. [3]), malware was present on the device (see e. g. [1]), parts of the implementation were manipulated (see e. g. [11,10]), or a secondary device might have been stolen.

Attacker model Here, we only give an informal discussion about the formalization of post-compromise security and refer the interested reader to [16] for formal definitions. Alice and Bob communicate via a sequence of *sessions*, which can be thought of as runs of an authenticated key exchange protocol. Each session s has its own *local state*, which includes e. g. the session key EK_s, the parties A and B, the randomness used in s, and all messages exchanged during the session. The parties A and B have a *global state*, which includes e. g. the long-term secrets IK and the public keys of all other parties.

Now, consider a sequence of sessions $s_1, s_2, \ldots, s_\ell$ between A and B, where the final session s_ℓ is the *test session*. The goal of an attacker is to break the confidentiality of this test session. To do so, we assume that an attacker can obtain the long-term secrets and the short-term secrets of all sessions, *except* for the session $s_{\ell-1}$ (which can be used as a refresh session) and the test session s_ℓ itself (see Fig. 1b) We furthermore assume that an attacker has the usual abilities: They can read all of the (encrypted) messages sent between A and B, are a valid user in the network, and can communicate with both A and B.

Multi-device support As noted before, we consider the situation that both A and B communicate via multiple, different devices. This multi-device setting already leads to non-trivial problems with regard to post-compromise security. Consider a single-device communication protocol Π that has post-compromise security. In order to adapt Π to a multi-device setting, several questions arise:

- How to synchronize the different devices of a single user?
- How to register a new device?

To still guarantee the post-compromise security of the multi-device protocol, these questions (and many more) need to be answered carefully.

To handle the synchronization of the different devices of a single user (and also handle asynchronous messaging), one could make use of a server. For each user A of the system, this server manages a *mailbox*, which stores all messages sent to A, all messages received by A, and all registered devices of A. The messages are stored encrypted. Now, whenever A uses one of their devices to send a message to B, the device would put this message in the mailbox of A and in the mailbox of B, if this device was successfully registered for A. To synchronize received messages, every successfully registered device of A could obtain and decrypt the content of the mailbox of A.

A straight-forward way to register a new device would be using the long-term secret key $\mathrm{sec}(IK_A)$ to add the new device to the device list of the mailbox of A. Unfortunately, such a strategy might already break the post-compromise security of the protocol: If, apart from knowing $\mathrm{sec}(IK_A)$, no further verification from A is required for such a registration, an attacker knowing $\mathrm{sec}(IK_A)$ can register a new device without alerting A. From this point on, the attacker would be able to observe the complete communication of A, thus breaking the post-compromise security of the protocol. The Sesame protocol used by Signal to handle multiple devices roughly follows this approach and, as we will show, is thus not post-compromise secure.

3 The Signal Protocol

The *Signal* (formerly *Axolotl*) protocol [26] provides end-to-end encryption for text messages and multimedia files. It is widely used in different communication applications such as WhatsApp [33], Skype [24] and the Signal messenger itself. The protocol is based on the Double Ratchet algorithm and uses a triple Elliptic-curve Diffie–Hellman handshake (X3DH) to initiate new conversations. The Sesame protocol is used to enable multi-device support. Signal uses a number of cryptographic primitives including

- Elliptic Curve Diffie-Hellman functions (implemented by X25519 or X448 [21]);
- a signature scheme called XEdDSA producing EdDSA-compatible signatures from X25519 or X448 using the hash function SHA-512 [26];
- a hash function (implemented by SHA-256 / SHA-512);
- a key derivation function KDF based on the HKDF algorithm [20];
- an authenticated encryption (AEAD) scheme [31,32]. Concretely, KDF is used to produce an encryption key, an authentication key, and an initialization vector (IV). The plaintext is then encrypted with AES-256 in CBC mode. Finally, HMAC with the hash function and the authentication key is used on the authenticated data.

We continue by giving an overview over the three protocol parts that jointly form the Signal protocol. For the remainder of this paper we use the term *user*

for one communicating entity that usually is a single person. Note, that one user may have multiple devices, that they use for their communication. The term *party* on the other hand is used more abstractly on the protocol level for one side of the communication, usually represented by a single device or server.

3.1 X3DH [29]

In order to setup a secure session, all parties have to agree on a key. Usually, this is done via a Diffie-Hellman key exchange, but this does not work well in a messenger setting, which heavily relies on asynchronous communication. If party A wants to send a message to party B, but party B is offline, party A needs a way to derive a shared secret key sk without any interaction with party B.

The X3DH protocol aims to solve that problem, by allowing B to store a set of public keys in a public location, which A can subsequently use for a Diffie-Hellman computation. In order to provide authentication and freshness, B offers their public identity key and a set of *prekeys*. A retrieves B's public keys and computes DH key exchanges with their own secret identity key and an ephemeral key. To allow B to later derive the same shared key, A subsequently sends their public identity key and the public ephemeral key. A can now encrypt messages with the shared key and send them to B. As soon as B gets online again, they can use A's public keys to derive the same shared secret and decrypt A's messages. In order to encrypt and send messages to A, B executes the same protocol steps as A, deriving another shared secret for the other direction of communication.

3.2 Double Ratchet [27]

While agreement on a shared secret key is sufficient for A and B to exchange encrypted messages, it is quite vulnerable against possible compromise: As soon as the shared key gets leaked, an attacker gains full access to all past and future communication between A and B. To avoid this, the shared key needs to be refreshed in regular intervals, to add new randomness and narrow down the possible damage from a leaked secret.

The Double Ratchet protocol solves this by introducing four cryptographic chains. The first one, the *Diffie-Hellman (DH) chain*, consists of an alternating series of public and private ephemeral keys, where the private part is provided by the local party, and the public part comes from the remote party. Ideally, each message sent from A to B also contains a new public ephemeral key from A, and vice versa. Each time a party receives a new public ephemeral key, they advance their local DH chain by one step.

The shared secret from the DH chain is then fed into a symmetric *root chain*, which is initialized with the initial secret from the X3DH key exchange. On each step of the DH chain, the root chain is advanced by one step as well. The root chain uses a keyed hash function to generate a *root key*, which is used as key for the next root chain step, and a *chain key* for sending or receiving. Each chain key spawns a new sending or receiving chain, which is in turn used to derive the keys for encrypting or decrypting messages. Since all chain keys are derived

using a keyed hash function, an attacker cannot compute their predecessors, so the protocol grants forward secrecy.

Note that each DH chain step leads to a new sending or receiving chain. Thus, if a sending or receiving chain key gets compromised, only the messages encrypted with that particular chain are affected. The same holds for the root chain: If a root key gets compromised, only the immediate sending or receiving chain and its associated messages are affected. The next step of the root chain can be considered secure again, as it incorporates fresh randomness from the DH chain. In case a long-term secret (e.g., the private identity key) gets leaked, the confidentiality of future messages from existing Double Ratchet sessions is still preserved, as long as the attacker does not also gain access to all new private ephemeral keys. Thus, intuitively, the Double Ratchet protocol also provides post-compromise security. For a formal security proof of Signal's forward secrecy and post-compromise security we refer to [15].

3.3 Sesame [28]

In order to allow users to send and receive messages from multiple devices, the Sesame protocol was introduced. The protocol describes two scenarios: A *per-user* scenario, where a single identity key is used on all the user's devices, and a *per-device* scenario, where each device has its own identity key. Both scenarios are handled in a similar fashion by the Sesame protocol, since the only difference is the location where the identity keys are stored – either in the user records or in the device records.

On the highest level, each device stores a list of users that it knows, including its owner. For each of these users, a non-empty list of their devices is stored, which in turn is associated with a list of Double Ratchet sessions. Additionally, each device has its own *mailbox* on the server, which is used to asynchronously fetch encrypted messages from other devices and which only contains messages that weren't yet received. For each device, exactly one session is *active* at a time, while the other ones are *stale* and only kept in case delayed messages arrive.

Whenever a device of user A sends a message to user B, it sends this message to each device associated with B, either via its current active session or by initializing a new session via X3DH. Additionally, the message is sent to all devices of A, using the same mechanism. The server then puts the messages into the respective mailboxes, where the receiving devices can obtain their messages from and then decrypt them using the corresponding session keys.

While Sesame describes how messages are kept synchronized on all devices in a multi-device scenario, it does not cover the registration of new devices: These details are fully left to the implementation, excluding them from considerations regarding Sesame's security. In Section 4, we show that the current implementation in the Signal messenger is indeed vulnerable, and allows an attacker to impersonate their victim.

4 Signal Implementation in the Signal Messenger

While the specification of the X3DH key exchange and the Double Ratchet are rather specific, Signal's multi-device extension Sesame only describes a high-level view of exchanging messages between multiple devices and sessions (compare e. g. [27] and [28]). Many important details, most notably the registration of new devices, are left to the programmer, and are thus not included in any security proofs. In this section, we take a closer look at these implementation details and show how these allow an attacker to work around Signal's post-compromise guarantees, gaining unconstrained access to a user's future communication. Our attack shows that the Signal messenger currently does not guarantee post-compromise security. Furthermore, in contrast to [18], our attack allows us to completely break the privacy of the communication, as it allows us to both send and receive messages.

For simplicity, we assume a single user with identity key IK and who uses the Signal app A. The device registration aims to add a new device D.

4.1 Reverse Engineering the Protocol Implementation

Since the device registration protocol is not specified anywhere, we had to analyze how it is implemented in the Signal messenger. Unfortunately, there is neither an official API specification nor any documentation of the procedure, so we had to dive into the implementation and try to piece together the relevant bits in order to get a full view of the device registration protocol and do a security analysis.

For our analysis, we checked out the source repositories of Signal's Android app (commit `fc41fb5`[1]) and Desktop client (commit `a1721ed`[2]). Apart from occasional source code comments, both implementations are almost entirely undocumented, and it proved difficult for us to get an overview by inspecting the various subfolders/packages. In order to roughly locate the relevant code parts, we searched for various strings which are shown in the UI, and then followed the call traces.

There is no built-in means for exporting sent/received packets in debug mode; Signal does certificate pinning with a custom TLS root certificate, which we weren't able to circumvent without losing connectivity, so setting up a proxy for intercepting the network communication was not an option. Thus, we mostly resorted to static analysis in order to understand what data is sent across the network, along with some custom debug outputs. Studying the server implementation[3] helped us infer the higher level information flow.

We lay out our reverse engineering results in the following section, where we explain the device registration process and the involved secrets.

[1] https://github.com/signalapp/Signal-Android/tree/fc41fb5
[2] https://github.com/signalapp/Signal-Desktop/tree/a1721ed
[3] https://github.com/signalapp/Signal-Server

4.2 Device Registration

Prerequisites To register a new device, several private and public values are required, which may be partially known to an attacker:

- The (private) identity key *IK*: As described in Section 3.1, this long-term key is required to setup new prekeys and start new conversation sessions. It is only stored on the user's device.
- The phone number *pn*: As, for the current implementation, the phone number is the only means for creating and identifying user accounts, it can be assumed to be known to the attacker.
- The app's API username un_A and password pw_A: These are used in HTTP authentication when communicating with the server. The username directly depends on the phone number and the device ID (which is constant for the primary device), and can thus be easily guessed; the password is random and needs to be leaked. Since the authentication data is sent in the clear, but inside the TLS layer, the attacker may either exfiltrate it through the same channel as the identity key, or by gaining (limited) access to the server, which is assumed untrusted by the Signal protocol.
- The profile key *pk*: The profile key allows accessing certain meta information, like the user's display name and their avatar image. It usually is transmitted when starting a new conversation, to allow the other peer to download and decrypt the user's profile information, so the attacker may have already gained access to that key by communicating with their victim at an earlier point of time. Anyway, we found that sending the profile key is optional for device registration, and does not influence detectability of our attack.

Adversarial Scenario We now concretize our generic attacker model, which we presented earlier. Throughout the rest of the paper, we assume an adversary who at some point managed to obtain the private identity key *IK*, the phone number *pn*, the API username un_A, and the password pw_A. After all of these information are retrieved, the attacker does not interact directly with the victim or interferes with their communication. Instead, the adversary will only interact with the public Signal servers once during the compromise stage, using these cloned credentials of the victim to impersonate the victim towards the server (but not toward any of the communication partners of the victim). Once registered, the adversary performs direct communication with a party (the Signal server), to collect messages from their mailbox. This corresponds to the scenario described in [18], where an attacker was able to clone the complete smartphone and uses this cloned copy at a later time.

The PIN Another secret, which is only known to the user, is the PIN *pin*: Signal PINs were introduced in 2020 [30], and are designed as a means for storing private information in an untrusted location. This information may be later used to recover key material and the contact list, e.g., after losing the primary phone (*Secure Value Recovery* [22]). The PIN cannot be acquired by breaking into the

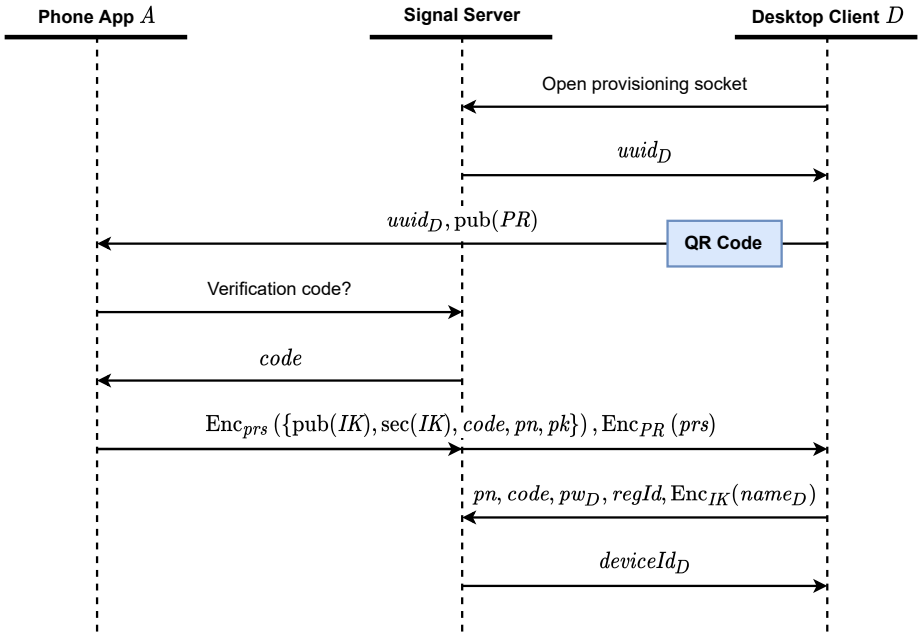


Fig. 2: The Signal device provisioning protocol. The user registers a new device D with their primary device A.

server, as it is claimed to be secured by an SGX Enclave, which only permits a small number of guesses. However, the current implementation of the app offers a PIN "reminder" feature, which asks the user to enter the PIN in regular intervals. This feature compares the hash of the entered PIN to a locally stored value, in order to avoid accidentally using up the number of allowed guesses on the server side. If an attacker manages to retrieve this hash value, e.g. by dumping the memory of the app, they may be able to determine the (likely short) PIN through an offline brute-force attack. However, knowing the PIN is not required for our attack, except if the attacker wants to obtain the full contact list of their victim.

Device Provisioning Protocol The protocol for registering a new device, called *provisioning* by the Signal implementation, is illustrated in Figure 2. As stated before, we use A to denote the phone (primary) instance of the user's Signal account, and D to denote the desktop client instance which the user tries to register as a new device.

Upon start of the desktop client, the software will open a *provisioning* WebSocket to the Signal servers, which will generate and send a random device UUID $uuid_D$. The desktop client then generates a provisioning key pair PR and encodes $uuid_D$ and $\text{pub}(PR)$ into a QR code, which is presented to the user.

After the user scanned the QR code using the Signal app, the app will first request a verification code *code* from the server, and then encrypt some of the app's private data with a random AES key *prs*:

$$\mathsf{Enc}_{prs}\left(\{\mathrm{pub}(IK), \mathrm{sec}(IK), code, pn, pk\}\right)$$

The encrypted data and the encrypted key $\mathsf{Enc}_{PR}(prs)$ is sent to the server, which relays it via the provisioning socket to the desktop client. The desktop client uses the private provisioning key $\mathrm{sec}(PR)$ to obtain *prs* and thus decrypt the data packet sent by the app.

The desktop client registers with the Signal servers by sending a packet containing the phone number *pn*, the string *code* for verification, a random password pw_D, a random registration ID *regId*, and the device name $name_D$, which is chosen by the user and is encrypted using the identity key *IK*. Upon receiving the registration packet, the servers return a new device ID $deviceId_D$, completing the protocol.

Since the Signal servers require HTTP authentication, the desktop client will include the username $un_D := pn.deviceId$ and the password pw_D in any future communication.

After the registration is done, the desktop client requests the current list of conversations, which is implemented via a hidden Double Ratchet session between the desktop client and the app. This "shadow" session is also used to synchronize messages sent by the user between their devices. There is no notification to the user that a device requested their conversations. Note that only the conversation metadata and the lists of participants are transmitted; the data does not include the chat history prior to device registration.

4.3 Registering a Malicious Device

If an attacker manages to temporarily compromise the victim's primary device in a fashion that reveals certain private values, namely the victim's identity key *IK* and the API password pw_A, they can simulate a device registration and add a malicious device.

As illustrated in Figure 2, the only points where the primary device interacts with the server during the device registration are requesting a verification code and sending the encrypted private data. The former only requires API credentials, while the latter additionally requires the private identity key. Since we assume that the attacker has gained access to these values, they can fully emulate the protocol and set up the new device, without any interaction from the victim or their app.

To demonstrate malicious device registration, we created a simple dummy app[4] in C#, which takes the private identity key and the API credentials, and then runs the described API calls. For testing, we started a new instance of the official desktop client, extracted the contents of the displayed QR code, and fed

[4] Code is available on GitHub: https://github.com/UzL-ITS/signal-attack

these into our dummy app. As soon as the API calls were completed, the desktop client started downloading the conversations from the victim's phone app. As the phone app happened to be up and online, all contacts and groups were successfully retrieved, without showing any notification to the victim (Figure 3). After the registration was completed, the victim and their peers started to forward new messages and conversations to the forged device as well, giving the attacker full access to their communication.

Since the server manages a list of all registered devices, the forged device will appear in the victim's device list, if they access it in their app. However, in case the attacker has some level of control over the (untrusted) server, they can easily manipulate the returned list to exclude their forged device, making the attack almost undetectable.

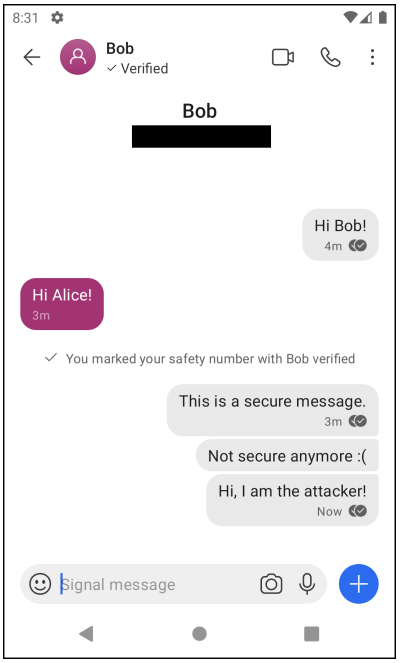
4.4 Implications for post-compromise security

The newly added device gives the attacker a high level of access to all communications of the victim.

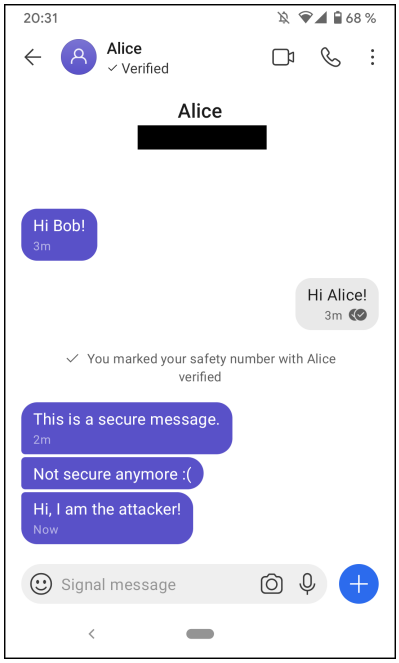
According to the Sesame specification, new messages shall be sent to the active sessions of each of the peers' devices, so each device can display the entire chat history from the point of device registration, even if the user switches their active device in between. Thus, the attacker receives all new communication directed to their victim, as well as all messages sent by the victim to other devices, as chat history is kept synchronous between all devices of a user.

The attacker may also impersonate the victim and send messages on their behalf. As sending such a message is easily detectable by other synchronized devices, one might suppress the synchronization to the victim's other devices. However, this may be detected as soon as one of the peers responds, since an answer will be sent to all registered devices, including the victim's own ones.

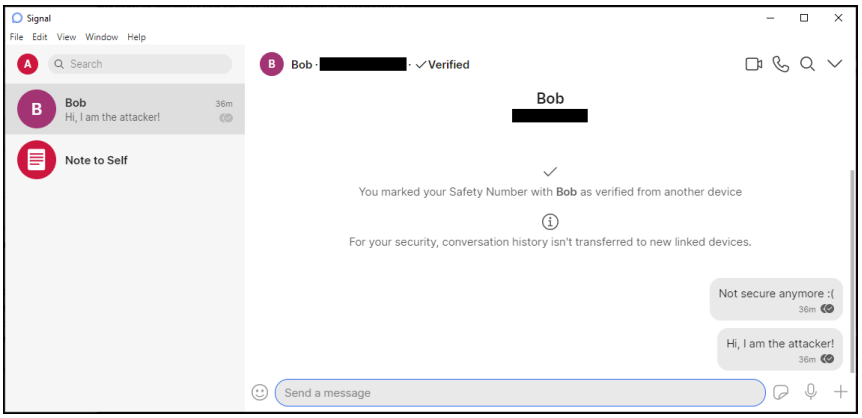
In summary, our attack shows that compromising just two secret values leads to a full disclosure of all future communications. Previously, it was only known that a cloned devices could be used, but this device was not able to send or receive messages [18]. In contrast, our attack shows that leaked long-term keys of Signal can directly be used to completely break the post-compromise security, both in theory *and* in practice. While Double Ratchet itself has strong post-compromise guarantees, this is subverted by the weak device registration and synchronization procedure in Signal's implementation of the Sesame protocol.



(a) Alice's view.



(b) Bob's view.



(c) Attacker's view.

Fig. 3: View of a conversation between Alice and Bob (screenshots (a) and (b) slightly shortened to save space, by removing the user icons). After Alice initiated the conversation and Bob answered, both verified their safety numbers. After another message ("This is a secure message"), Alice's account got compromised. The attacker installed another device, and was able to read Alice's last message ("Not secure anymore"). Finally, the attacker impersonated Alice and sent a message on her behalf. There is no indication to Bob or Alice that a new device was added to the conversation, or which device authored a given message.

5 Countermeasures

To counter the attack described above, there are several possibilities. Note that according to the assumptions of the Signal protocol, we do not trust the server. This lack of trust makes fixing the above problems much harder.

5.1 UI Changes

Right now, there is no active indication that a new device has been added, neither for the user themselves, nor for their peers. While indicating a change or addition of devices does not mitigate our attack, it greatly enhances detectability.

Notification for own device addition Currently, the attack can only be detected when the victim checks their device list and spots the malicious device. Since their own devices automatically sync settings and messages to all other devices, they need to have a current list of all devices at all times. Whenever a new device appears in this list, which the user did not actively add via scanning a QR code, their primary app should issue a high-priority notification in order to warn the user of a possible breach. However, this only works if the attacker is not able to somehow suppress their malicious device at the server-side, which would hide it from the victim, at the cost of not receiving the victim's own messages anymore.

Notification of device list changes In order to communicate in accordance with the Sesame protocol, each user maintains ratchet sessions with all devices of their peers as well as all other devices of themselves. As soon as a new device is added, this may be indicated in the chat history, if the users choose to opt-in to such a feature (there may be privacy concerns). If, in order to escape such measures, an attacker fully suppresses their device at the server-side, they won't receive any messages, making the attack useless.

Device-specific security codes Another optional extension are device-specific security codes: Instead of verifying a user once and then trusting all their devices, the conversation peers could do a pair-wise verification of all devices. In this setting, messages are only sent to authenticated devices, and a warning is issued whenever a non-verified device is present. This approach is taken e.g. by the Matrix messenger platform [7], which allows either a device-based or a user-based verification.

5.2 Alternative Multi-Device Protocols

An alternative, more radical approach would be replacing the Sesame protocol itself. Just recently, Campion, Devigne, Duguey, and Fouque devised a replacement protocol for Sesame [13]. While Sesame realizes multi-device support by

opening separate Signal channels between all devices used by the two communicating parties, the approach presented in [13] uses a single Signal channel for all communication between two users A and B. The usage of multiple devices on A's side is therefore transparent to B (and vice versa), meaning that higher privacy guarantees are achieved. A's devices can only use the same Signal channel to communicate with B, if all of them use the same double ratchet session, which can only be achieved when they synchronize the used ratchet key. In order to achieve such a synchronization the authors introduce the Ratcheted Dynamic Multicast (RDM). Based on asymmetric keys that are renewed regularly (hence ratcheting), this allows every device of A to use a non-interactive multicast channel to send session updates to all other devices owned by A, while providing forward secrecy and post-compromise security (called healing properties in [13]).

In our attack we exploited the device registration and management, which are mostly handled by the Signal server. The device registration presented in [13] enforces the use of another already registered device. This is possible because the registered devices of a user keep track of all other registered devices in a decentralized fashion. If A wants to register a new device, they must use one of their registered devices, which in turn uses the RDM to notify all other devices of A about the new device. Note that the RDM can only be used if the current ephemeral keys are known, which means that an attacker who has only extracted long-term secrets is not able to register a new device.

6 Conclusion

In this work, we presented a security analysis of the Sesame protocol and its current implementation in the Signal messenger, with focus on post-compromise security. To enable a detailed security analysis, we first had to reverse-engineer several implementation details of the Signal messenger. Based on the detailed knowledge, we showed that the multi-device support of the Signal messenger can be abused to eavesdrop on all communication after a one-time credential breach. Thus, currently, the Signal messenger does not provide message privacy in the post-compromise security scenario. We further discussed possible mitigations of the described attack, where some are easy to implement and have minimal impact on the user experience of the Signal messenger, while providing enhanced detectability of our attack.

Acknowledgements

This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under Grant No. 427774779 and through the ERDF project EMSIK.

References

1. Barcode Scanner app on Google Play infects 10 million users with one update . https://blog.malwarebytes.com/android/2021/02/barcode-scanner-app-on-google-play-infects-10-million-users-with-one-update/, accessed 2021-02-22

2. Microsoft Teams . https://teams.microsoft.com, accessed 2021-02-22
3. More Keys Than A Piano: Finding Secrets In Publicly Exposed Ebs Volumes . https://www.defcon.org/html/defcon-27/dc-27-speakers.html#Morris, accessed 2021-02-22
4. Slack . https://slack.com/, accessed 2021-02-22
5. The Messaging Layer Security (MLS) Protocol (11) . https://tools.ietf.org/id/draft-ietf-mls-protocol-11.html, accessed 2021-02-22
6. Webex Teams . https://teams.webex.com, accessed 2021-02-22
7. Matrix. https://matrix.org/, accessed 2021-02-16
8. Council resolution on encryption. Council of the European Union, November 24 (2020), https://data.consilium.europa.eu/doc/document/ST-13084-2020-REV-1/en/pdf. Accessed 2021-02-22
9. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: EUROCRYPT (1). Lecture Notes in Computer Science, vol. 11476, pp. 129–158. Springer (2019)
10. Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: Proc. CCS. pp. 1431–1440. ACM (2015)
11. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Proc. CRYPTO. Lecture Notes in Computer Science, vol. 8616, pp. 1–19. Springer (2014)
12. Bergman, R., Fassihi, F.: Iranian hackers found way into encrypted apps, researchers say (2020), https://www.nytimes.com/2020/09/18/world/middleeast/iran-hacking-encryption.html. Accessed 2020-10-13
13. Campion, S., Devigne, J., Duguey, C., Fouque, P.: Multi-device for signal. In: ACNS (2). Lecture Notes in Computer Science, vol. 12147, pp. 167–187. Springer (2020)
14. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: CCS. pp. 1802–1819. ACM (2018)
15. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: EuroS&P. pp. 451–466. IEEE (2017)
16. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: CSF. pp. 164–178. IEEE Computer Society (2016)
17. Cox, J.: How police secretly took over a global phone network for organized crime. Motherboard Tech by VICE, July 2 (2020), https://www.vice.com/en/article/3aza95/how-police-took-over-encrochat-hacked. Accessed 2020-10-13
18. Cremers, C., Fairoze, J., Kiesl, B., Naska, A.: Clone detection in secure messaging: Improving post-compromise security in practice. In: CCS. pp. 1481–1495. ACM (2020)
19. Feiner, L.: Republican senators introduce bill that tech advocates have warned would weaken privacy. CNBC, June 24 (2020), https://www.cnbc.com/2020/06/24/gop-senators-introduce-bill-that-would-create-a-backdoor-for-encryption.html. Accessed 2021-02-22
20. Krawczyk, H., Eronen, P.: Hmac-based extract-and-expand key derivation function (HKDF). RFC **5869**, 1–14 (2010)
21. Langley, A., Hamburg, M., Turner, S.: Elliptic curves for security. RFC **7748**, 1–22 (2016)
22. Lund, J.: Technology Preview for secure value recovery. https://signal.org/blog/secure-value-recovery/ (2019), accessed 2021-02-15

23. Meyer, D.: Russia's online censorship machine is no longer running smoothly. FORTUNE, June 24 (2020), https://fortune.com/2020/06/24/russia-online-censorship-faltering-telegram-kasparov/. Accessed 2021-02-22
24. Microsoft: Skype private conversation. https://az705183.vo.msecnd.net/onlinesupportmedia/onlinesupport/media/skype/documents/skype-private-conversation-white-paper.pdf (2018), accessed 2020-09-28
25. Open Whisper Systems: Advanced cryptographic ratcheting. https://signal.org/blog/advanced-ratcheting/, accessed 2021-02-16
26. Open Whisper Systems: Signal Protocol Specifications. https://signal.org/docs/, accessed 2020-09-28
27. Open Whisper Systems: The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, accessed 2020-09-28
28. Open Whisper Systems: The Sesame Algorithm: Session Management for Asynchronous Message Encryption. https://signal.org/docs/specifications/sesame/, accessed 2020-09-28
29. Open Whisper Systems: The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/, accessed 2020-09-28
30. Randall: Introducing Signal PINs. https://signal.org/blog/signal-pins/ (2020), accessed 2021-02-15
31. Rogaway, P.: Authenticated-encryption with associated-data. In: ACM Conference on Computer and Communications Security. pp. 98–107. ACM (2002)
32. Rogaway, P., Shrimpton, T.: A provable-security treatment of the key-wrap problem. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 4004, pp. 373–390. Springer (2006)
33. WhatsApp: Whatsapp encryption overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf (2017), accessed 2020-09-28