Security Assessment of Delta Chat's Primary Rust Libraries on behalf of the Open Technology Fund

TABLE OF CONTENTS

TABLE OF CONTENTS 2

EXECUTIVE SUMMARY 4

Scope and Methodology 4

Assessment Objectives 4

Findings Overview 4

Next Steps 4

ASSESSMENT RESULTS 5

INTRODUCTION 6

CRITICAL-RISK FINDINGS 8

HIGH-RISK FINDINGS 8

MEDIUM-RISK FINDINGS 8

M1: Cleartext Transmission of Security Relevant Information 8

M2: Messages, Encryption Keys, and Email Password Stored in Cleartext 12

M3: DeltaChat Message Export/Backups Are Unencrypted 15

M4: Server-Side Request Forgery (SSRF) 17

M5: Social Engineering Attack via Group Messaging UI 20

M6: File Transfers Do Not Validate Filetypes 22

LOW-RISK FINDINGS 25

L1: Confidential Information in Logs 25

L2: Content Security Policy Allows Unsafe-inline 26

L3: Potential Denial of Service via Large File Transfer or Large Messages 28

L4: Cryptographically Deprecated SHA1 Hashing Algorithm in Use 29

L5: Application Build Does Not Employ Position-Independent Executable (PIE) Flag, RELRO, and Stack Cookie Protections 30

L6: Unsafe Dereference Used 31

L7: Homograph Attacks Possible in Various Parts of DeltaChat 32

L8: UI Alert Does Not Convey Potential for Confidentiality Disruption 35

INFORMATIONAL FINDINGS 38

I1: Potential Filesystem Path Traversal Sequence Which Would Allow Arbitrary File Write 38

I2: Additional Security Considerations (Quantum Computing, Traffic Flow Confidentiality, and DeltaChat Attack Surface Reduction) 41

EXECUTIVE SUMMARY

Scope and Methodology

IncludeSec performed a security assessment of Delta Chat's Primary Rust Libraries on behalf of the Open Technology Fund. The assessment team performed a 12 day effort spanning from August 6th – August 21st, 2020, using a Standard Grey Box Assessment Methodology which included a detailed review of all the components described above in a manner consistent with the original Statement of Work (SOW).

Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. IncludeSec also provided remediation steps which Delta Chat could implement to secure its applications and systems.

Findings Overview

IncludeSec identified 16 categories of findings. There were 0 deemed a "Critical-Risk," 0 deemed a "High-Risk," 6 deemed a "Medium-Risk," and 8 deemed a "Low-Risk," which pose some tangible security risk. Additionally, 2 "Informational" level findings were identified that do not immediately pose a security risk.

IncludeSec encourages Delta Chat to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

Next Steps

IncludeSec advises Delta Chat to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by Delta Chat as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist Delta Chat in improving their SDLC in future engagements by providing security assessments of additional products.

ASSESSMENT RESULTS

At the conclusion of the assessment, Include Security categorized findings into four levels of perceived security risk: critical, high, medium, or low. Any informational findings for which the assessment team perceived no direct security risk, were also reported in the spirit of full disclosure. The risk categorizations below are guidelines that IncludeSec believes reflect best practices in the security industry and may differ from internal perceived risk. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. All findings are described in detail within the final report provided to Delta Chat.

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration issues, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application.

The findings below are listed by a risk rated short name (e.g., C1, H2, M3, L4, I5) and finding title. Each finding includes: Description (including proof of concept screenshots and lines of code), Recommended Remediation, and References.

INTRODUCTION

Project Scoping

On July 13, 2020, the assessment team began analyzing the Delta Chat application for security vulnerabilities (version 1.39.0). The assessment team focused on the DeltaChat core Rust software and utilized the Android DeltaChat and Desktop DeltaChat applications for testing. The Electron code of the DeltaChat and the mobile-specific code from the Android or IOS application were not included in this assessment but a limited amount of dynamic testing was performed while trying to identify vulnerabilities in the DeltaChat Rust code and the vulnerabilities identified have been documented. The source code repositories in-scope in order of priority were:

- deltachat-core-rust
- async-smtp
- async-imap
- async-native-tls

Threat Modeling

The following areas were of key focus during the assessment:

- Correctness – Assessing if the implementation follows its defined specification.
- Backdoors – Assessing if the implementations voluntary or involuntary contain backdoors. Examples include weak parameters, oracles that can be leveraged to obtain keys or plaintext and use of non-cryptographically secure pseudo-random number generators.
- Supply Chain Attacks – Assessing if the implementation uses known vulnerable components.
- Ease of Secure Use – Assessing if a user could make mistakes while utilizing the tool that would allow an attacker to take advantage of them.
- Secrets Management – Assessing how sensitive values are handled.
- Source Code Vulnerabilities – Vulnerabilities that could allow an attacker to abuse to extract sensitive information or gain remote code execution on the DeltaChat application environment.

Testing Methodology

As RSA, OpenPGP, and Autocrypt have well-defined specifications, prior research regarding known vulnerabilities was investigated. Dynamic testing and manual source code review were performed to identify vulnerabilities. The **cargo-fuzz** and siderophile framework were used sparingly to identify run-time vulnerabilities but because of time restrictions, additional testing and time would be necessary to fully leverage this testing strategy. Appropriate proofs-of-concept were developed to verify discovered findings. Please also note that the level of depth of attacks was limited by the time-boxed nature of the assessment (11 total workdays).

Suggested Future Areas of Investigation

The assessment team recommends performing additional testing coverage on the following areas:

- Desktop DeltaChat Electron source code
- DeltaChat Android and IOS specific code repositories

References

PKCS #1: RSA Cryptography Specifications Version 2.2
RFC 4880 – OpenPGP Message Format
Twenty Years of Attacks on the RSA Cryptosystem
Autocrypt
DeltaChat

CRITICAL-RISK FINDINGS

No “Critical-Risk” findings were identified during the course of the engagement.

HIGH-RISK FINDINGS

No “High-Risk” findings were identified during the course of the engagement.

MEDIUM-RISK FINDINGS

M1: Cleartext Transmission of Security Relevant Information

Description:

The **DeltaChat** application sends potentially confidential information over a cleartext channel, including initial chat messages, registration HTTP requests, and email credentials. Cleartext credentials are only sent if the client is misconfigured (i.e. IMAP/SMTP encryption is turned off). However, if misconfigured, information traveling this way is susceptible to a man-in-the-middle attack, in which the data is intercepted by an attacker situated anywhere along the network path between the user and the target server. The attacker could be on the local area network, the corporate network, within the ISP, etc. If an attacker were to intercept and modify information submitted by the user or returned by the application, then an attacker could potentially read out initial messages, prevent an encrypted transport channel from being established, steal email credentials and modify registration requests. While the application warns the user that the initial messages could not be encrypted there could be improvements to notify the user when key material has been exchanged. For example, the send (arrow) button currently does not indicate whether a message will be sent encrypted or not but could be upgraded to show a lock symbol after key material has been successfully exchanged as improved security-focused UI/UX.

While the assessment team understands that in some circumstances unencrypted mediums are design decisions, as a defense-in-depth approach, the assessment team recommends removing unencrypted features (e.g. HTTP QR codes, SMTP login, etc.) and providing additional notifications around the initial establishment of PGP key material. The DeltaChat application also supports disabling certificate verification for IMAP/SMTP. Removing support for this feature would also prevent users from accidentally misconfiguring their client.

Affected Locations

- **src/qr.rs**
- **src/login_param.rs**

Steps to Reproduce (QR HTTP Registration)

1. Go to https://www.the-qrcode-generator.com/.

2. Enter **DCACCOUNT:**http://includesecurity.com.
3. Right-click and save the image.
4. Start **Wireshark** and capture on the network interface.
5. Deauthenticate if already authenticated and click the top right settings button and then **Switch Account**.
6. Click **Scan QR Code** on bottom of Delta Welcome Chat Page.
7. Open created QR file.
8. Notice that an HTTP request is sent unencrypted to http://includesecurity.com.

Steps to Reproduce (Initial Unencrypted Messages)

1. Create an email account and authenticate via DeltaChat application
2. Scan a QR code to add a DeltaChat user.
3. Send some messages to the DeltaChat user.
4. Authenticate into the email service and notice that the initial messages were not sent encrypted.

Steps to Reproduce (IMAP/SMTP Unencrypted)

1. Open the DeltaChat application.
2. Configure the email address to **bob@comcast.net** and the password to **password123**.
3. Use the following configurations for IMAP:

```
(Inbox)
login name = bob
IMAP server = mail.comcast.net
IMAP port = 143
IMAP security = off

(Outbox)
SMTP login name = bob
SMTP password = password123
SMTP server = mail.comcast.net
SMTP port  = 25
SMTP security = off
```

1. Open **Wireshark** and capture on the network interface.
2. Click **Login** in DeltaChat application.
3. Notice that Wireshark shows that the DeltaChat application tries to authenticate to **mail.comcast.net** via IMAP on port **143** using the configured username and password.

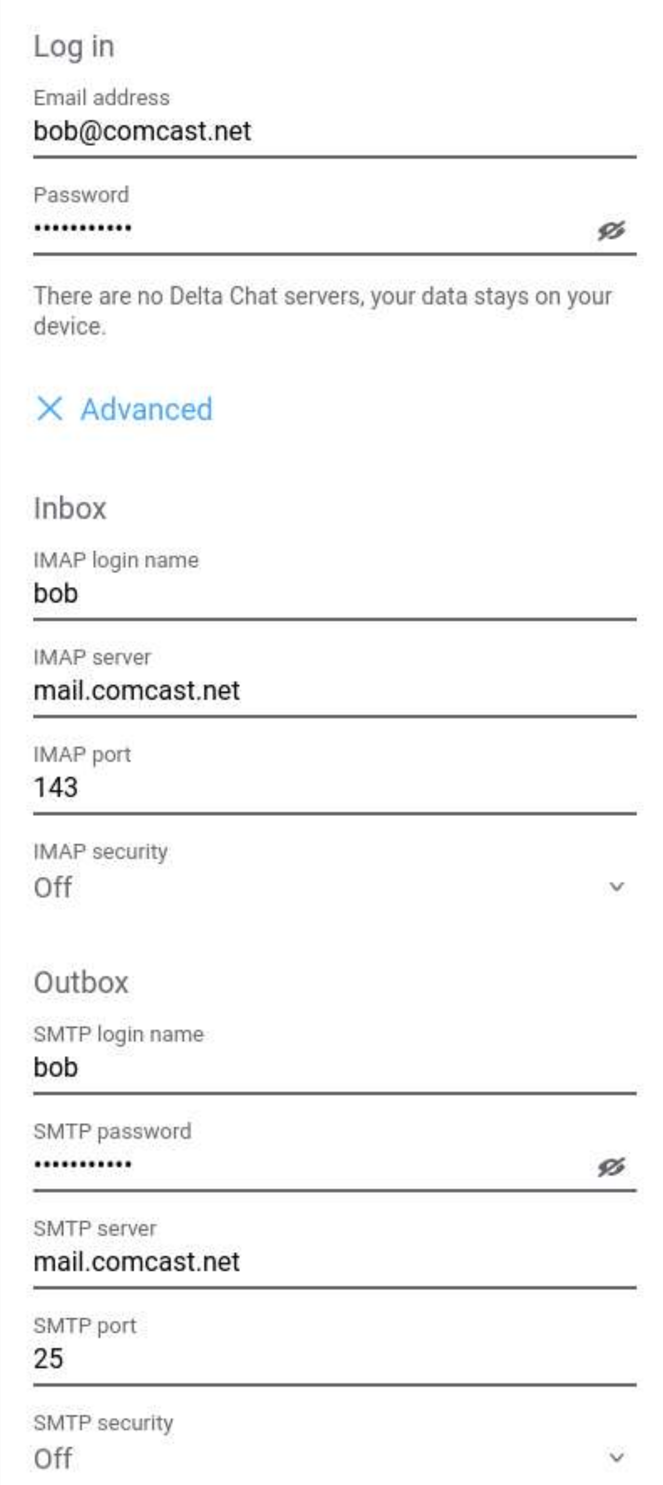
The following is an initial conversation where the initial messages were not encrypted:

```
<pre>brockefella509@gmail.com


        Wed, Jul 8, 11:38 PM (9 days ago)
Secure-Join: vc-request
brockefella509@gmail.com
```

```
        Wed, Jul 8, 11:38 PM (9 days ago)
hi buddy
brockefella509@gmail.com

        Wed, Jul 8, 11:39 PM (9 days ago)
hi buddy2
brockefella509@gmail.com

        Wed, Jul 8, 11:39 PM (9 days ago)
Secure-Join: vc-request
brockefella509@gmail.com

        Wed, Jul 8, 11:39 PM (9 days ago)
yo
brockefella509@gmail.com

        Wed, Jul 8, 11:39 PM (9 days ago)
aaaa
brockefella509@gmail.com

        Wed, Jul 8, 11:40 PM (9 days ago)
ffff
brockefella509@gmail.com

        Wed, Jul 8, 11:40 PM (9 days ago)
ddddd
Mark

Wed, Jul 8, 11:41 PM (9 days ago)

to me
Secure-Join: vc-request
</pre>
```

The following is a screenshot of a registration request via a **DCACCOUNT** QR code registration unencrypted over HTTP to http://includesecurity.com.

The following screenshots show authentication into an email address over SMTP via an unencrypted connection:



The following screenshot shows a network capture of the username and password being sent in cleartext.

```
162 276.555988011 10.139.1.1      10.137.0.53      TCP   54 53 → 41216 [FIN, ACK] Seq=875 Ack=74 Win=6
163 276.556024734 10.137.0.53     10.139.1.1       TCP   54 41216 → 53 [ACK] Seq=74 Ack=876 Win=64128
164 276.580180438 96.118.19.7     10.137.0.53      TCP   66 143 → 37496 [SYN, ACK] Seq=0 Ack=1 Win=268
165 276.580220724 10.137.0.53     96.118.19.7      TCP   54 37496 → 143 [ACK] Seq=1 Ack=1 Win=64256 Le
166 276.621502102 96.118.19.7     10.137.0.53      IMAP  169 Response: * OK [CAPABILITY IMAP4rev1 SASL-
167 276.621539622 10.137.0.53     96.118.19.7      TCP   54 37496 → 143 [ACK] Seq=1 Ack=116 Win=64256
168 276.621653422 10.137.0.53     96.118.19.7      TCP   59 37496 → 143 [PSH, ACK] Seq=1 Ack=116 Win=6
169 276.665624022 96.118.19.7     10.137.0.53      TCP   54 143 → 37496 [ACK] Seq=116 Ack=6 Win=27136
170 276.665661076 10.137.0.53     96.118.19.7      IMAP  82 Request: A0001 LOGIN "bob" "password123"
171 276.705027264 96.118.19.7     10.137.0.53      TCP   54 143 → 37496 [ACK] Seq=116 Ack=34 Win=27136
172 276.705058023 96.118.19.7     10.137.0.53      IMAP  306 Response: A0001 NO [PRIVACYREQUIRED] Plain
173 276.705248144 10.137.0.53     96.118.19.7      TCP   54 37496 → 143 [FIN, ACK] Seq=34 Ack=368 Win=
174 276.747593763 96.118.19.7     10.137.0.53      TCP   54 143 → 37496 [FIN, ACK] Seq=368 Ack=35 Win=
175 276.747623181 10.137.0.53     96.118.19.7      TCP   54 37496 → 143 [ACK] Seq=35 Ack=369 Win=64128
176 281.544301569 fe:ff:ff:ff:ff:ff Xensourc_5e:6c:00 ARP 42 Who has 10.137.0.53? Tell 10.137.0.6
177 281.544312203 Xensourc_5e:6c:00 fe:ff:ff:ff:ff:ff ARP 42 10.137.0.53 is at 00:16:3e:5e:6c:00
▸ Frame 170: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface eth0, id 0
▸ Ethernet II, Src: Xensourc_5e:6c:00 (00:16:3e:5e:6c:00), Dst: fe:ff:ff:ff:ff:ff (fe:ff:ff:ff:ff:ff)
▸ Internet Protocol Version 4, Src: 10.137.0.53, Dst: 96.118.19.7
▾ Transmission Control Protocol, Src Port: 37496, Dst Port: 143, Seq: 6, Ack: 116, Len: 28
    Source Port: 37496
    Destination Port: 143
    [Stream index: 12]
    [TCP Segment Len: 28]
    Sequence number: 6    (relative sequence number)
    Sequence number (raw): 91405569
    [Next sequence number: 34    (relative sequence number)]
```

Recommended Remediation:

While the assessment team understands that many of these unencrypted mediums are design decisions to allow a greater number of users to utilize the client, by supporting unencrypted transmission of sensitive information the DeltaChat client introduces additional risk to the tool and potentially to unsuspecting users. The assessment team recommends protecting sensitive information such as credentials from eavesdropping by removing support for unencrypted protocols such as HTTP/IMAP/SMTP and using transport layer security mechanisms such as HTTPS/IMAPS/etc for the affected application areas. In addition, removing support for disabling TLS certificate verification.

The assessment team also recommends modifying the **Send** button to include a lock icon after the successful establishment and verification of key material. In addition, showing an unlock icon within the send button upon the initial key material establishment and verification could help indicate to the user that they are about to send an unencrypted message.

References:

[OWASP Top 10 2010-A9-Insufficient Transport Layer Protection]
[Transport Layer Protection Cheat Sheet]

M2: Messages, Encryption Keys, and Email Password Stored in Cleartext

Description:

The **DeltaChat** application stores messages, encryption keys, and email passwords in cleartext. The DeltaChat application utilizes a **sqlite3** database for storing messages and other sensitive information. If this database file were recovered, an attacker could retrieve DeltaChat messages, encryption keys, contact information, geolocation data, security tokens, and other security-relevant data. Mobile application backups such as Android and iPhone are stored in the cloud. These application backups could have sensitive information and could be retrieved by law enforcement or advanced persistent

threat (APT) adversaries with backend cloud access. Note that Google added an optional key encryption protection for backups that can be leveraged with some mobile devices that would help mitigate the retrieval of cloud data.

Note from Delta Chat Team:
"On Android and iOS you typically get device encryption and even on Desktop devices it's more common. It's what we typically recommend to our users — don't just protect your chat data, protect your whole account. See the related Autocrypt specification note on this:"

Secret Key Protection at Rest

Affected Locations

- **deltachat-core-rust/src/sql.rs**
- **/home/user/.config/DeltaChat/accounts/ac1/db.sqlite**

Steps to Reproduce (DeltaChat Linux Desktop Application)

1. Setup DeltaChat and start chatting with another DeltaChat user.
2. Change the directory to the application directory.

```
cd /home/user/.config/DeltaChat/accounts/ac1
```

3. Open with sqlite3 database file:

```
sqlite3 db.sqlite
```

4. Display the database tables:

```
.tables
```

5. Read information from the tables:

```
select * from msgs;
```

6. Notice that the messages are stored in cleartext.

Steps to Reproduce (DeltaChat Android Application)

1. Install Genymotion, Android Emulator or utilize a Rooted Android Device.
2. Install DeltaChat and start chatting with another DeltaChat user.
3. Initialize adb as root:

```
adb root
```

4. Download the application data:

```
adb pull [directory of mobile application on device] ./
```

5. Find the sqlite3 database file and open the file:

```
sqlite3 db.sqlite
```

6. Display the tables:

```
.tables
```

5. Read information from the tables:

```
select * from msgs;
```

6. Notice that the messages are stored in cleartext.

The following screenshot displays an encrypted message exchange with the DeltaChat Mobile application:

The next screenshot shows that the sqlite3 database file can be retrieved from the Android device and are not encrypted.

```
--
test1111
|c=1|0|1595373410|1595373417|0||<Mr.N3MD9Q_ONw1.2ES8tvLeKMg@dubby.org>
om> <Mr.N3MD9Q_ONw1.2ES8tvLeKMg@dubby.org>|1|0|
20|Mr.cpYp5tOArRg.w8Y-KaoU6Sm@dubby.org|DeltaChat|112|12|10|1|15953734
m brockefella509@gmail.com

9999999

--
test1111
|c=1|0|1595373425|1595373434|0||<Mr.xT8x7x6br-E.34bVinHdXoA@dubby.org>
om> <Mr.xT8x7x6br-E.34bVinHdXoA@dubby.org>|1|0|
21|Mr.lYjTlNYxak2.uLo1WBNQVG5@dubby.org|DeltaChat|113|12|10|1|15953734
m brockefella509@gmail.com

test123

--
```

Notice that the **test123** and **9999999** messages are sent by the mobile application encrypted (i.e. have the lock icon) and is located in the sqlite3 database unencrypted in cleartext.

Recommended Remediation:

The assessment team recommends encrypting the database content with AES-256-GCM128 or a similar encryption mode and deriving a strong symmetric private key. A symmetric private key could be generated by utilizing a user password and a key derivation function (KDF) like PBKDF2 or Scrypt. Some mobile devices support hardware security modules that could be used to create and store encryption keys that can be accessed by passwords or biometric information. By leveraging a strong encryption key and a strong encryption algorithm, it would make it difficult for an attacker to decrypt the contents of the messages or retrieve sensitive information. Encrypted user information would also help protect mobile backups and lost or the information on stolen devices.

References:

Salt and Hash Password with PBKDF2
Rust Crypto
Rust Encryption

M3: DeltaChat Message Export/Backups Are Unencrypted

Description:

The **DeltaChat** application creates backups that are unencrypted. If an attacker were able to gain access to a DeltaChat backup file, then the attacker could read messages and message information from a targeted user. While the DeltaChat application does provide guidance to store the backup file in a secure location, users could ignore or not understand the impact of this advice and fall prey to a situation where the expectation of confidentiality is disrupted.

Affected Location

- **deltachat-core-rust/src/imex.rs**

Steps to Reproduce

1. Click the top left-hand corner button.
2. Click **Settings**.
3. Scroll down to the bottom and click **Export Backup**.
4. Select a directory to output the file.
5. Run the **file** command on the backup file and notice that the file is a sqlite3 database file.
6. Install the SQLite database application if the application is not already installed (e.g. **sudo apt-get install sqlite**).
7. Open the backup database file with sqlite (i.e. **sqlite3 delta-chat-2020-07-17-0.bak**).
8. View all messages (i.e. **select * from msgs;**).

The following is an example set of commands demonstrating this issue:

```
file delta-chat-2020-07-17-0.bak
delta-chat-2020-07-17-0.bak: SQLite 3.x database, last written using SQLite version 3031001

sqlite3 delta-chat-2020-07-17-0.bak

sqlite> .tables
acpeerstates    chats_contacts  devmsglabels    leftgrps        msgs_mdns
backup_blobs    config          jobs            locations       tokens
chats           contacts        keypairs        msgs

sqlite> select * from msgs
   ...> ;
|c=1
r=1|0|1594248300|1594248302|0||<Gr.J3jX6ORtCky.C3fGQFl4FOS@dubby.org>|<Gr.J3jX6ORtCky.aX5iykc0FrE@deltachat
.de> <Gr.J3jX6ORtCky.C3fGQFl4FOS@dubby.org>|1|0|
40|Gr.J3jX6ORtCky.VOVDJGhX2iq@dubby.org||0|13|1|0|1594248331|10|26|1|0|yup. good to meat you digitally
holger||c=1|0|0|0|0||Gr.J3jX6ORtCky.XGFgkl_23_N@deltachat.de|<Gr.J3jX6ORtCky.aX5iykc0FrE@deltachat.de>
Gr.J3jX6ORtCky.XGFgkl_23_N@deltachat.de|1|0|
41|Gr.J3jX6ORtCky.nBPuSdDHpwT@deltachat.de|DeltaChat|7|13|10|1|1594248331|10|16|1|0|you can change your
display name in your profile settings.|Re: Sec-Review2
```

Notice that the messages, while encrypted in transit, are not encrypted in the backup sqlite3 database file.

Recommended Remediation:

The assessment team recommends implementing encrypted backups. Encrypted backups could leverage AES-256-GCM-128 and a strong password (or device generated that the user could write down) and a key derivation algorithm PBKDF2 or scrypt. For ease of use, a user should use a strong password (or device generated key), and the PBKDF2 or scrypt algorithms can be leveraged to generate a symmetric encryption key that can be used to encrypt the database file.

References:

PBKDF2 Wikipedia
Scrypt Wikipedia
Rust Crypto Crate

M4: Server-Side Request Forgery (SSRF)

Description:

A Server-Side Request Forgery (SSRF) issue was discovered in the **DeltaChat** application. SSRF issues occur when a user can supply a hostname or URL to the server which will cause the server to make a request to that host. Attackers can use SSRF vulnerabilities to attack or probe internal network services that are available to the server (but not available externally on the Internet) to attack other services on the Internet or cause requests from the server to be made into an attacker-controlled server enabling the attacker to control the response. An SSRF could also be leveraged against DeltaChat users to de-anonymize their identity or attack services on localhost (e.g. **127.0.0.1**).

Affected Location

- **deltachat-core-rust-master/src/qr.rs**

The following source code demonstrates this issue.

```
189 /// scheme: `DCACCOUNT:https://example.org/new_email?t=1w_7wDjgjelxeX884x96v3`
190 fn decode_account(_context: &Context, qr: &str) -> Lot {
191     let payload = &qr[DCACCOUNT_SCHEME.len()..];
192
193     let mut lot = Lot::new();
194
195     if let Ok(url) = url::Url::parse(payload) {
196         if url.scheme() == "https" {
197             lot.state = LotState::QrAccount;
198             lot.text1 = url.host_str().map(|x| x.to_string());
199         } else {
200             lot.state = LotState::QrError;
201             lot.text1 = Some(format!("Bad scheme for account url: {}", payload));
202         }
203     } else {
204         lot.state = LotState::QrError;
205         lot.text1 = Some(format!("Invalid account url: {}", payload));
206     }
207
208     lot
```

```
209 }
...
217 /// take a qr of the type DC_QR_ACCOUNT, parse it's parameters,
218 /// download additional information from the contained url and set the parameters.
219 /// on success, a configure::configure() should be able to log in to the account
220 pub async fn set_config_from_qr(context: &Context, qr: &str) -> Result<(), Error> {
221     let url_str = &qr[DCACCOUNT_SCHEME.len()..];
222
223     let response: Result<CreateAccountResponse, surf::Error> =
224         surf::post(url_str).recv_json().await;
225     if response.is_err() {
226         bail!("Cannot create account, request to {} failed", url_str);
227     }
228     let parsed = response.unwrap();
229
230     context
231         .set_config(Config::Addr, Some(&parsed.email))
232         .await?;
233     context
234         .set_config(Config::MailPw, Some(&parsed.password))
235         .await?;
236
237     Ok(())
238 }
```

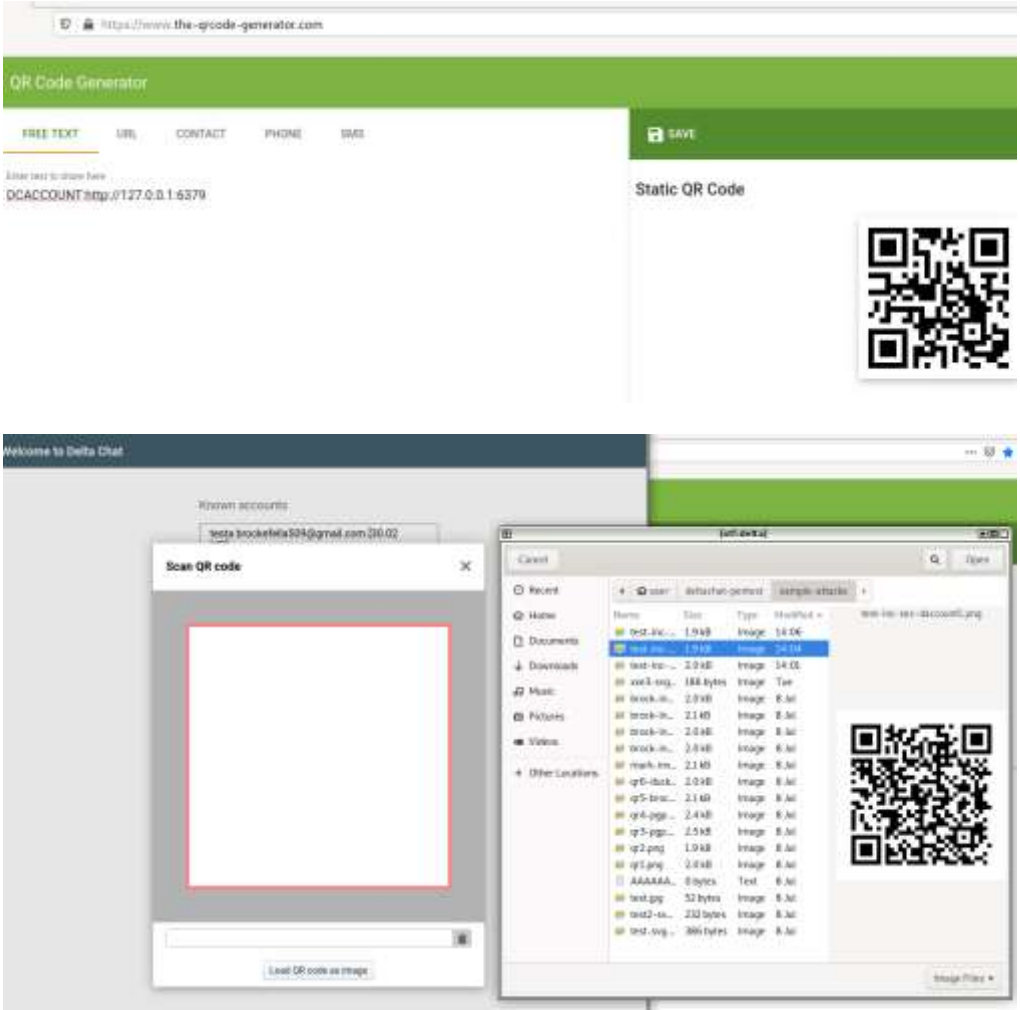
Steps to Reproduce

1. Go to https://www.the-qrcode-generator.com/.
2. Enter **DCACCOUNT:**http://127.0.0.1:6379.
3. Right-click and save the image.
4. Start python webserver.

```
mkdir ./tmp/ && cd ./tmp && python3 -m http.server 6379
```

5. Deauthenticate if already authenticated by clicking the top right settings button and clicking **Switch Account**.
6. Click Scan QR Code on bottom of Delta Welcome Chat Page
7. Open the created QR file.
8. Notice that a request is sent to http://127.0.0.1:6379.

The following screenshots demonstrate this issue:



The result of opening the QRCode generated the following HTTP request:

```
python -m http.server 6379
Serving HTTP on 0.0.0.0 port 6379 (http://0.0.0.0:6379/) ...
127.0.0.1 - - [17/Jul/2020 14:44:26] code 501, message Unsupported method ('POST')
127.0.0.1 - - [17/Jul/2020 14:44:26] "POST / HTTP/1.1" 501 -
```

Recommended Remediation:

Whenever possible, do not trust user-controllable URLs when web requests need to be made by the server to other services. The code should not be allowed to make requests to internal network hosts or localhost, even via redirects from external hosts. If user-controllable URLs must be requested, then sanitizing them in a manner similar to the SafeCurl library is recommended (see the link in the reference section).

Additionally, a whitelist of acceptable characters could be created and support for HTTPS could be mandated.

References:

[Safecurl Libraries]
[Paranoid Request]

M5: Social Engineering Attack via Group Messaging UI

Description:

The assessment team identified a potential social engineering attack whereby an attacker could trick one user into believing they are another via cleverly named DeltaChat group. In this case, an attacker who can learn the name of a user and their user icon could create a group with the name of the target and invite another targeted user to the chat to potentially impersonate and disrupt expected confidentiality. While this attack could be detected by a user by right-clicking on the group name or by clicking on the top middle part of the GUI after selecting the group, it is possible that the user would not realize that they are sending information to a group before it is too late.

Steps to Reproduce

1. Initialize a chat with a user named Bob.
2. Download Bob's chat icon.
3. Create a group chat named Bob.
4. Set the group chat icon to Bob's icon.
5. Invite a user named Alice to the Group chat.
6. Wait for Alice to send an unsuspecting message to the group chat named Bob.

The following screenshot is an example of creating a Group named **holger** and utilizing holger's icon to socially engineer a test user Brock from the test user Mark.

If a user knows the DeltaChat application well, they might be able to see that the top middle of the application says, 2 members. However, a small oversight might allow an attacker to steal some information.

Recommended Remediation:

The assessment team recommends changing the user interface to more easily discern whether a chat entity is a group or an individual. This could help prevent related social engineering attacks.

References:

Twitter Social Engineering Attack Security Incident
Social Engineering Wikipedia

M6: File Transfers Do Not Validate Filetypes

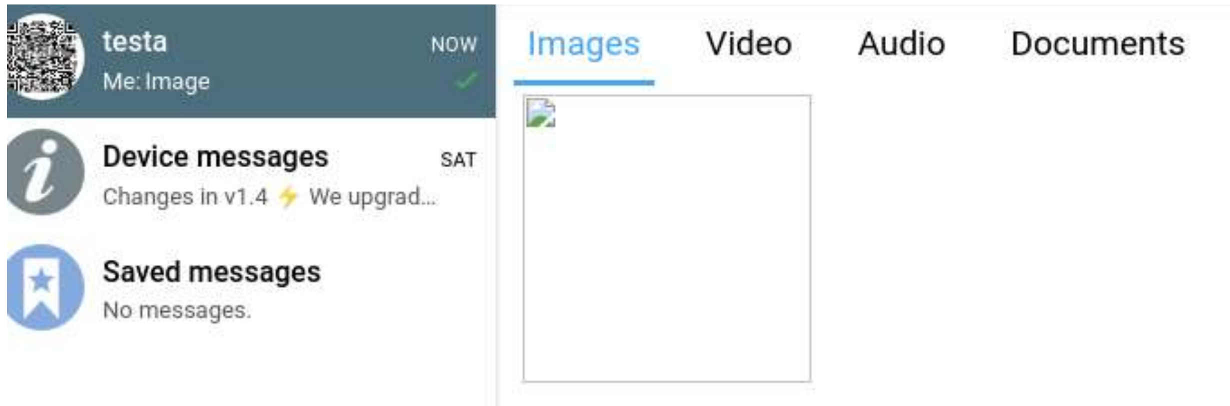
Description:

The assessment team found that the **DeltaChat** application can be used to transfer files. The files sent are organized in a panel that separates them by images, video, audio or documents. The filetypes or content of the files are not validated, which allows an attacker to trick a user by renaming a file's extension. Files are automatically written to the filesystem when transferred and could result in file attacks on the operating system or lead to de-anonymization attacks.

Steps to Reproduce (file transfer filetype mismatch)

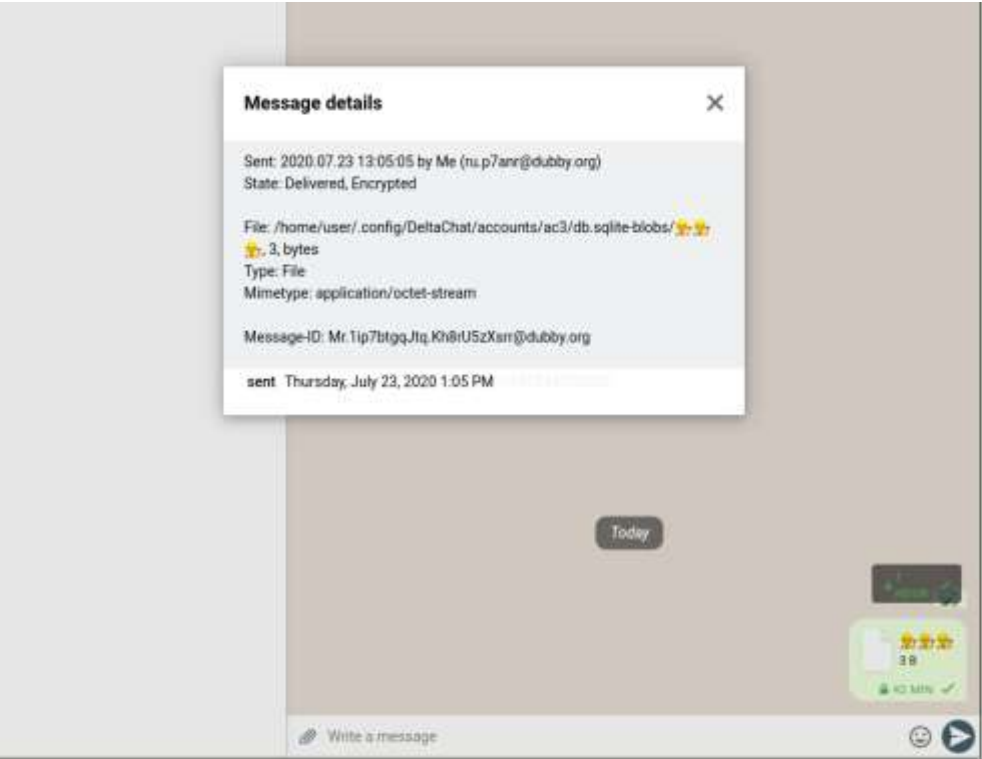
1. Download and install the DeltaChat desktop application.
2. Authenticate into the application with an email.
3. Begin communication with another user.
4. Create a file where the content does not match its extension such as **test123.jpg**.
5. Send the file using the DeltaChat application.
6. Notice that the file is stored in the images section of the DeltaChat client but the file is not a valid JPG image.

The following image shows the results of transferring and displaying a JPG file that is not actually a JPG file:



Steps to Reproduce (transferring a file with UTF-8 characters in the name)

1. Download and install the DeltaChat desktop application
2. Login to the application with an email
3. Begin communication with another user
4. Create a file that includes UTF-8 characters in the name
5. Transfer the file to a user
6. Notice that the transferred file was stored on the filesystem and contains UTF-8 characters



Steps to Reproduce (de-anonymization with AVI file and Kmplayer)

1. Download and install the DeltaChat desktop application.
2. Authenticate into the application with an email.
3. Begin communication with another user.
4. Create a file that includes the following content and name it **test.avi**:

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://incudesecurity.com
#EXT-X-ENDLIST
```

5. Install kmplayer on Linux

```
sudo apt-get install kmplayer
```

6. Transfer the file to a user.
7. Open the AVI file that was transferred.
8. Notice that a request was made to **includesecurity.com**.
9. Note that depending on the operating system, applications installed and the filename, there are ways to cause applications to send remote requests that could cause de-anonymization, credential theft, or in some cases remote code execution. Sometimes, it is not required that the user opens the file (e.g. Windows LNK vulnerabilities, Evince PDF viewer thumbnail vulnerabilities, Android Stagefright Vulnerability, Microsoft Windows Defender vulnerabilities, etc.)

In this case a user would need to double click the AVI file from within the DeltaChat application and have kmplayer installed. The assessment team is certain that there are many other ways that this could be done. For example, there are some cases on Windows when a file hits disk Windows can automatically send out a network call to an SMB service remotely. If a user's firewall rules allow outgoing SMB, then that could disclose SMB hashes.

Recommended Remediation:

The assessment team recommends implementing the following in order of priority:

1. Verify that the magic bytes of the file (first several bytes) match the extension or utilize the magic bytes of the file to denote the filetype.
2. If a secure communication channel has been established between two users, then do not accept attachments that are sent unencrypted via email from the user who has established a secure communication.
3. In cases where an active man-in-the-middle (MitM) attack could have occurred, then do not automatically download files to the filesystem as automatically downloading files could result in de-anonymization attacks or allow an attacker to launch exploits via operating system file parsing issues or allow for social engineering attacks.
4. Do not allow UTF-8 characters or special characters (e.g. `, %, ', #) in filenames that are being transferred.
5. Do not automatically parse untrusted KML files.
6. Notify the user before transferring files especially if PGP key material has changed. For example, “The user [insert_username] wants to transfer you a file but the PGP public keys have changed. Would you like to allow this file transfer?”
7. Create a maximum size for filenames and do not allow filenames to transfer that are greater than a certain length.

Please note that the above-recommended remediation steps are a “practical” recommendation. A more complete solution would be more complete and correct in the file format checking to avoid corkami inspired filetype attacks.

References:

List of File Signatures
Windows LNK Exploit
Project Zero Windows Defender Exploitation
Evince Command Injection Exploit
Stealing Windows Credentials Using Google Chrome
Facebook and FBI De-anonymization Exploit
Stagefright Bug Wikipedia
Email Right to Left Override Aids Email Attacks

LOW-RISK FINDINGS

L1: Confidential Information in Logs

Description:

Confidential information such as communication times, number of messages, communication email addresses, and sent emojis are stored in various potentiality unencrypted and/or unintended locations. These include log files, configuration files, and local storage. Additionally, these locations and not deleted when ephemeral chat messaging is enabled. If sensitive information is being transmitted and ephemeral chat mode is enabled, then it would be best practice not to log that information and to not store emoji information in local storage of the DeltaChat desktop application.

Affected Location

- **deltachat-core-rust/src/ephemeral.rs**
- **/home/user/.config/DeltaChat/config.json**
- **/home/user/.config/DeltaChat/logs/[name-of-logfile]**

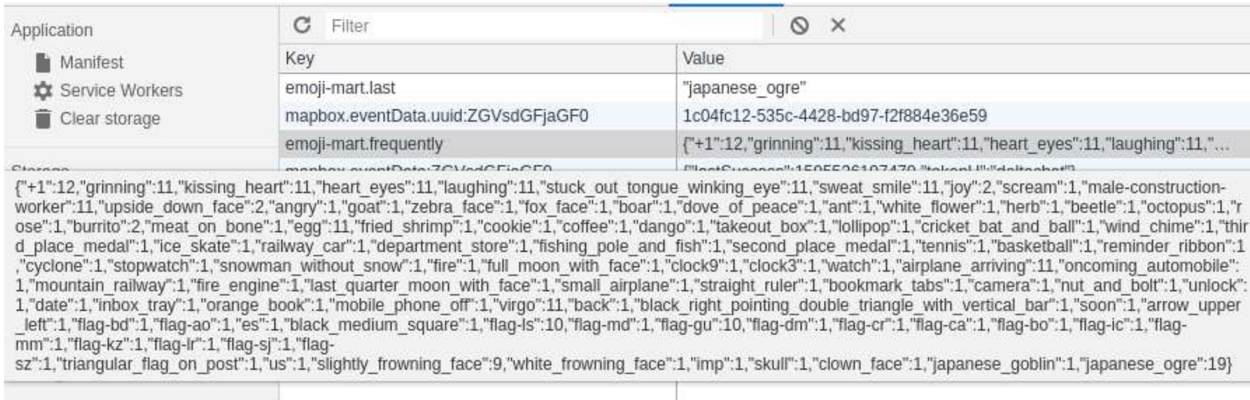
Steps to Reproduce (emojis)

1. Install the DeltaChat desktop application.
2. Authenticate and set up a DeltaChat account.
3. Create a secure chat connection with another DeltaChat user.
4. In settings, turn on ephemeral messaging and set ephemeral messages to delete every 1 hour.
5. Send emojis to the user via the chat window.
6. Wait 1 hour.
7. Click ViewDeveloperDeveloper Tools.
8. Click the Local Storage **Error! Hyperlink reference not valid.** dropdown.
9. Notice that the emojis and number of emojis are stored in local storage and were not deleted.

Steps to Reproduce (log file)

1. Install the DeltaChat desktop application.
2. Run the DeltaChat desktop application and notice the log file that is being used.
3. Authenticate and set up a DeltaChat account.
4. Create a secure chat connection with another DeltaChat user.
5. In settings, turn on ephemeral messaging and set ephemeral messages to delete every 1 hour.
6. Send chat information to another user.
7. Wait 1 hour.
8. Open the log file and notice that the metadata regarding the messages sent to a user is stored in the log file.

The following is a screenshot of emojis used when ephemeral messaging is enabled:



Recommended Remediation:

The assessment team recommends not logging metadata when ephemeral messaging is enabled or removing information when messages are sent. In addition, the assessment team recommends not storing data in local storage for the DeltaChat desktop application. As additional protection, the number of messages in the **config.js** should also not be updated when performing ephemeral messaging.

References:

[Information Exposure Through Log Files](#)

L2: Content Security Policy Allows Unsafe-inline

Description:

The desktop version of the **DeltaChat** application utilizes **Electron** for the front-end user interface. Electron applications run in an un-sandboxed Chromium instance. If an attacker identifies a cross-site scripting flaw in the Electron application, then it will likely result in remote code execution. Desktop applications such as Signal Private Messenger, Wire, and Discord have had multiple flaws that would allow an attacker to achieve remote code execution.

To help mitigate this threat, the desktop DeltaChat application leverages a content security policy (CSP) and an **eval()** function deny list. The CSP policy for the desktop application is the following:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'none';
style-src 'self' 'unsafe-inline';
font-src 'self';
script-src 'self';
worker-src blob: ;
child-src blob: ;
img-src 'self' data: blob: ;
```

```
media-src 'self';
connect-src https://*.tiles.mapbox.com https://api.mapbox.com https://events.mapbox.com 'self'">
```

Notice that **style-src** allows unsafe-inline HTML tags. This could potentially allow an attacker to subvert CSP protections and achieve HTML injection or cross-site scripting (XSS) if an XSS vulnerability existed. The **connect-src** policy also allows a wildcard policy. If an attacker performed a subdomain takeover of a domain on **tiles.mapbox.com**, then an attacker could leverage that domain to perform Ajax and WebSocket communication and potentially exfiltrate sensitive information.

Recommended Remediation:

The assessment team recommends the following steps be taken to improve the CSP. The list below is in order of ROI priority:

1. Remove **unsafe-inline** from the CSP policy.
2. Add support for CSP whitelist hashing. This technique will only allow a whitelist of scripts that have the correct SHA256 hash.
3. Remove the * **from the *connect-src "https://** from the **connect-src https://.tiles.mapbox.com** policy and only allow a list of trusted domains. This can help mitigate against subdomain registration or takeover, which is a popular technique when websites do not renew their domain and/or SSL certificates.
4. Harden the Electron Sandbox security configurations of the Electron Application. This can be done by making sure **nodeIntegration** is false, **webSecurity** is enabled, experimental features is not enabled, **allowPopups** is not enabled, Electron framework/NodeJS is up-to-date and **allowRunningInsecureContent** is not set to true. Additional recommendations can be found in the Do Not Enable NodeJS Integration link in the references section.
5. Audit the Electron code for XSS vulnerabilities. The Electron code was out of scope for this engagement but hardening this code could be helpful in mitigating XSS attacks which may result in preventing remote code execution vulnerabilities.
6. Add additional Electron/JavaScript functions to the deny list.
7. Remove the developer tools (ViewDeveloperDeveloper Tools) option from the production release of the application. This could help mitigate against privileged JavaScript functions from being loaded.
8. Remove or reduce support for UTF-8 where possible.

References:

Locking Down Your Website Scripts With CSP Hashes Nonces and Report URI
Electron Framework Security
Signal Private Messenger RCE Vulnerability
Do Not Enable NodeJS Integration for Remote Content
Preloading Insecurity in Your Electron

L3: Potential Denial of Service via Large File Transfer or Large Messages

Description:

The assessment team noticed that there are limits on the size of outgoing attachments and messages but there do not appear to be limits on the size of incoming attachments or messages. If an attacker were to gain control of an email provider or the email messages in transit, then they could potentially inject large amounts of data to cause a Denial of Service (DOS) on the DeltaChat application.

Note: Creating a from-scratch email server setup to test arbitrarily large attachments was out of scope for this assessment due to time constraints. That being said, the assessment team consistently sent large amounts of data via messages and did not notice any of the messages being blocked. As such, creating a full proof-of-concept is expected to be possible given the time to create a full testing environment for this scenario.

Affected Location

- **deltachat-core-rust/src/dc_receive_imf.rs**

The following code sample is where messages can be inserted into the database. The assessment team did not identify any incoming message size checks or incoming limits.

```
 710     // fine, so far.  now, split the message into simple parts usable as "short messages"
711     // and add them to the database (mails sent by other messenger clients should result
712     // into only one message; mails sent by other clients may result in several messages
713     // (eg. one per attachment))
714     let icnt = mime_parser.parts.len();
715
716     let subject = mime_parser.get_subject().unwrap_or_default();
717
718     let mut parts = std::mem::replace(&mut mime_parser.parts, Vec::new());
719     let server_folder = server_folder.as_ref().to_string();
720     let is_system_message = mime_parser.is_system_message;
721     let mime_headers = if save_mime_headers {
722         Some(String::from_utf8_lossy(imf_raw).to_string())
723     } else {
724         None
725     };
726     let sent_timestamp = *sent_timestamp;
727     let is_hidden = *hidden;
728     let chat_id = *chat_id;
729
730     // TODO: can this clone be avoided?
731     let rfc724_mid = rfc724_mid.to_string();
732
733     let (new_parts, ids, is_hidden) = context
734         .sql
735         .with_conn(move |mut conn| {
736             let mut ids = Vec::with_capacity(parts.len());
737             let mut is_hidden = is_hidden;
738
739             for part in &mut parts {
740                 let mut txt_raw = "".to_string();
741                 let mut stmt = conn.prepare_cached(
```

```
742                     "INSERT INTO msgs \
743          (rfc724_mid, server_folder, server_uid, chat_id, from_id, to_id, timestamp, \
744          timestamp_sent, timestamp_rcvd, type, state, msgrmsg,  txt, txt_raw, param, \
745          bytes, hidden, mime_headers,  mime_in_reply_to, mime_references, error, ephemeral_timer) \
746          VALUES (?,?,?,?,?,?, ?,?,?,?,?,?, ?,?,?,?,?,?, ?,?, ?,?);",
747                 )?;
```

Recommended Remediation:

The assessment team recommends implementing stronger attachment file size and message size limitations for incoming messages. If large messages need to be sent, then message and attachment size restrictions could be implemented for non-verified users. Alternatively a global setting, or individual prompting may be ways to reduce the impact of large file transfers.

References:

[Denial of Service Wikipedia](#)

L4: Cryptographically Deprecated SHA1 Hashing Algorithm in Use

Description:

The **DeltaChat** application supports the SHA1 hashing algorithm as a cryptographic primitive of OpenPGP for messaging. SHA1 has known cryptographic weaknesses. The current cryptography best practices recommend the use of the SHA256, SHA384, and SHA512 alternatives.

Affected Location

- **deltachat-core-rust/src/pgp.rs**

The following source code displays the option of leveraging SHA1:

```
148 pub(crate) fn create_keypair(
149     addr: EmailAddress,
150     keygen_type: KeyGenType,
151 ) -> std::result::Result<KeyPair, PgpKeygenError> {
152     let (secret_key_type, public_key_type) = match keygen_type {
153         KeyGenType::Rsa2048 => (PgpKeyType::Rsa(2048), PgpKeyType::Rsa(2048)),
154         KeyGenType::Ed25519 | KeyGenType::Default => (PgpKeyType::EdDSA, PgpKeyType::ECDH),
155     };
156
157     let user_id = format!("<{}>", addr);
158     let key_params = SecretKeyParamsBuilder::default()
159         .key_type(secret_key_type)
160         .can_create_certificates(true)
161         .can_sign(true)
162         .primary_user_id(user_id)
163         .passphrase(None)
164         .preferred_symmetric_algorithms(smallvec![
165             SymmetricKeyAlgorithm::AES256,
166             SymmetricKeyAlgorithm::AES192,
```

```
167             SymmetricKeyAlgorithm::AES128,
168         ])
169         .preferred_hash_algorithms(smallvec![
170             HashAlgorithm::SHA2_256,
171             HashAlgorithm::SHA2_384,
172             HashAlgorithm::SHA2_512,
173             HashAlgorithm::SHA2_224,
174             HashAlgorithm::SHA1,
175         ])
```

Recommended Remediation:

The assessment team recommends removing support for SHA1 as a preferred hashing algorithm. Removing the SHA1 algorithm would help protect the integrity of messages and prove the origin of the message (i.e. non-repudiation).

References:

Collision Resistance
Announcing First SHA1 Collision
From Collisions to Chosen Prefix Attacks

L5: Application Build Does Not Employ Position-Independent Executable (PIE) Flag, RELRO, and Stack Cookie Protections

Description:

The **DeltaChat** desktop application is not compiled with the position-independent executable (**-pie**) flag, RELRO, or stack cookies. This PIE flag will allow operating systems such as Linux to easily randomize and remap memory regions. Compiling an executable to be position-independent is an effective security mitigation that can make it much more difficult for an attacker to exploit memory corruption vulnerabilities. RELRO is a relocation read-only protection that helps protect against Global Offset Table (GOT) overwrites.

If a binary is not compiled with the PIE flag, then it will not take full advantage of address space layout randomization (ASLR) on Linux-like OSes. ALSR is a memory corruption mitigation technology that randomizes the layout of process memory which increases the exploit complexity for an attacker.

If a binary is compiled with stack cookie protection, then it will make it more difficult for an attacker to exploit stack-based buffer overflow memory corruption vulnerabilities.

The following commands show that the DeltaChat desktop application is not compiled with the PIE flag, RELRO or with stack cookie protections:

```
checksec --file=DeltaChat-1.4.3.AppImage
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
```

```
FORTIFY Fortified        Fortifiable      FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH    No RUNPATH    No Symbols
No     0               15               DeltaChat-1.4.3.AppImage
```

Recommended Remediation:

The assessment team recommends that all DeltaChat executables be recompiled with the PIE flag, RELRO and with stack cookie protection. While there are small performance trade-offs when using these protections, the performance impacts on Linux x86_64bit are typically negligible. A full exploration of how to accomplish this within the confines of AppImage was out of the scope of this assessment, but do note that these protections might not be comparable with that system.

References:

Position Independent Executable
Address Space Layout Randomization
XCode Enable PIE
GCC hardening for 16.10

L6: Unsafe Dereference Used

Description:

The **DeltaChat** application utilizes several unsafe calls. One such unsafe code block leverages the **Pin** module which states that the **new_unchecked()** function could cause memory corruption issues in the form of unsafe pointer operations.

The Pin Rust documentation states the following:

```
because we cannot guarantee that the data pointed to by pointer is pinned, meaning that the data will not
be moved or its storage invalidated until it gets dropped. If the constructed Pin<P> does not guarantee
that the data P points to is pinned, that is a violation of the API contract and may lead to undefined
behavior in later (safe) operations.
```

While in the codes current state it is probably not exploitable, as code changes values could be changed or the code could become exploitable.

Affected Location

- **deltachat-pentest/async-smtp/src/types.rs**:116

Notice that on line 116, the unsafe keyword is used with the **Pin** module.

```
106 impl Read for Message {
107     #[allow(unsafe_code)]
108     fn poll_read(
```

```
109         self: Pin<&mut Self>,
110         cx: &mut Context,
111         buf: &mut [u8],
112     ) -> Poll<io::Result<usize>> {
113         match self.project() {
114             MessageProj::Reader(mut rdr) => {
115                 // Probably safe..
116                 let r: Pin<&mut _> = unsafe { Pin::new_unchecked(&mut **rdr) };
117                 r.poll_read(cx, buf)
118             }
119             MessageProj::Bytes(rdr) => {
120                 let _: Pin<&mut _> = rdr;
121                 rdr.poll_read(cx, buf)
122             }
123         }
124     }
125   }
126
```

Recommended Remediation:

The assessment team recommends refactoring all code to avoid unsafe calls, and more specifically to avoid using the Pin unsafe **new_unchecked()** call. By refactoring unsafe Rust code, the risk of potential memory corruption can be reduced or eliminated and memory corruption primitives can be minimized. The DeltaChat application has already dramatically reduced its attack surface by implementing much of their code in Rust. Removing more unsafe operations will only help mitigate additional risks.

References:

Pin Documentation

L7: Homograph Attacks Possible in Various Parts of DeltaChat

Description:

The **DeltaChat** application is vulnerable to a homograph attack. Homograph attacks occur when an application supports UTF characters and an attacker can embed UTF characters to disguise URLs, emails, or other sensitive information to trick a user into thinking that the content is from a trusted party. For example, an attacker could potentially register a domain name and email address with a UTF-8 character that resembles a trusted domain name or email address and leverage that in a social engineering attack.

While the assessment team understands that UTF-8 is a desired feature in many locations of the code, reducing the locations that use UTF-8 could reduce potential homograph attacks. The following are locations that were spotted in code that utilize UTF-8:

Affected Locations

- **deltachat-core-rust/src/qr.rs**:374
- **deltachat-core-rust/src/dehtml.rs**:145
- **deltachat-core-rust/src/dehtml.rs**:123
- **deltachat-core-rust/src/dehtml.rs**:94
- **deltachat-core-rust/src/message.rs**:320
- **deltachat-core-rust/src/message.rs**:309
- **deltachat-core-rust/src/securejoin.rs**:104
- **deltachat-core-rust/src/securejoin.rs**:106
- **deltachat-core-rust/src/securejoin.rs**:113
- **deltachat-core-rust/src/location.rs**:133
- **deltachat-core-rust/src/location.rs**:153
- **deltachat-core-rust/src/location.rs**:157
- **deltachat-core-rust/src/location.rs**:178

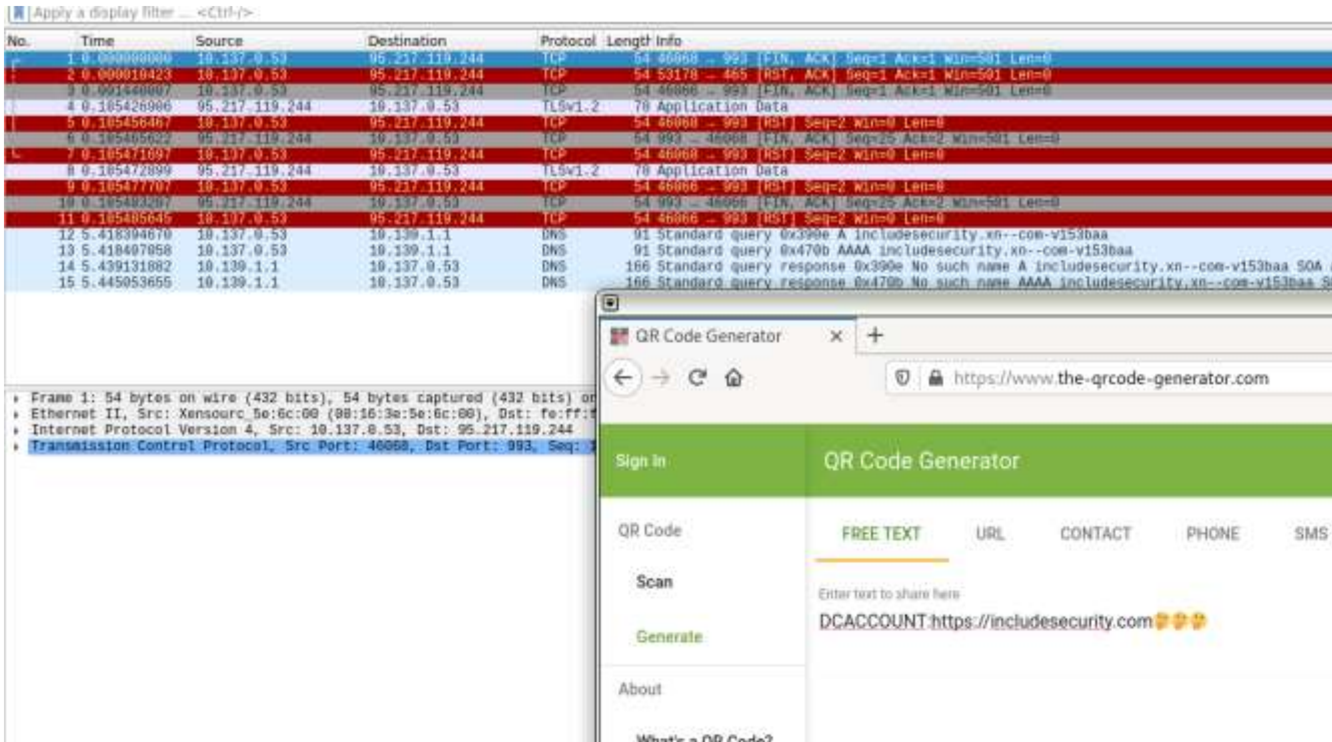
Some of these locations require UTF-8, but some of them could probably run without UTF-8 support. This will need to be a design decision made by the developers and security engineers to determine which locations to potentially alter.

The following source code in **qr.rs** demonstrates this issue:

```
371 /// URL decodes a given address, does basic email validation on the result.
372 fn normalize_address(addr: &str) -> Result<String, Error> {
373     // urldecoding is needed at least for OPENPGP4FPR but should not hurt in the other cases
374     let new_addr = percent_decode_str(addr).decode_utf8()?;
375     let new_addr = addr_normalize(&new_addr);
376
377     ensure!(may_be_valid_addr(&new_addr), "Bad e-mail address");
378
379     Ok(new_addr.to_string())
380
```

Steps to Reproduce

1. Go to https://www.the-qrcode-generator.com.
2. Under freetext insert **DCACCOUNT:**https://includesecurity.com[UTF-8_Characters]].
3. Download the QR code.
4. Open DeltaChat.
5. Click **Switch Users**.
6. Click **Scan QR Code**.
7. Start **Wireshark** and capture on the appropriate network interface.
8. Select the downloaded QR code.
9. Notice that a request is sent to the domain with UTF-8 characters.



Recommended Remediation:

The assessment team recommends removing or reducing support for UTF-8 characters throughout the application, particularly for QR Codes, email addresses, filenames, links, and domain names. This could be done by creating a whitelist of appropriate characters for certain areas of the application. This will reduce the potential that a user will trust a UTF-8 remote entity.

References:

[Email Right to Left Override Aids Email Attacks](#)
[IDN Homograph Attack Wikipedia](#)
[Phishing with Unicode Domains](#)

L8: UI Alert Does Not Convey Potential for Confidentiality Disruption

Description:

The **DeltaChat** application utilizes PGP, QR Codes, and Autocrypt to establish end-to-end communication. There are a series of controls within the application that verify cryptography to ensure that passive or active man-in-the-middle attacks do not succeed. However, if an unsuspecting user does not notice a change in key material (i.e. does not notice the “Changed setup for [email_address]” message), it could mean that an attacker has automatically updated public key material. Future messages from the established secure communication could be compromised or social engineering attacks could be conducted.

This type of attack could occur if a user's email credentials are compromised. Email credentials could be compromised using brute-force attacks, credential stuffing, compromise of a device, keylogging (software or physical), side-channel/sensor/tempest attacks, email server attacks, or unencrypted SMTP/IMAP. An attacker can use these email credentials, authenticate into the DeltaChat client and send and receive encrypted messages to and from DeltaChat users.

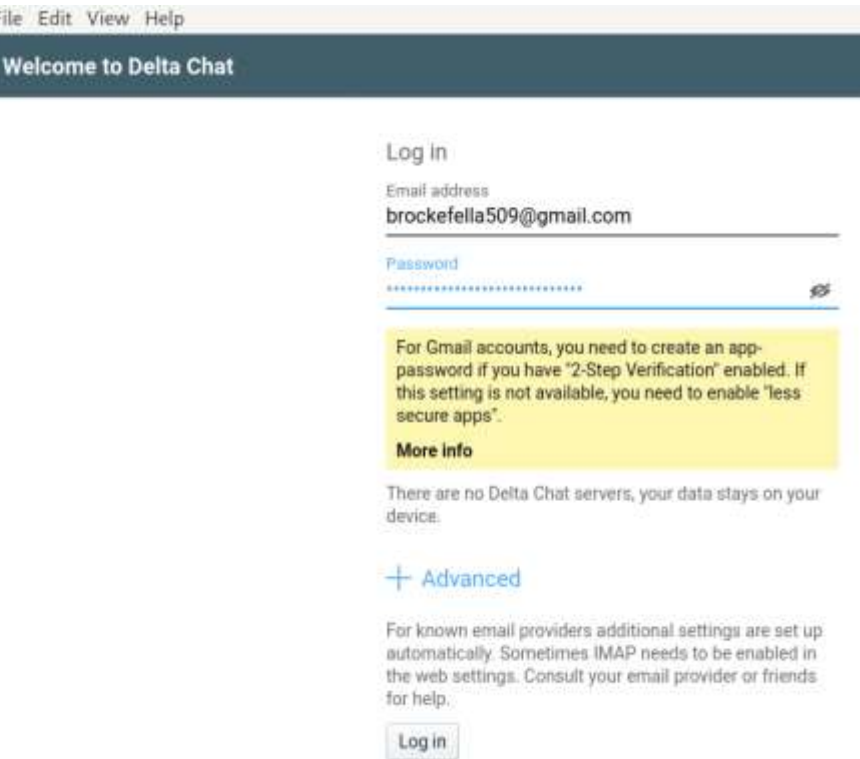
Affected Locations

- **deltachat-core-rust/src/securejoin.rs**
- **deltachat-core-rust/src/pgp.rs**
- **deltachat-core-rust/src/dc_receive_imf.rs**
- **deltachat-core-rust/src/peerstate.rs**

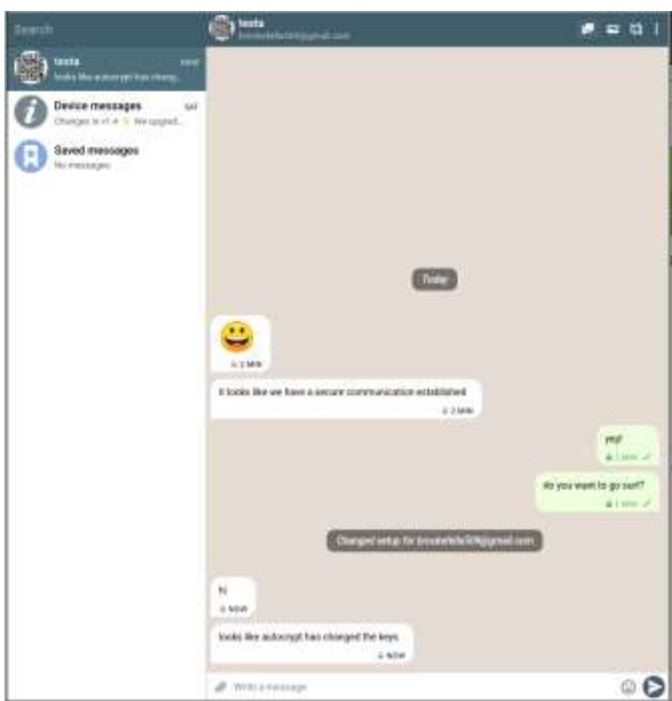
Steps to Reproduce

1. Download and set up DeltaChat with **user_a**.
2. Download and setup DeltaChat with **user_b**.
3. Download DeltaChat for **user_c**.
4. Exchange public key material from **user_a** to **user_b** with Autocrypt or QR Codes.
5. Send chat messages from **user_a** to **user_b** and **user_b** to **user_a**.
6. Authenticate to DeltaChat with **user_a's** credentials on **user_c's** machine.
7. Send a message from **user_b** to **user_a**.
8. Notice that the message cannot be read yet from **user_c's** machine and that new key material is being established.
9. Wait a small amount of time.
10. Send a message from **user_c's** machine to **user_b**.
11. Notice that the message **Changed setup for [email_address_of_user_a]** is displayed.
12. Notice that the following messages are decrypted.

The following is an example of a simulated malicious user authenticating into a targeted user's account via DeltaChat after a verified encrypted PGP communication has been established:

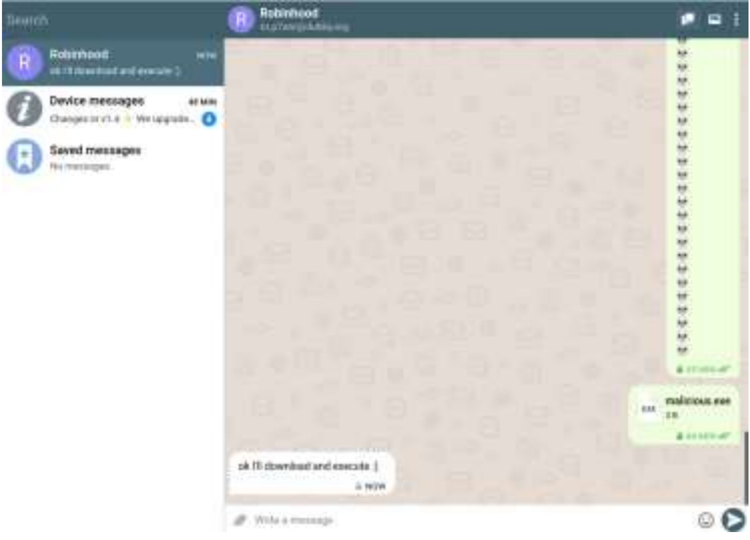


The following screenshot shows the targeted user's screen:



Notice that a message displays that the setup has changed.

The following screenshot displays the DeltaChat client from the attacker's perspective after a few more messages have been sent and a malicious file has been sent to the targeted user.



Recommended Remediation:

The assessment team recommends changing the **Changed setup for..** message to messaging that better conveys the potential disruption of expected confidentiality. Alternatively, different colors or images could be used to represent the different levels of security. Additionally, a popup could be displayed to the user to accept new communications for the new encryption keys similar to the message that is delivered upon initial communication.

It is the understanding of the assessment team that the security levels of different messages would be ordered in the following ways:

1. Manually verified PGP keys via QR Codes + TLS
2. Autocrypt + TLS
3. Manually verified PGP keys via QR Codes
4. Autocrypt
5. No PGP + TLS
6. No PGP

It is important that when a security level (as stated above by the assessment team) is decreased or changed that the information is effectively communicated and presented to the end-user.

References:

Autocrypt
DeltaChat Specification

INFORMATIONAL FINDINGS

I1: Potential Filesystem Path Traversal Sequence Which Would Allow Arbitrary File Write

Description:

The **DeltaChat** application triggers a file system write operation with user-controllable data which could potentially leave the application vulnerable to path traversal attacks. In vulnerable systems, an attacker could provide the path traversal sequence (i.e., dot-dot-slash, **../**) as part of their input to move outside the expected target directory and cause file writes elsewhere on the file system. This could allow the attacker to overwrite arbitrary files on the desktop version of DeltaChat and gain remote code execution.

Note: The DeltaChat application does protect against **/** and **** characters by normalizing the filenames. **This added protection prevented the assessment team from overwriting critical operating system related files.** However, the assessment team is including this information to provide additional defense-in-depth informational findings to help further improve DeltaChat security.

Affected Locations

- **deltachat-core-rust/src/mimeparser.rs**
- **deltachat-core-rust/src/blob.rs**

The following source code located in **deltachat-core-rust/src/mimeparser.rs** displays that the code looks for KML files, and if the file contains a location in the name or message in the name, it automatically parses it. It then writes the file.

```
698         if decoded_data.is_empty() {
699             return;
700         }
701         // treat location/message kml file attachments specially
702         if filename.ends_with(".kml") {
703             // XXX what if somebody sends eg an "location-highlights.kml"
704             // attachment unrelated to location streaming?
705             if filename.starts_with("location") || filename.starts_with("message") {
706                 let parsed = location::Kml::parse(context, decoded_data)
707                     .map_err(|err| {
708                         warn!(context, "failed to parse kml part: {}", err);
709                     })
710                     .ok();
711                 if filename.starts_with("location") {
712                     self.location_kml = parsed;
713                 } else {
714                     self.message_kml = parsed;
715                 }
716                 return;
717             }
718         }
719         /* we have a regular file attachment,
720         write decoded data to new blob object */
721
722         let blob = match BlobObject::create(context, filename, decoded_data).await {
```

```
723             Ok(blob) => blob,
724             Err(err) => {
725                 error!(
726                     context,
727                     "Could not add blob for mime part {}, error {}", filename, err
728                 );
729                 return;
730             }
731         };
732         info!(context, "added blobfile: {:?}", blob.as_name());
```

The **BlobObject create()** function resides in the **deltachat-core-rust/src/blob.rs** file. It performs a sanitization and then creates a new file.

```
51      pub async fn create(
52          context: &'a Context,
53          suggested_name: impl AsRef<str>,
54          data: &[u8],
55      ) -> std::result::Result<BlobObject<'a>, BlobError> {
56          let blobdir = context.get_blobdir();
57          let (stem, ext) = BlobObject::sanitise_name(suggested_name.as_ref());
58          let (name, mut file) = BlobObject::create_new_file(&blobdir, &stem, &ext).await?;
59          file.write_all(data)
60              .await
61              .map_err(|err| BlobError::WriteFailure {
62                  blobdir: blobdir.to_path_buf(),
63                  blobname: name.clone(),
64                  cause: err.into(),
65              })?;
66          let blob = BlobObject {
67              blobdir,
68              name: format!("$BLOBDIR/{}", name),
69          };
70          context.emit_event(Event::NewBlobFile(blob.as_name().to_string()));
71          Ok(blob)
72      }
73
74      // Creates a new file, returning a tuple of the name and the handle.
75      async fn create_new_file(
76          dir: &Path,
77          stem: &str,
78          ext: &str,
79      ) -> Result<(String, fs::File), BlobError> {
80          let max_attempt = 15;
81          let mut name = format!("{}{}", stem, ext);
82          for attempt in 0..max_attempt {
83              let path = dir.join(&name);
84              match fs::OpenOptions::new()
85                  .create_new(true)
86                  .write(true)
87                  .open(&path)
88                  .await
89              {
90                  Ok(file) => return Ok((name, file)),
91                  Err(err) => {
92                      if attempt == max_attempt {
93                          return Err(BlobError::CreateFailure {
94                              blobdir: dir.to_path_buf(),
95                              blobname: name,
96                              cause: err,
97                          });
98                      } else {
```

The **sanitise_name()** function performs the following checks:

```
309     fn sanitise_name(name: &str) -> (String, String) {
310         let mut name = name.to_string();
311         for part in name.rsplit('/') {
312             if !part.is_empty() {
313                 name = part.to_string();
314                 break;
315             }
316         }
317         for part in name.rsplit('\\') {
318             if !part.is_empty() {
319                 name = part.to_string();
320                 break;
321             }
322         }
323         let opts = sanitize_filename::Options {
324             truncate: true,
325             windows: true,
326             replacement: "",
327         };
328
329         let clean = sanitize_filename::sanitize_with_options(name, opts);
```

An additional check in **blob.rs** for some file operations can be found in the following code snippet:

```
345     fn is_acceptible_blob_name(name: impl AsRef<OsStr>) -> bool {
346         let uname = match name.as_ref().to_str() {
347             Some(name) => name,
348             None => return false,
349         };
350         if uname.find('/').is_some() {
351             return false;
352         }
353         if uname.find('\\').is_some() {
354             return false;
355         }
356         if uname.find('\0').is_some() {
357             return false;
358         }
359         true
360     }
```

If there was a bypass in the sanitization or validation of blob names, then an attacker could achieve remote code execution. While the assessment team was not able to bypass the filters in the timebox of this engagement. It was noticed that UTF-8 characters were allowed in filenames and a file that started with **..** wrote a file to the filesystem that was a random negative number.

Recommended Remediation:

The assessment team recommends creating a whitelist of acceptable characters and not allowing or converting UTF-8 characters in filenames. SELinux or Apparmor policies could be created to prevent writes to locations outside of the application directory. In addition, seccomp or sandbox filters could be leveraged to prevent the application from writing outside of its appropriate directory.

References:

OWASP File System Page
Rust syzcallz-rs

I2: Additional Security Considerations (Quantum Computing, Traffic Flow Confidentiality, and DeltaChat Attack Surface Reduction)

Description:

DeltaChat relies on public key exchanges and symmetric encryption of messages to provide users end-to-end encryption. This security is provided by OpenPGP. While there are no known practical cryptanalysis attacks that fully break public key algorithms like RSA2048, there could be potential attacks in the future (e.g. quantum computing). Attacks on cryptographic algorithms usually only get better. There are some open defense strategies proposed and implemented to defend against quantum computing attacks. Some of these defenses are documented within the source code and published papers of the WireGuard VPN product and protocol.

Traffic Flow Confidentiality (TFC) is a feature that can provide additional protection against statistical analysis of message metadata (e.g. size of message, time sent, destination, source, etc.). For example, researchers have presented statistical attacks against TLS and Google Maps that showed that they could predict the geolocation of where someone was searching based on message size, IP destination, IP source and message timing with relative decent accuracy. Random data or padding could be introduced to a message to help mitigate against this type of threat. For example, random spaces could be added to the beginning or end of a cleartext message to make it more difficult for an attacker to perform statistical analysis on the encrypted message.

Recommended Remediation:

The assessment team suggests researching and implementing protections against quantum computing and implementing TFC.

The assessment team also suggests adding an option or a set of options to decrease the attack surface of the DeltaChat application. For example, options to turn off file transfers, turn off group messaging, disable parsing of KML files, only allow manual QR PGP key exchanges and drop all other messages could be examples of options to provide users with reduced attack surface. To reduce that attack surface unsafe Rust code could be removed or rewritten.

References:

WireGuard – Optional Pre-shared Symmetric Key Mode
Traffic Flow Confidentiality in IPSec Protocol and Implementation