Breaking Bridgefy, again
Adopting libsignal is not enough

Raphael Eikenberg[1], Martin R. Albrecht[2], and Kenneth G. Paterson[1]

[1] Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zurich
reikenberg@ethz.ch, kenny.paterson@inf.ethz.ch
[2] Information Security Group
Department of Information Security
Royal Holloway, University of London
martin.albrecht@royalholloway.ac.uk

Abstract Bridgefy is a messaging application that uses Bluetooth-based mesh networking. It has been advertised by its developers and others for use in areas witnessing large-scale protests and often violent confrontations between protesters and agents of the state. After a security analysis in August 2020 reported severe vulnerabilities that invalidated Bridgefy's claims of confidentiality, authentication and resilience, the Bridgefy developers adopted the Signal protocol. The Bridgefy developers then continued to advertise their application as being suitable for use by higher-risk users.
In this work, we analyse the revised security architecture of Bridgefy and report severe vulnerabilities. We show that an adversary can compromise the confidentiality of one-to-one messages by using a TOCTOU attack side-stepping Signal's guarantees. We also show that an adversary can impersonate other users in the broadcast channel. Finally, we show that any nodes in the network that receive a single carefully crafted message become unable to participate in further network communication.
Overall, we find that the fixes deployed in response to the August 2020 analysis fail to remedy the previously reported vulnerabilities.

1 Introduction

Bridgefy is a mobile application and software development kit (SDK) that provides communication capabilities over Bluetooth. It allows users to build a mesh network to exchange one-to-one and broadcast messages. Its primary target application areas are large events such as sports events where existing Internet infrastructure may not be able to cope with demand. However, its developers also actively promote their application for use in protests and other situations of social unrest, where mobile telecommunications and Internet connections may be unreliable [5, 13, 18, 21, 29–35]. According to Bridgefy, their mobile application has been downloaded by more than 6 million people [36], while they report their SDK is used by more than 40 companies [4].

In August 2020, Albrecht, Blasco, Jensen, and Mareková [1] performed a security analysis of the Bridgefy application and reported severe vulnerabilities: Bridgefy permitted its users to be tracked, offered no authenticity, no effective confidentiality protections and lacked resilience against adversarially crafted messages. Thus, if protesters relied on Bridgefy, an adversary was able to produce social graphs about them, read their messages, impersonate anyone to anyone and shut down the entire network with a single maliciously crafted message.

In response, in October 2020, Bridgefy announced [4] an overhaul of their security architecture with the high-level changes being given as:

> – *All messages will be end-to-end encrypted*
> – *A third person will no longer be able to impersonate any other user*
> – *Man-in-the-middle attacks done by modifying stored keys will no longer be possible*
> – *One-to-one messages sent over the mesh network will no longer contain the sender and receiver IDs in plain text*
> – *A third person will no longer be able to use the server's API to learn others' usernames*
> – *All payloads will be encrypted*
> – *Historical proximity tracking will not be possible*
>
> *This basically means that all messages and users are now safe from unwanted prying eyes. [...] We are aware of the tremendous responsibility we have towards our users, and we're committed to improving our security continuously to make sure the chances of attacks are reduced even further.*

The key technical change implemented by Bridgefy is the adoption of the Signal protocol [12]. In addition, all traffic (direct, one-on-one mesh based, broadcast) is now additionally encrypted with a network-wide symmetric key in AES-ECB mode. Since then, no public independent security assessment of the Bridgefy application has been performed, but Bridgefy started advertising their application again for higher-risk scenarios [6].

1.1 Contributions

In this work, we report severe, practically exploitable vulnerabilities in the Bridgefy application. All but one attacks focus on the setting where the shared AES-ECB key is known to the adversary. This assumption is well justified for the Bridgefy application being advertised for use in protest settings, since in this case the network-wide encryption key can be retrieved by an adversary using dynamic instrumentation. In particular, we make the following contributions:

In Section 3, we give an overview of the inner workings of Bridgefy in version 3.1.3, providing an outline of the architecture and the Bridgefy protocol.

In Section 4, we present a new attack that breaks the confidentiality of Bridgefy private chats. This attack exploits a difference in time that arises between the recipient's key being checked and being used. Thus the attack is of the 'time of check, time of use' (TOCTOU) variety. It works by associating the attacker's public-key with the session between the target and a sender. We report that many vulnerabilities reported in [1] remain unfixed or insufficiently fixed. Specifically, we show that (i) Bridgefy users can still be tracked. (ii) Broadcast messages remain unauthenticated; an attacker can exploit this to mount impersonation attacks. (iii) The protocol remains susceptible to an attacker in the middle. While such an attack is now limited to the first exchange between a pair of users (i.e. it abuses a 'trust on first use' or TOFU assumption), we note that Bridgefy offers users no option to verify the public keys of their contacts. (iv) The previous Denial of Service (DoS) attack remains applicable, albeit in a limited form.

In Section 5, we evaluate our results and suggest possible remediation techniques for unfixed vulnerabilities. We, however, strongly recommend against relying on Bridgefy in any security critical setting until a comprehensive third-party audit has been completed and the company commits to performing regular such audits.

1.2 Disclosure

We notified the Bridgefy developers about our findings on 2021-05-21. They confirmed receipt a few days later and described their plan to remediate the vulnerabilities. On 2021-07-21, Bridgefy informed us they were not going to publicly disclose the problems we reported, explaining they feared to put their users' safety at risk if they did. However, they promised to remove the term 'end-to-end' from all of their social media and blog publications. Our most critical finding—the TOCTOU vulnerability—was still exploitable until 2021-08-12 as originally described, but patched with version 3.1.7.

1.3 Related Work

Time-of-Check to Time-of-Use (TOCTOU) TOCTOU vulnerabilities exploit a change in state between when a certain property is checked and when it is used [28]. Bishop and Dilger [3] were among the first to describe this class of vulnerabilities and studied them in the context of file systems.

Signal The Signal protocol can be used to provide end-to-end encrypted communication. It was subjected to extensive study by Cohn-Gordon, Cremers, Dowling, Garratt, and Stebila [9], who analysed the key agreement as well as the ratcheting mechanism of Signal. Their analysis revealed no significant flaws in the design of the protocol.

Bluetooth Low Energy (BLE) BLE is a widely adopted wireless technology used in mobile and Internet of Things (IoT) devices. Ryan [22] conducted an early analysis of BLE security, demonstrating packet injection and breaking the key exchange as part of the encryption. Sivakumaran and Blasco [25] showed that pairing protected BLE data needs to be secured on the application layer in Android to prevent co-located applications on the device from accessing it. Wu, Nan, Kumar, Tian, Bianchi, Payer, and Xu [37] found a weakness in the BLE specification that enabled an attacker to impersonate a device to another. Zhang, Weng, Dey, Jin, Lin, and Fu [38] reported practically exploitable downgrade attacks on BLE.

1.4 gzip

gzip [10] is a file format for lossless compressed data. It makes use of the DEFLATE compression algorithm, which is based on LZ77 [39] and Huffman [16] coding. Overall, the algorithm first replaces any repeating block of data with a reference to a previous occurrence. Then, broadly speaking, it ranks bytes and references by occurrence and assigns them a Huffman symbol accordingly.

2 Preliminaries

The Bridgefy application allows peers in close proximity to exchange messages over Bluetooth by spanning a mesh network. It supports Bluetooth Low Energy (BLE) as well as Classic Bluetooth, with BLE being the default mode of operation. Under certain conditions messages can also be transmitted over the Internet, however, if a device is offline, it will communicate over Bluetooth exclusively. In this work we focus exclusively on BLE-based communication.

There is a broadcast room, where a peer can announce messages to all other peers as a group. Peers can also send direct messages privately to other peers, but only if their devices have previously been in direct reach to exchange cryptographic keys.

In the background, the Bridgefy app makes use of Bridgefy's SDK. It is the SDK that provides the necessary mechanics to (i) establish trust between devices, (ii) encrypt and decrypt packets, and (iii) transmit packets via the Bluetooth functionality offered by the underlying operating system.

Peers in Bridgefy are identified by a universally unique identifier (UUID) of 128 bit called *userId*. This UUID is randomly generated on each device when the app is launched for the first time. Users must also pick a *display name* when they install the app, however, it is not unique and can be arbitrarily chosen.

2.1 Methodology

The details of Bridgefy's inner working are not public. Since both the Bridgefy application and the SDK are closed-source software, we needed to reverse engineer them.

We analysed the Bridgefy Android application in version 3.1.3 and the SDK in version 2.0.2 dated 2021-04-27 and 2021-02-09 respectively. The APK file was downloaded from Google Play to an Android phone and fetched from there directly via ADB without the involvement of a third party. The SDK came packaged as an AAR file and was available publicly on the Internet.

We approached the assessment in two steps: a *static* analysis revealed a rough overview of the mechanics in both software components, while a *dynamic* analysis to confirm our observations.

Static Analysis. We decompiled Bridgefy to reconstruct Java source code from the official artefacts as follows.

The APK file was directly decompiled using Jadx, but also converted into a JAR file using enjarify for further processing [14, 26]. The SDK's AAR file was extracted to retrieve a JAR file. Both JAR files were then decompiled to Java source, leveraging multiple Java decompilers with different strengths: CFR [2], Fernflower [17], Krakatau [15], and Procyon [27]. While the output was obfuscated, Bridgefy's code sometimes references class and method names similar to the statement in Fig. 1. This helped us make sense of more complex blocks of code.

```java
Log.e("ClassName", "funcName() failed: " + e.getMessage(), e);
```

Figure 1: Some statements in Bridgefy reveal the intended names of classes and methods.

Dynamic Analysis. After a close manual inspection of the generated Java source code, we instrumented a running instance of the Bridgefy app via Frida [20] and objection [23]. This allowed us to hook into existing functions of the app, and thereby monitor method calls and change method behaviour. In particular, we could observe packets as they were being encrypted and decrypted.

2.2 libsignal

Bridgefy makes use of Signal's official Java library, where endpoints are identified by a `SignalProtocolAddress` [24]. This type is merely a combination of a `name` that identifies the user and a `deviceId` that is unique for each device a user owns. However, Bridgefy always sets the `deviceId` to 0, while using a peer's userId in the addresses `name` field.

In Signal, for two parties to communicate, they need to retrieve a pre-key bundle (PKB) from one another. The PKB contains several keys of a party to establish a Signal session with them.

Signal maintains state for all established sessions in a `SignalProocolStore`. When a new PKB is received from a peer, Bridgefy instantiates a `SessionBuilder` which is supplied with the protocol store and the peer's protocol address. A new session is then created by passing the PKB to `SessionBuilder.process()`.

When the SDK needs to encrypt data using Signal for a particular peer, it instantiates a `SessionCipher` which is supplied with the protocol store and the peer's protocol address. The data is then passed to `SessionCipher.encrypt()`.

2.3 Interface of the SDK

To illustrate the Bridgefy application and SDK, we briefly introduce the interworking between these two components. The basic use of the SDK is documented in a GitHub repository together with official sample applications [7]. A description of all exposed functionality is available in the official SDK documentation [8].

As a first step, the app calls `Bridgefy.initialize()` with a registration callback and an API key. The SDK will then validate the API key and notify the app of the result via the callback.

On success, the app next calls `Bridgefy.start()` with two different callbacks:

- a **message listener** that is called when a new message is received, and
- a **state listener** that is called when a connection with a nearby peer is established or closed.

Finally, if the app wants to send a message itself, it calls `Bridgefy.sendMessage()` or `Bridgefy.sendBroadcastMessage()`.

3 Bridgefy Architecture

In this section, we explain how messages are transmitted and how Bridgefy attempts to achieve confidentiality.

3.1 Message Types

Users can decide between sending broadcast and private messages. On the network layer, Bridgefy differentiates two different packet types: those that are routed through the mesh network and those that are sent directly. The former packets are referenced as type `ForwardMessage`, while the latter are of type `BleEntityContent`.

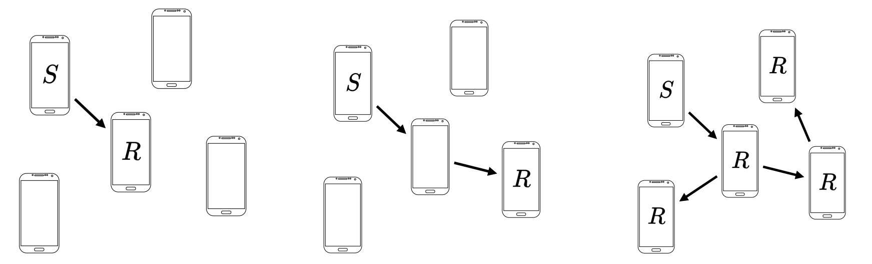
There are three different kinds of messages to consider in Bridgefy:

- A **broadcast message** that is sent from one peer to multiple other peers over the mesh network.
- A private message that is sent from one peer to another over the mesh network. We call this a **multi-hop message**.
- A private message that is sent from one peer to another directly. We call this a **one-to-one message**. Note that this setting is only applicable when the peers are in close proximity.

These message types are illustrated in Fig. 2.

3.2 Handshake

When two devices get physically close enough to establish a Bluetooth connection, they perform a handshake (assuming that they have not performed a handshake



(a) One-to-one message (b) Multi-hop message (c) Broadcast message

Figure 2: We consider three different types of messages. S and R denote sender and receivers respectively.

previously). This process is handled by the SDK, meaning it is not transparent to the app.

In the handshake, each party generates a PKB and sends it to the other party. Based on the exchanged PKBs, a Signal session is established, enabling the parties to encrypt and authenticate messages.

Assume Alice A and Bob B come within range of one another for the first time, then the handshake proceeds as follows:

$$\begin{aligned} A \to B: &\quad \texttt{ResponseTypeGeneral}(\mathrm{userId}_A) && (1)\\ B \to A: &\quad \texttt{ResponseTypeGeneral}(\mathrm{userId}_B) && (2)\\ A \to B: &\quad \texttt{ResponseTypeKey}(\mathrm{PKB}_A) && (3)\\ B \to A: &\quad \texttt{ResponseTypeKey}(\mathrm{PKB}_B) && (4)\end{aligned}$$

Here, userId_A denotes the userId of peer A and PKB_A denotes the PKB generated by A. After (1), B checks if any Signal session has already been established for userId_A, and aborts the handshake if this is the case. Peer A may also abort the handshake after (2).

Note that we have made some simplifications here that are not relevant for our analysis. In reality, the packets contain CRC checksums and version information. Further, all four packets of the handshake are wrapped in a `BleHandshake` packet, which itself is wrapped in a `BleEntity` packet.

Note that the handshake is not performed over the mesh network. As a result, only peers that have previously met can later exchange messages privately over the mesh network.

Also note how the handshake follows the trust on first use (TOFU) principle: in (3) and (4), the parties implicitly trust the PKB they receive. Note that users cannot verify the keys of peers manually, as Bridgefy's user interface offers no way to do so.

3.3 Packet Encoding

On the lowest layer, Bridgefy encapsulates all packets into the type `BleEntity`. Its `et` field (presumably for 'entity type') is used to indicate the type of packet it contains. Table 1 lists the possible packet types.

Name	*Value*	*Type*
`ENTITY_TYPE_HANDSHAKE`	0	`BleHandshake`
`ENTITY_TYPE_MESSAGE`	1	`BleEntityContent`
`ENTITY_TYPE_BINARY`	2	unused
`ENTITY_TYPE_MESH`	3	`ForwardTransaction`
`ENTITY_MESH_REACH`	4	unused
`ENTITY_TYPE_FILE`	5	unused

Table 1: Packet types in Bridgefy. The values 2, 4, and 5 are defined by the SDK, but never used. The name refers to the variable name of the constant.

Multi-hop messages and broadcast messages are represented by the type `ForwardPacket`. Among other things, this packet type features a time to live (TTL) field. This field is decremented by network nodes upon packet forwarding. This prevents packets from circulating in the mesh network indefinitely. For efficiency, multiple objects of type `ForwardPacket` are bundled into a packet of type `ForwardTransaction` on the network layer. Figure 3 illustrates the relations between these types in a UML diagram.

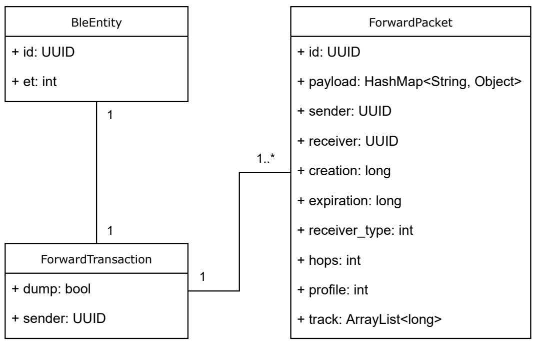


Figure 3: A `ForwardPacket` is always encapsulated in a `ForwardTransaction`, which itself is encapsulated in a `BleEntity`.

The `receiver_type` field is used to differentiate broadcast messages from multi-hop messages; it is set to the values 1 and 0 for broadcast and multi-hop, respectively.

Going forward, we will assume that a `ForwardTransaction` contains only a single `ForwardPacket`. This simplifies the description of serialisation and encryption to make it more comprehensible. It also corresponds to a low-traffic mesh network.

Because one-to-one messages are not carried over the mesh network, they are encoded in the packet type `BleEntityContent`, illustrated in Fig. 4 as a UML diagram.

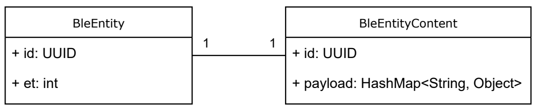


Figure 4: A `BleEntityContent` is also encapsulated in a `BleEntity`.

The message typed by a user is referred to as 'payload content'. In a `ForwardTransaction`, the payload content is stored as a string under the key `ct` in the `payload` map of the `ForwardPacket`. For one-to-one messages, it is encoded in the `payload` map of the `BleEntityContent` respectively.

3.4 Packet Encryption

Before a `BleEntity` is sent to another peer, it is serialised using MessagePack [11], compressed using gzip, and then encrypted. The encryption step can involve Signal encryption in combination with AES in ECB mode with PKCS#7 padding, or AES-ECB with PKCS#7 padding only.

For AES-ECB, a symmetric key is shared between all peers that use the same application. In the case of the Bridgefy application, an adversary can easily obtain this symmetric key, as the application is publicly available. More generally, depending on the nature of the threats considered—inside, outside—the shared symmetric key may be considered known or unknown to the adversary.

Signal encryption is only used for the `payload` field of multi-hop and one-to-one messages. Broadcast messages as well as the metadata of any other packets are encrypted with the shared AES-ECB key. In what follows we will ignore this layer of encryption except when explicitly stated otherwise. This is aligned with our assumption that this key is known to the adversary. Table 2 summarises which encryption method is used for the different packet types.

Remark 1. Previous versions of Bridgefy implemented a custom scheme based on RSA in place of the Signal protocol. With Bridgefy's adoption of the Signal protocol in place of RSA, the padding oracle attack reported in [1] is no longer applicable.

3.5 Devices and Sessions

In the Bridgefy SDK, the `DeviceManager` is responsible for maintaining a list of nearby Bluetooth devices. Each device is associated with a session, which itself is

Data Category	`BleHandshake`	`BleEntityContent`	`ForwardTransaction`
Metadata	AES-ECB	AES-ECB	AES-ECB
Payload	AES-ECB	libsignal	libsignal

Table 2: Encryption of packets in Bridgefy by data category and packet type.

managed by a `SessionManager`. Since the co-existence of devices and sessions appears arbitrary, we will in the following refer to sessions only.

During the handshake, a userId is received from the other peer and saved in the corresponding session.

When the SDK is instructed to send a message to a userId, it checks for the existence of a session that is associated with that userId. The message is then queued in the `TransactionManager` together with the session. Once Android requests for more Bluetooth data to send, the SDK pops the queued message, encrypts it for the userId saved in the session, and dispatches it.

When a Bluetooth packet is received, the SDK looks up the correct session based on the remote Bluetooth address. After assembling and decrypting the packet, it is passed to a generic message handler.

4 Attacks

We present a new attack on Bridgefy that affects the confidentiality of private messages. We assume that the adversary has knowledge of the network-wide symmetric key. We stress once more that this condition is satisfied for the flagship Bridgefy application. Further, as outlined in Section 1.1, several vulnerabilities described in [1] remain unfixed. We discuss these here in more detail.

4.1 Breaking Confidentiality of Private Messages

We identified a time-of-check to time-of-use (TOCTOU) vulnerability [19, pg. 157] in the SDK that can be leveraged to read private messages between two users of the Bridgefy app.

For simplicity, we assume that the communicating parties are not directly connected via Bluetooth. While this assumption is not strictly necessary, it makes the exploitation of this vulnerability easier.

Accompanying the textual description of the attack that follows, the packet flow used in the attack is illustrated in Fig. 5. The numbering on the very left of the illustration matches the numbering in the individual steps in the following paragraphs.

Assume a setting where Alice and Bob's devices have already performed a handshake and have exchanged messages (e.g. M_0 in Fig. 5). Bob's device then goes out of range of Alice's so that the Bluetooth connection is terminated (step 1 in Fig. 5). If Alice's device was to now send a message to Bob's device, it would send it into the mesh network, as Bob's device is now not a directly connected peer.

Next, Mallory performs a full handshake with Alice's device so that Alice's device registers Mallory's PKB (step 2 in Fig. 5). Until this point, Mallory behaves normally as any honest peer would.

Mallory again sends the first packet of the handshake, this time using Bob's userId in place of Mallory's own (step 3 in Fig. 5). No mechanism in Bridgefy prevents Mallory's message from being accepted. Alice's device will now associate the established session with Bob. In particular, Alice's device will queue any subsequent packets intended for Bob in this session.

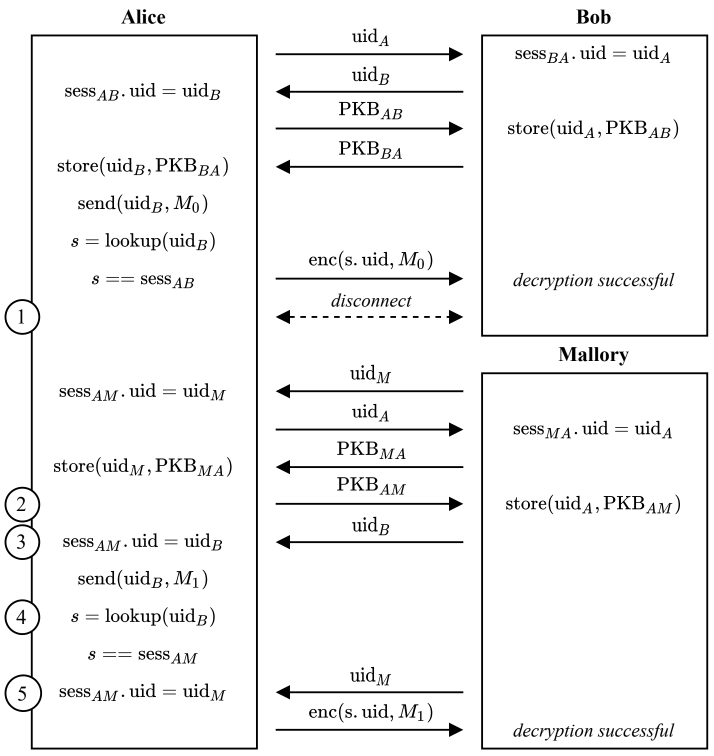
Because Mallory initiated a new handshake using Bob's userId, Alice's device will indicate to Alice that Bob's device is in range. Suppose then Alice types a message intended for Bob (M_1 in Fig. 5). The SDK looks for any active session where the userId equals that of Bob's device as per our description in Section 3.5 (step 4 in Fig. 5). Since Mallory provided the userId of Bob's device in its second handshake, Alice's session with Mallory yields a match. Hence, the message is queued in the `TransactionManager` for the session with Mallory. If the packet was dispatched at *this* point, the packet would be encrypted with Signal for Bob (this is because libsignal also uses the userId to decide which key to use in the encryption). So Mallory would not be able to read it. However, instead of being dispatched, the packet is typically queued.

Now, Mallory sends the first packet of the handshake for a third time, using Mallory's own userId (step 5 in Fig. 5). The userId of the session from the perspective of Alice's device now equals that of Mallory again. When the SDK on Alice's phone is asked for more data to transmit via Bluetooth, the packet is encrypted by Signal for Mallory and dispatched. (Again, this is because libsignal uses the userId to decide which key to use in the encryption.)

The above attack exploits a race condition: because Mallory sends the userId of Bob's device in its second handshake, Alice thinks she has a session with Bob and so types a message to be sent to Bob. This message is then queued. But Mallory switches the userId back to its own userId in the third handshake, so that when the message is dequeued and the libsignal encryption is actually performed, it is done using Mallory's public key.

Remark 2. If no proper Signal session was established in the beginning, switching back to Mallory's real userId would require a full 2-round-trip handshake. Given that this attack exploits a race condition, it is important for Mallory to initiate an honest handshake before proceeding with the attack.

We implemented a proof of concept (PoC) for this attack to confirm that it works. We sent 100 messages from Alice's phone, 56 of which were received by Mallory in our tests. The reason why Mallory does not receive all messages is that the attack is based on a race condition. What plays into the hands of Mallory is that Bridgefy reschedules a private message if it cannot be delivered to the receiver. If the SDK looks up a session matching the receiver's userId while the session is associated with Mallory's userId, it will be rescheduled. Still,



Alice
Bob
Mallory
uid_A
$\mathrm{sess}_{BA}.\mathrm{uid} = \mathrm{uid}_A$
$\mathrm{sess}_{AB}.\mathrm{uid} = \mathrm{uid}_B$
uid_B
PKB_{AB}
$\mathrm{store}(\mathrm{uid}_A, \mathrm{PKB}_{AB})$
PKB_{BA}
$\mathrm{store}(\mathrm{uid}_B, \mathrm{PKB}_{BA})$
$\mathrm{send}(\mathrm{uid}_B, M_0)$
$s = \mathrm{lookup}(\mathrm{uid}_B)$
$s == \mathrm{sess}_{AB}$
$\mathrm{enc}(s.\mathrm{uid}, M_0)$
decryption successful
disconnect
1
$\mathrm{sess}_{AM}.\mathrm{uid} = \mathrm{uid}_M$
uid_M
uid_A
$\mathrm{sess}_{MA}.\mathrm{uid} = \mathrm{uid}_A$
$\mathrm{store}(\mathrm{uid}_M, \mathrm{PKB}_{MA})$
PKB_{MA}
PKB_{AM}
$\mathrm{store}(\mathrm{uid}_A, \mathrm{PKB}_{AM})$
2
3
$\mathrm{sess}_{AM}.\mathrm{uid} = \mathrm{uid}_B$
uid_B
$\mathrm{send}(\mathrm{uid}_B, M_1)$
4
$s = \mathrm{lookup}(\mathrm{uid}_B)$
$s == \mathrm{sess}_{AM}$
5
$\mathrm{sess}_{AM}.\mathrm{uid} = \mathrm{uid}_M$
uid_M
$\mathrm{enc}(s.\mathrm{uid}, M_1)$
decryption successful

Figure 5: The packet flow of our TOCTOU attack on Bridgefy. Alice tries to send a message to Bob twice. The first message M_0 is sent to Bob normally. However, Mallory can decrypt the second message M_1.

packets can get 'lost' for Mallory when the packet is encrypted right after Mallory switches the userId back to Bob's.

Note that when Mallory intercepts a message, Bob will not receive it. This is because Alice encrypts the packet for Mallory only, while Mallory cannot re-encrypt it for Bob in Alice's name. If Mallory was to encrypt and send a message to Bob while using Alice's userId during the handshake, Bob would fail to decrypt the packet. Instead, if Mallory used their real userId, Bob would process the packet before Mallory gets the chance to change the userId of the session again. In other words, the attack breaks confidentiality but not authentication.

4.2 Active Attacker-in-the-middle (MITM)

Due to Bridgefy's architecture, any PKB received from a new peer is inherently trusted, following the TOFU principle. This implies that Bridgefy is vulnerable to a MITM attack similar to the one reported in [1]. However, with the adoption of libsignal, the conditions necessary to perform the attack have changed slightly:

Mallory now needs to perform the handshake with Bob before Alice does, whereas in earlier versions of Bridgefy this was not required.

The updated attack proceeds as follows: Assume that Alice and Bob have not met before. Mallory performs a handshake with each of Alice and Bob and impersonates them to one another. Any message then sent from one party is then relayed by Mallory to the other party.

If Mallory tries to perform the attack after Bob has already run a handshake with Alice, the following would happen: Mallory would try to impersonate Alice by performing a full handshake with Bob, using Alice's userId but Mallory's own PKB. When the SDK tries to store Mallory's PKB under Alice's userId, libsignal would throw an exception since Alice has already established a Signal session with Bob and so a PKB is already present under Alice's userId.

Note that Alice and Bob will never be able to confirm if they are directly exchanging messages or if they are instead subject to an MITM attack. This is because, in contrast to popular messaging applications like Signal, Bridgefy does not provide any mechanism to allow users to manually verify the keys of other peers.

4.3 Impersonation in the Broadcast Channel

An adversary can forge arbitrary broadcast messages. The adversary can send messages under the name of any userId and freely choose a payload content and display name. This is because broadcast messages are not authenticated at all.

We implemented a PoC of this attack to verify it. We found, however, while Mallory can leverage this vulnerability to send a message with Alice's userId, they cannot do so when also choosing a different display name. Any peer that has received a legitimate message from Alice before will remember her original display name and associate it with her userId. Hence, when Mallory supplies a new display name together with Alice's userId, such peers will still show the old display name for the new message.

4.4 Denial of Service (DoS)

We confirmed that Bridgefy remains vulnerable to a ZIP bomb attack as reported in [1]. This attack exploits that all packets are decompressed using gzip after decryption. Thus, an adversary can inject a specifically crafted packet that decompresses to more bytes than are available in the memory of a target. The target's app will first freeze and become unresponsive, and eventually crash. This allows Mallory to prevent specific devices from participating in the mesh network.

With its overhaul, Bridgefy now encrypts all metadata with the shared key, making it necessary for a peer to decompress a packet before being able to determine what type of message was received. Hence, the attack reported in [1] where only a single message can shut down the entire network no longer works, as it requires peers to forward mesh packets before decompression.

However, this vulnerability can still be used to interfere with the correct functioning of the mesh network by shutting down several parts of the network.

Specifically, all the peers that are one hop from the adversarially controlled peers can be taken offline. Given that resilience is a key requirement for Bridgefy's adoption in higher-risk environments, this attack invalidates one of Bridgefy's most central features.

4.5 Building a Social Graph

As reported in [1], Bridgefy used to transmit the `sender` and `receiver` fields of one-to-one and multi-hop messages in plaintext. These are now encrypted under the shared AES-ECB key. Thus, an adversary in the mesh network spanned by the Bridgefy application remains able to learn who is privately communicating with whom.

Furthermore, the `track` field of a `ForwardPacket` lists nodes that have been involved in the delivery of a packet at the time of capture, which permits building a model of the psychical topology of the mesh network. An adversary can also use this feature to trace back the location of a peer that repeatedly sends messages.

4.6 Historical Proximity Tracing

Bridgefy announced that they now protect against the historical proximity tracing method reported in [1]. However, our tests show that the attack is still possible: a full handshake is performed when two devices have not been in close proximity to each other before, while only a partial handshake is performed otherwise.

An adversary can leverage this, e.g. to find out if a peer was physically present at a protest. Given that the timing and the approximate size of the handshake packets are known to the adversary, the attack is even possible without knowledge of the shared symmetric key.

5 Discussion

We demonstrated that an adversary can read private messages by exploiting the TOCTOU vulnerability. Given that anyone has access to the app and hence also to the shared symmetric key, the most restricting requirement is physical presence: an adversary must be in close enough proximity with the target to establish a direct Bluetooth connection. In the use-case considered here, a protest, this does not pose a major obstacle: the adversary simply pretends to be one of the protesters.

Furthermore, during the attack, the app will indicate to Alice that Bob is around, even if he is not. This is because Mallory associates Bob's userId with the session on Alice's device. The indicator could suggest that the session is more secure, making Alice feel more secure in her conversation.

The easiest mitigation for the overall issue would be to encrypt a packet already when it is queued in the `TransactionManager`. Alternatively, Bridgefy could implement a proper state machine in their `Session` class to allow only for a single handshake to happen. This would block an attacker from switching to another userId after the first round trip of the handshake.

Furthermore, we demonstrated that an adversary can impersonate any user in the broadcast channel. During a protest, the adversary could simply impersonate the protest leaders to announce a new tactic.

As a mitigation strategy, Bridgefy could employ cryptographic signatures and enforce uniqueness of usernames. Since users need to register their device online already, the application could generate a key pair and send the public key to the Bridgefy server to retrieve a certificate. The certificate would bind the key pair to the reserved username. Each message would then be signed with the private key, such that the username is protected against impersonation. Ideally, the public key of the server is pinned in this setting. This would not prevent replay attacks without further mechanisms being introduced.

The privacy issues of Bridgefy remain largely unresolved. While sender and receiver identities are no longer transmitted in plaintext, any peer with access to the application can decrypt the relevant fields. An adversary can leverage this to build communication graphs and thereby identify protest leaders. Moreover, other metadata in Bridgefy packets can be used to locate a peer among a crowd of protesters.

The Bridgefy developers are aware that their application is used by protesters in highly volatile and dangerous situations. Indeed, they continue to promote their application as being suitable for this setting. After having been informed by previous work about serious security vulnerabilities, Bridgefy adopted the Signal protocol and announced a security overhaul of their application. Yet, our findings establish that the Bridgefy application fails again to meet the basic security requirements for these highly adversarial environments.

5.1 Acknowledgements

This research was supported by the EPSRC grants EP/S020330/1, EP/P009417/1, EP/L018543/1.

Bibliography

[1] Albrecht, M.R., Blasco, J., Jensen, R.B., Mareková, L.: Mesh messaging in large-scale protests: Breaking Bridgefy. In: Paterson, K.G. (ed.) Topics in Cryptology - CT-RSA 2021, Lecture Notes in Computer Science, vol. 12704, pp. 375–398, Springer (2021), https://doi.org/10.1007/978-3-030-75539-3_16

[2] Benfield, L.: https://www.benf.org/other/cfr/ (no date)

[3] Bishop, M., Dilger, M.: Checking for race conditions in file accesses **9**(2), 131–152 (Mar 1996), ISSN 0895-6340

[4] Bridgefy: Press release – major security updates at bridgefy! https://bridgefy.me/press-release-major-security-updates-at-bridgefy/ (Oct 2020)

[5] Bridgefy: https://twitter.com/bridgefy/status/1359200080700600322 (Feb 2021), https://web.archive.org/web/20210209175856/https://twitter.com/bridgefy/status/1359200080700600322

[6] Bridgefy: https://twitter.com/bridgefy/status/1356603238674538496 (Feb 2021), https://web.archive.org/web/20210514094051/https://twitter.com/bridgefy/status/1356603238674538496

[7] Bridgefy: https://github.com/bridgefy/bridgefy-android-sdk-sample (no date)

[8] Bridgefy: https://www.bridgefy.me/docs/javadoc/ (no date)

[9] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013 (2016), https://eprint.iacr.org/2016/1013

[10] Deutsch, L.P.: GZIP file format specification version 4.3. RFC 1952 (May 1996), https://doi.org/10.17487/RFC1952, URL https://rfc-editor.org/rfc/rfc1952.txt

[11] Furuhashi, S.: https://msgpack.org/ (no date)

[12] GitHub – Bridgefy: https://github.com/bridgefy/bridgefy-android-sdk-sample/blob/56ad2acc7c8893cb2ba53f0aa5839b867ebea446/CHANGELOG.md (no date)

[13] Goodin, D.: Bridgefy, the messenger promoted for mass protests, is a privacy disaster. Ars Technica, https://arstechnica.com/features/2020/08/bridgefy (Aug 2020)

[14] Grosse, R.: https://github.com/Storyyeller/enjarify (no date)

[15] Grosse, R.: https://github.com/Storyyeller/Krakatau (no date)

[16] Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the IRE **40**(9), 1098–1101 (1952), https://doi.org/10.1109/JRPROC.1952.273898

[17] JetBrains: https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine (no date)

[18] Koetsier, J.: Hong Kong protestors using mesh messaging app China can't block: Usage up 3685%. https://web.archive.org/web/20200411154603/https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/ (Sep 2019)

[19] van Oorschot, P.C.: Computer Security and the Internet: Tools and Jewels. Springer, Cham (2020)

[20] Ravnås, O.A.V.: https://frida.re/ (no date)

[21] Reuters: Offline message app downloaded over million times after myanmar coup. https://www.reuters.com/article/amp/idUSKBN2A22H0 (2021)

[22] Ryan, M.: Bluetooth: With low energy comes low security. In: 7th USENIX Workshop on Offensive Technologies (WOOT 13), USENIX Association, Washington, D.C. (Aug 2013), URL https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan

[23] SensePost: https://github.com/sensepost/objection (no date)

[24] Signal: https://github.com/signalapp/libsignal-protocol-java (no date)

[25] Sivakumaran, P., Blasco, J.: A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape. In: Heninger, N., Traynor, P. (eds.) USENIX Security 2019: 28th USENIX Security Symposium, pp. 1–18, USENIX Association (Aug 2019)

[26] skylot: https://github.com/skylot/jadx (no date)

[27] Strobel, M.: https://github.com/mstrobel/procyon (no date)

[28] The MITRE Corporation: https://cwe.mitre.org/data/definitions/367.html (Jul 2021)

[29] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1197191632665415686 (Nov 2019), http://archive.today/aNKQy

[30] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1216473058753597453 (Jan 2020), http://archive.today/x1gG4

[31] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1268905414248153089 (Jun 2020), http://archive.today/odSbW

[32] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1268015807252004864 (Jun 2020), http://archive.today/uKNRm

[33] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1356750830955884552 (Feb 2021), https://web.archive.org/web/20210516231628/https://twitter.com/bridgefy/status/1356750830955884552

[34] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1356680753338318859 (Feb 2021), https://web.archive.org/web/20210516231655/https://twitter.com/bridgefy/status/1356680753338318859

[35] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1371507779299590144 (Mar 2021), https://web.archive.org/web/20210514094031/https://twitter.com/bridgefy/status/1371507779299590144

[36] Twitter – Bridgefy: https://twitter.com/bridgefy/status/1417129132169830410 (Jul 2021), https://web.archive.org/web/20210731103342/https://twitter.com/bridgefy/status/1417129132169830410

[37] Wu, J., Nan, Y., Kumar, V., Tian, D.J., Bianchi, A., Payer, M., Xu, D.: BLESA: Spoofing attacks against reconnections in bluetooth low energy. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Association (Aug 2020), URL https://www.usenix.org/conference/woot20/presentation/wu

[38] Zhang, Y., Weng, J., Dey, R., Jin, Y., Lin, Z., Fu, X.: Breaking secure pairing of bluetooth low energy using downgrade attacks. In: Capkun, S., Roesner, F. (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 37–54, USENIX Association (Aug 2020)

[39] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory **23**(3), 337–343 (1977), https://doi.org/10.1109/TIT.1977.1055714

All links were last checked on 2021-05-17.