REPORT

Penetration test of Vexl mobile application

Trezor Company s.r.o

Report: 20220816001 / 16.08.2022

CITADELO
HACKERS ON YOUR SIDE

Revision record			
Version	Date	Author	Comment
1.0	16.08.2022	František Uhrecký	Initial draft
1.1	16.08.2022	Marek Tamaškovič	Document review

Table of contents

Table of images

1. EXECUTIVE SUMMARY

During the penetration test, we identified several medium-risk vulnerabilities which lead to bypassing authorization schema, data leakage in specific scenarios. Further we identified several issues during manual code audit.

The application tries to keep user anonymous and uses heavily public key cryptography. We identified several issues, where user identity can leak using side channels. For example user identity can be mapped to his public key, because same public is used for chat communication.

The authorization implementation lacks proper user checks. In specific cases it is possible to bypass authorization, e.g., it is possible to impersonate user in chat. But it is required to know his public and offer public. Both keys can be get by initiating communication as described in the finding.

We performed code audit as well, during which we identified weaker verification code algorithm. The problem is the process of generation a 6 digit random code, where the first digit is fixed to 0. Therefore, the space is lowered only to 5 digits.

Also we identified multiple hardcoded secrets within the source code. They were identified in both API and mobile application sourced. Using these credentials it is possible to decrypt sensitive data, or access publicly exposed DB.

This mean that there is lack of automated detection of the secrets in code. We recommend modifying your commit hooks and CI/CD pipeline to prevent this kind of data leaks.

We recommend fixing at least medium, high and critical risk issues before a production release or as soon as possible. Although some of the misconfiguration or issues cannot be easily misused by an attacker we still recommend fixing them to conform to defense-in-depth policy.

During the penetration test we encountered multiple application functionality issues. The application is in development stage and quite a lot of functionality. But during the pentest we contacted client and we arranged online meeting.

During the penetration test, we encountered the following problems, application outages, which complicated overall penetration testing:

Not implemented (Android):

- FAQ
- Terms and Privacy
- Set PIN / FaceID
- Private settings
- Contacts Imported
- Import facebook

Not implemented (iOS):

- FAQ
- Terms and Privacy
- allow screenshots
- Set PIN / FaceID

- Private settings
- Import facebook
- Request known data
- Unable to change offer location

1.1. Summary table

Section	Test ID	Test name	Risk level
4.1.1	M1	User identity side channel data leakage (API)	Medium
4.1.2	M1	Exposed DB service to public (code)	Medium
4.6	M6	Insecure Authorization	Medium
4.7.1	M7	Code verification invalid stream seed (code)	Medium
4.7.2	M7	Hardcoded secrets (Android, iOS, API)	Medium
4.1.3	M1	Phone verification SMS	Low
4.2	M2	Insecure Data Storage	Low
4.4	M4	Insecure Authentication	Low
4.7.3	M7	Account delete - storage not cleared (Android)	Low
4.1.4	M1	Offer side channel data leakage (API)	Note
4.3.1	M3	Insecure communication ATS (iOS)	Note
4.3.2	M3	SSL pinning (Android, iOS)	Note
4.7.4	M7	File path manipulation (code, API)	Note
4.8	M8	Code Tampering	Note
4.9	M9	Reverse Engineering	Note
4.10	M10	Extraneous Functionality	Note

2. ASSESSMENT SCOPE AND DETAILS

Our audit is based on methodologies from OWASP Mobile Security Project (https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10, TOP 10 Mobile Risks). This methodology is primarily targeted for mobile applications. This test is focused on ten most frequent mobile security risks as described by OWASP Mobile Top 10 2016 methodology. Our test consists of every applicable step from Top 10 Mobile Risks. No denial of service tests have been performed.

2.1. Goals

Our primary goal was to find all possible attack vectors applicable to the mobile application and value-associated risk. This report is intended primarily to two target groups:

- Security officer and security department for security management and risk management purposes
- For developers to mitigate discovered risks and make application more secure to discovered threats

2.2. Target system

The target of penetration tests was mobile application Vexl. The application is community based crypto exchange it does not process any payments. Following APK and IPA files were provided for testing:

- vexl.apk - com.cleevio.vexl.staging 1.1.0-stage (636) - 9bf0a1d808902fbae4a303b27707fd08d03fcad8b0d742338e9c65b2e791682c
- vexl.ipa - vexl (com.cleevio.vexl.appstore) 1.1.0 (117) - 69053dc7df06cc72fd2874bf8d4c29ecae51d60e341f9828ba586773e667afec

In the scope of the penetration test was also code audit of following source code repositories (https://gitlab.cleevio.cz/):

- backend / vexl / vexl-chat-ms
- backend / vexl / vexl-contact-ms
- backend / vexl / vexl-offer-ms
- backend / vexl / vexl-user-ms
- clients / vexl-android
- clients / vexl-cryptography
- clients / vexl-ios

Following services were out of scope:

- https://contact.vexl.staging.cleevio.io/api-docs/swagger-ui/index.html?configUrl=%2Fapi-docs%2Fswagger-config#/Group

Penetration test was performed from following IP addresses:

- 91.210.181.37 (vpn.citadelo.com)

All tests have been performed remote during the period from 1st August to 11th August 2022.

3. RISK RATING METHODOLOGY

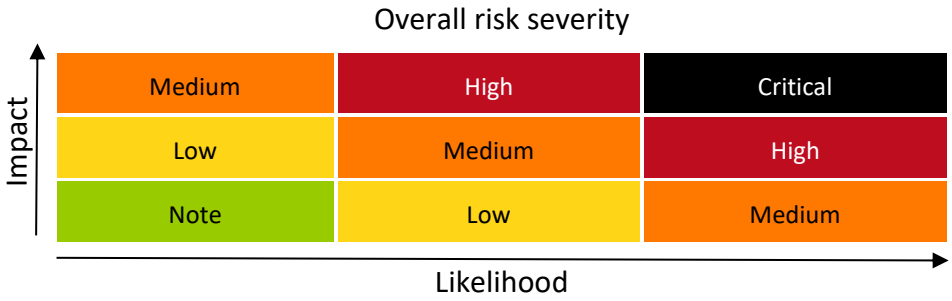
There are many different approaches to risk analysis. Our approach is based on OWASP Risk Rating Methodology, which is based on standard risk model:

*Risk = Likelihood * Impact*

We have considered factors for both likelihood and impact and rated them from 0 to 9. Following table describes severity for the respective score.

Likelihood and Impact Levels	
6 to 9	High
3 to <6	Medium
0 to <3	Low

And from that we have final severity rating for the risk:



4. TEST RESULTS

Following is a list of results of tests in respective categories of OWASP Top 10 Mobile Risks 2016 and risk evaluation for every finding.

4.1. Improper Platform Usage

4.1.1. User identity side channel data leakage (API)

Risk level	Impact	Likelihood
Medium	Medium	Medium

Finding

Vulnerable. We identified that it is possible to determine user identity based on his public key. Users can communicate using chat service. For this communication there are public key in use:

- `senderPublicKey`
- `receiverPublicKey`

Usually these keys are equivalents to:

- user's public key
- offer public key

The problem is, that if the user which reacts to offer and once reveals his identity, it can be bound to his public key. Because the same public key is used for chat communication every time (when he reacts to public offers). Although in new offer chat, the application does not show his identity without approval. But it is possible to pair his public key with previous chat, where he revealed his identity.

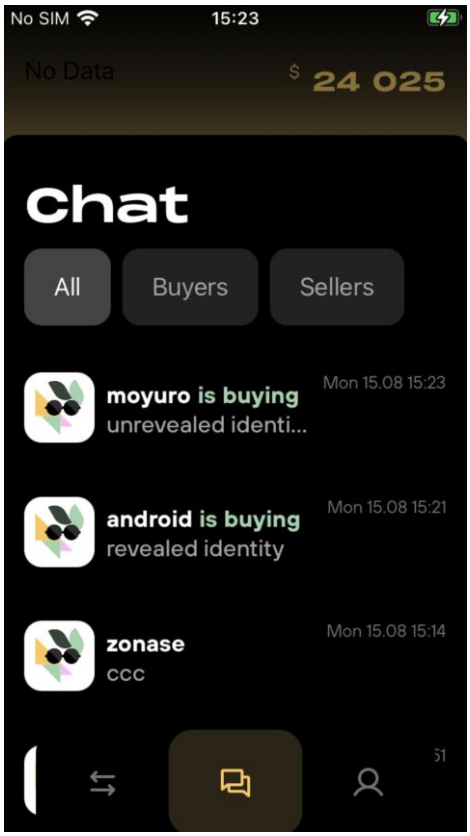


Image 1 Identity side channel data leakage

Users `moyuro` and `android` are the same user.

Responses from `PUT /api/v1/inboxes/messages` - both sender public key are the same.

Message from `android` chat (revealed identity):

```
{"messages":[{"message":"---SNIP---
","senderPublicKey":"LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJe
mowQ0FRWUZLNEVFQUNFRE9nQUVOU0RQSHNKdEhNVTF0dGtNRXd2RWNHVTd4VWErcXZJUQpONnJ
UaExOOTNrUjI4NTBoNXBvYWxVL3lEcU94QVVjQlR1clA5bXh5cE9BPQotLS0tLUVORCBQVUJMS
UMgS0VZLS0tLS0K","messageType":"MESSAGE"}]}
```

Message from `moyuro` chat:

```
{"messages":[{"message":"---SNIP---
","senderPublicKey":"LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJe
mowQ0FRWUZLNEVFQUNFRE9nQUVOU0RQSHNKdEhNVTF0dGtNRXd2RWNHVTd4VWErcXZJUQpONnJ
UaExOOTNrUjI4NTBoNXBvYWxVL3lEcU94QVVjQlR1clA5bXh5cE9BPQotLS0tLUVORCBQVUJMS
UMgS0VZLS0tLS0K","messageType":"MESSAGE"}]}
```

Solution

We recommend using ephemeral keypair for every new chat. This would achieve user's anonymity.

4.1.2. Exposed DB service to public (code)

Risk level	Impact	Likelihood
Medium	High	Low

Finding

Vulnerable. During the code audit we checked for hardcoded secrets. We identified publicly exposed DB service.

We were able successfully connect to PostgreSQL from the internet:

```
Nmap scan report for 46.101.113.19

PORT     STATE SERVICE
5432/tcp open  postgresql

[+] 46.101.113.19:5432 - Login Successful: vexl-contact-ms-api-
staging:<redacted>@vexl-contact-ms-api-staging
```

These kind of services should be not exposed to the internet. They are storage services which should be access only from internal network, by middleware.

Solution

Do not expose unnecessary services to the public. Restrict access by firewall settings and allow access only from whitelisted IP addresses/ranges.

4.1.3. Phone verification SMS

Risk level	Impact	Likelihood
Low	Medium	Low

Finding

Vulnerable. It is possible to request phone verification code without waiting the 30 seconds timeout. The time throttling mechanism is only implemented client side in the application UI.

Request:

```
POST /api/v1/user/confirmation/phone HTTP/2
Host: user.vexl.staging.cleevio.io
Accept: application/json
Content-Type: application/json
Accept-Encoding: gzip, deflate
X-Platform: ios
X-Install-Uuid: 1D6AEB12-5EB1-54BF-BD84-B09B13B2333E
User-Agent: vexl/117 CFNetwork/1335.0.3 Darwin/21.6.0
Accept-Language: en-GB,en;q=0.9
Content-Length: 34

{"phoneNumber":"+421 910 865 313"}
```

Response:

```
HTTP/2 200 OK
Date: Tue, 02 Aug 2022 07:59:52 GMT
---SNIP---

{"verificationId":101,"expirationAt":"2022-08-02T08:00:22.787+0000"}
```

Solution

Implement proper server throttling mechanism. Limit calling the SMS code API for the same number. It is possible to implement CAPTCHA mechanism if malicious activity was detected or enforce it by default.

4.1.4. Offer side channel data leakage (API)

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. The application encrypts offer data by user's (contact) public key. Almost every offer option is encrypted, for example:

- `active`
- `activePriceState`
- `activePriceValue`
- `amountBottomLimit`
- `amountTopLimit`
- `btcNetwork`
- `commonFriends`
- `currency`
- `feeAmount`
- `feeState`
- `friendLevel`
- `groupUuid`
- `location`
- `locationState`
- `offerDescription`
- `offerPublicKey`
- `offerType`
- `paymentMethod`
- `userPublicKey`

The problem is, that for example if there is more options like `paymentMethod`, there will be every method encrypted standalone. It means, that it reveals some kind of information to the observer.

Solution

We recommend encrypting data in a form, where no data size/count would leak any information. For example encrypt all payment options in a one payload.

4.2. Insecure Data Storage

Risk level	Impact	Likelihood
Low	Medium	Low

Finding

Vulnerable. **Android**

Sqlite database Android - `databases/CleevioDatabase.db` store multiple data including user messages. These messages are not encrypted.

```
INSERT INTO ChatMessageEntity VALUES(1,'EC0E80E8-5378-4E0B-853F-
480E33A9F03F','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
RWUZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNT
nVEM3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0FRWU
ZLNEVFQUFvRFFnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZvNHJp
ZGUvTGlyTVkyMGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLUVORC
BQVUJMSUMgS0VZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb
1pJemowQ0FRWUZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo
5RnhPL2NNTnVEM3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQV
UJMSUMgS0VZLS0tLS0K','Sell22',NULL,'REQUEST_MESSAGING',1660139033480,NULL,
NULL,0,0);
INSERT INTO ChatMessageEntity VALUES(2,'a7803140-a212-44b7-a190-
9fc2b5a07741','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
RWUZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNT
nVEM3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0FRWU
ZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNTnVE
M3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0VZLS
0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0FRWUZLN
EVFQUFvRFFnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZvNHJpZGU
vTGlyTVkyMGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLUVORCBQV
UJMSUMgS0VZLS0tLS0K','The conversation has
started',NULL,'APPROVE_MESSAGING',1660139293750,NULL,NULL,1,0);
INSERT INTO ChatMessageEntity VALUES(3,'bbbeaf19-7fa2-4777-a4a1-
01e81f30d01b','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
RWUZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNT
nVEM3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0FRWU
ZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNTnVE
M3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0VZLS
0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0FRWUZLN
EVFQUFvRFFnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZvNHJpZGU
vTGlyTVkyMGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLUVORCBQV
UJMSUMgS0VZLS0tLS0K','The conversation has
started',NULL,'APPROVE_MESSAGING',1660139835941,NULL,NULL,1,0);
INSERT INTO ChatMessageEntity VALUES(4,'ca4fd07c-7877-420c-9a50-
1f52ebaa0036','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
```

```
RWUZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNT
nVEM3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0FRWU
ZLNEVFQUNFRE9nQUU5V3BmS2dLSEZuRWNEL09waGVKdSthZWFQeFlUR0xNNgo5RnhPL2NNTnVE
M3piY0R6WHBzWDFKTmJBamV6WVRGZi9iV2lyL05aSDVRPQotLS0tLUVORCBQVUJMSUMgS0VZLS
0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0FRWUZLN
EVFQUFvRFFnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZvNHJpZGU
vTGlyTVkyMGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLUVORCBQV
UJMSUMgS0VZLS0tLS0K','The conversation has
started',NULL,'APPROVE_MESSAGING',1660139981338,NULL,NULL,1,0);
INSERT INTO ChatMessageEntity VALUES(5,'BCD95D4A-93A3-4470-8A00-
41AF2657842B','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
RWUZLNEVFQUNFRE9nQUU1YUc4OGpHOUVYS29vaWxCSEVCY2hrYUhsZWR2b283MApRR2xhcUp6b
GovYm12dXNrZ3dGWm00Q01oOFRXTFJwczg3UFJzOXJlRHhZPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K3EE=','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0
FRWUZLNEVFQUFvRFFnQUUwRGZzd0JHcHpSa2JaYzNxMVVhaU5nUXU4bkZIT1VNegpMSXNxV2FX
MGl3S2dFNHdMV3dHb1VjVnREUWF0bmx4NTdKYjAweEF6V1M1ZHM2MEE0ZHZocnc9PQotLS0tLU
VORCBQVUJMSUMgS0VZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBW
UhLb1pJemowQ0FRWUZLNEVFQUNFRE9nQUU1YUc4OGpHOUVYS29vaWxCSEVCY2hrYUhsZWR2b28
3MApRR2xhcUp6bGovYm12dXNrZ3dGWm00Q01oOFRXTFJwczg3UFJzOXJlRHhZPQotLS0tLUVOR
CBQVUJMSUMgS0VZLS0tLS0K3EE=','111',NULL,'REQUEST_MESSAGING',1660140056107,
NULL,NULL,0,0);
INSERT INTO ChatMessageEntity VALUES(6,'D05150BA-809F-4951-9568-
57760E5DA18A','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0F
RWUZLNEVFQUNFRE9nQUU1YUc4OGpHOUVYS29vaWxCSEVCY2hrYUhsZWR2b283MApRR2xhcUp6b
GovYm12dXNrZ3dGWm00Q01oOFRXTFJwczg3UFJzOXJlRHhZPQotLS0tLUVORCBQVUJMSUMgS0V
ZLS0tLS0K3EE=','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0
FRWUZLNEVFQUFvRFFnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZv
NHJpZGUvTGlyTVkyMGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLU
VORCBQVUJMSUMgS0VZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBW
UhLb1pJemowQ0FRWUZLNEVFQUNFRE9nQUU1YUc4OGpHOUVYS29vaWxCSEVCY2hrYUhsZWR2b28
3MApRR2xhcUp6bGovYm12dXNrZ3dGWm00Q01oOFRXTFJwczg3UFJzOXJlRHhZPQotLS0tLUVOR
CBQVUJMSUMgS0VZLS0tLS0K3EE=','111',NULL,'REQUEST_MESSAGING',1660140214725,
NULL,NULL,0,0);
INSERT INTO ChatMessageEntity VALUES(7,'ef3ea1c9-2f1d-4cce-9bd6-
a995e0f48792','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0F
RWUZLNEVFQUFvRFFnQUVIK1AxWm5pRFF3NUxQVXlxUkZUL0gxbHgrRUg0RENpaAp5amdHN284c
VVCY0Jra213bEo4NVhUd1ZZNTR3RTlDbmZTTlg5K0lkTXl1ZkJ6TlMvOTA0UGc9PQotLS0tLUV
ORCBQVUJMSUMgS0VZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWU
hLb1pJemowQ0FRWUZLNEVFQUNFRE9nQUUrZUJQa3Q0b1oyc3NvS005ZnpNbHluSVdrWEQ1WUtU
QQowaG54UW5kSWptSjRlT2pLRWh6NjBnT2hVVDNyVHpzUFF4MFQwSk9CbUNNPQotLS0tLUVORC
BQVUJMSUMgS0VZLS0tLS0K','LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb
1pJemowQ0FRWUZLNEVFQUFvRFFnQUVIK1AxWm5pRFF3NUxQVXlxUkZUL0gxbHgrRUg0RENpaAp
5amdHN284cVVCY0Jra213bEo4NVhUd1ZZNTR3RTlDbmZTTlg5K0lkTXl1ZkJ6TlMvOTA0UGc9P
QotLS0tLUVORCBQVUJMSUMgS0VZLS0tLS0K','dsadsa',NULL,'REQUEST_MESSAGING',166
0140645395,NULL,NULL,1,0);
```

iOS

Local EC keypair is stored in sqlite database: `Library/Application\ Support/VexlDataModel.sqlite`

This database contains a lot of tables, but interesting to attackers are:

- `ZMANAGEDKEYPAIR` - contains encrypted private key
- `ZMANAGEDPROFILE` - contains user profile detail and reference to encrypted private key
- `ZMANAGEDUSER` - contains encrypted signature
- `ZMANAGEDMESSAGE` - contains not encrypted chat messages

Encryption is done by following function calls:

```
./vexl-ios-devel/vexl/Database/ManagedObjects/ManagedKeyPair+.swift:14:
set { encryptedPrivateKey = try? newValue?.aes.encrypt(password:
Constants.localEncryptionPassowrd) }
./vexl-ios-devel/vexl/Database/ManagedObjects/ManagedUser+.swift:13:
set { encryptedSignature = try? newValue?.aes.encrypt(password:
Constants.localEncryptionPassowrd) }
```

Encryption key is hardcoded as described in - "Hardcoded secrets (Android, iOS)".

Solution

iOS provides a mechanism for working with encryption keys, referred to as Secure Enclave. Similarly to Android, any interactions with these keys are only possible via programming interfaces - not by reading files directly.

You can also generate keys and store them in the Keychain yourself.

If iOS apps use keys stored in Keychain, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The secure enclave provides functionality, where signing can be performed in this secured environment. It means that private key never leaves secure enclave processor. For more information see: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave.

Android offers an interface known as Keystore. Apps can only obtain encryption keys via the programming interface, not by reading files directly. The files containing the encryption keys are themselves stored in `/data/misc/keystore/` and belong to the keystore user.

If Android apps encrypt all sensitive data using keys from Keystore, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The same can be applied for singing.

4.3. Insecure Communication

4.3.1. Insecure communication ATS (iOS)

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. We have identified insecure configuration for iOS Application Transport Security (ATS - https://developer.apple.com/documentation/bundleresources/information_property_list/nsapptransportsecurity). This allows developer to configure restrictions about application connections.

```
<key>NSAppTransportSecurity</key>
<dict>
<key>NSAllowsArbitraryLoadsInWebContent</key>
<true/>
</dict>
```

Solution

We recommend to properly configure ATS settings. We recommend to keep default iOS settings which follows current best practices. In general for production environment no exceptions should be defined.

For more information: https://github.com/OWASP/owasp-mstg/blob/master/Document/0x06g-Testing-Network-Communication.md

4.3.2. SSL pinning (Android, iOS)

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. Both applications does not implement SSL pinning as a prevention against man in the middle attackers.

Solution

Pro production release implement SSL pinning.

4.4. Insecure Authentication

Risk level	Impact	Likelihood
Low	Medium	Low

Finding

Vulnerable. The application uses signature authentication schema. For example, while approving message request https://chat.vexl.staging.cleevio.io/api/v1/inboxes/approval/confirm the request body contains `publicKey`, `signature`, `publicKeyToConfirm`, `message` and `approve`. The interesting are `publicKey`, `signature`, `publicKeyToConfirm`. The `publicKey` is public key of the inbox and the `signature` is signed challenge by this keypair. Let's consider scenario, where this request is leaked. Malicious user can modify this request, exchange the value `publicKeyToConfirm` and send it again (while challenge signature is valid - 30 minutes window). This would mean that the attacker could be able to send message to this inbox.

This issue just expose very low security impact on the application, but if the whole request would be signed, this would not be possible.

Signing the whole request could achieve authenticity of the content.

The second problem of the currently used scheme is that it never expires. Following HTTP headers sent to the server:

```
Signature:
MD0CHEIa9/BJbSiqzWkgpIvnMjVlfFO66om85AMvTN8CHQCvy+lgPjR3qNi0NiLWpr/IYk1DVb
ZJ8v/GkBdhAA==
Hash: Et8z4iD3VtagMmQzXfGHsML04975hnryoW5vsBb2l6M=
Public-Key:
LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0FRWUZLNEVFQUNFRE
9nQUV6Ynp1QzVCWTYwK0tpK1NIcWJVcTAyU3NGVVZxMW5HVwp1cWo1YXY2em9KVFEzWU9HMG12
dG9NL28wd3R0bzJ0Zm5ldWFRNVMrdFFBPQotLS0tLUVORCBQVUJMSUMgS0VZLS0tLS0K
```

Where hash is the user's phone HMAC.

We would recommend to include a timestamp - expiration time. For example it could be like server time + 30 minutes.

Solution

Sign whole request including request body, get parameters and suer specific headers. To implement session timeout include expiration time in signature. Expiration time could be send e.g. to server time + 30 minutes.

iOS provides a mechanism for working with encryption keys, referred to as Secure Enclave. Similarly to Android, any interactions with these keys are only possible via programming interfaces - not by reading files directly.

You can also generate keys and store them in the Keychain yourself.

If iOS apps use keys stored in Keychain, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The secure enclave provides functionality, where signing can be performed in this secured environment. It means that private key never leaves secure enclave processor. For more information see: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave.

Android offers an interface known as Keystore. Apps can only obtain encryption keys via the programming interface, not by reading files directly. The files containing the encryption keys are themselves stored in `/data/misc/keystore/` and belong to the keystore user.

If Android apps encrypt all sensitive data using keys from Keystore, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The same can be applied for singing.

4.5. Insufficient Cryptography

Finding

Not vulnerable. We have not identified any cryptography issues other than mentioned in different sections.

4.6. Insecure Authorization

Risk level	Impact	Likelihood
Medium	High	Low

Finding

Vulnerable. We identified multiple issues within the user authorisation. In general the problem is following. The application uses signature based user authentication. User's HMAC of phone number is signed by his private key. This signature is user for user authentication.

HTTP headers sent to the server:

```
Signature:
MD0CHEIa9/BJbSiqzWkgpIvnMjVlfFO66om85AMvTN8CHQCvy+lgPjR3qNi0NiLWpr/IYk1DVb
ZJ8v/GkBdhAA==
Hash: Et8z4iD3VtagMmQzXfGHsML04975hnryoW5vsBb2l6M=
Public-Key:
LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUU0d0VBWUhLb1pJemowQ0FRWUZLNEVFQUNFRE
```

```
9nQUV6Ynp1QzVCWTYwK0tpK1NIcWJVcTAyU3NGVVZxMW5HVwp1cWo1YXY2em9KVFEzWU9HMG12
dG9NL28wd3R0bzJ0Zm5ldWFRNVMrdFFBPQotLS0tLUVORCBQVUJMSUMgS0VZLS0tLS0K
```

In general most of the services just check whether the signature is valid. There is no check whether the user is authorised to perform target action.
The issue is that the endpoint does not check authenticity of the user. Bellow are specific code snippets which are responsible for security checks.

But the probability of the misuse is not so critical. Because another key parameter is usually required - public of the target object. Public is meant to be public, but there is no functionality where you could list all public keys - related to offers, users or chats.

Chat service

Chat service uses following security filter. In general it is possible to send messages behalf the other users, but you need to know their public key.

Security filter checks only if the signature is valid.

File `vexl-chat-ms-master/src/main/java/com/cleevio/vexl/common/security/filter/SecurityFilter.java`:

```
try {
if (signatureService.isSignatureValid(publicKey, hash, signature)) {
AuthenticationHolder authenticationHolder;

authenticationHolder = new AuthenticationHolder(null, null);
authenticationHolder.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticationHolder)
;
} else {
SecurityContextHolder.clearContext();
}
} catch (Exception e) {
SecurityContextHolder.clearContext();
handleError(response, "Signature verification failed: " + e.getMessage(),
400);
}
```

Api endpoints: `/api/v1/inboxes`

- `PUT` - `updateInbox` - any user is able to update inbox firebase token by inbox public key, at least the person on the otherside of the communication know the public key
- `POST` - `createInbox` - user does not need to be owner of the inbox, not a security issue
- `POST /messages` - `sendMessage` - in case of public knowledge, any user is able to send messages

- `DELETE /messages - deletePulledMessages` - with knowledge of inbox public key, any user is able to delete pulled messages
- `DELETE - deleteInbox` - with knowledge of inbox public key, any user is able to delete all messages

For example with cooperation of "User anonymity side channel leakage (API)" issue, where the same public is used for chat communication, it can be possible to impersonate the user in chat. Lets imagine following setup:

- we have 3 users, A, B and M
- A creates offer
- M creates offer
- B contacts both A and M
- M contacts A
- using this vulnerability, M is able to send message behalf the user B to A, because, he has his public key and public key of the A offer

User service

Security filter checks for signature validity and if user's public key is in DB.

File `vexl-user-ms-master/src/main/java/com/cleevio/vexl/common/security/filter/SecurityFilter.java`:

```
try {
if (signatureService.isSignatureValid(publicKey, phoneHash, signature)) {
AuthenticationHolder authentication = userService
.findByPublicKey(publicKey)
.map(user -> {
AuthenticationHolder authenticationHolder = new
AuthenticationHolder(user);
authenticationHolder.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

return authenticationHolder;
})
.orElse(null);

SecurityContextHolder.getContext().setAuthentication(authentication);
} else {
SecurityContextHolder.clearContext();
}
} catch (Exception e) {
SecurityContextHolder.clearContext();
handleError(response, "Signature verification failed: " +
e.getMessage());
}
```

Api endpoints: `/api/v1/users`

- `PUT - updateFirebaseToken` - any user is able to update user's firebase token by his public key and hash, at least the person on the otherside of the communication know the public key, hash value, exactly HMAC, the phone number/facebook ID can be guessed. It is possible because of the service select user by its public key and hash.

File `vexl-contact-ms-master/src/main/java/com/cleevio/vexl/module/user/service/UserService.java`:

```
@Transactional
public void updateFirebaseToken(final String publicKey, final String
hash, @Valid final FirebaseTokenUpdateRequest request) {
final User user =
this.userRepository.findUserByPublicKeyAndHash(publicKey, hash)
.orElseThrow(UserNotFoundException::new);
user.setFirebaseToken(request.firebaseToken());
}
```

Offer service

Security filter checks only if the signature is valid.

File `vexl-offer-ms-master/src/main/java/com/cleevio/vexl/common/security/filter/SecurityFilter.java`:

```
try {
if (signatureService.isSignatureValid(new
CheckSignatureValidityQuery(publicKey, hash, signature))) {
AuthenticationHolder authenticationHolder;

authenticationHolder = new AuthenticationHolder(null, null);
authenticationHolder.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticationHolder)
;
} else {
SecurityContextHolder.clearContext();
}
} catch (Exception e) {
SecurityContextHolder.clearContext();
handleError(response, "Signature verification failed: " + e.getMessage(),
400);
}
```

Solution

In general it it recommended to use unified authorization layer, same for all services. Further proper authorization control must be performed on every user action. For example if user sends message, there must be verification whether the user is owner of the sender public key. This can be archived by singing the message and verifying its signature with sender's public key.

4.7. Client Code Quality

4.7.1. Code verification invalid stream seed (code)

Risk level	Impact	Likelihood
Medium	High	Low

Finding

Vulnerable. We noticed that SMS verification codes first digit is fixed. Therefore the entropy of the random 6 digits code is lowered to 5 digits. Every code is prefixed with `0`. It is because of the following function `static IntStream iterate(int seed, IntUnaryOperator f)`. Generated `Stream` consisting of `seed`, `f(seed)`, `f(f(seed))`, etc.

Documentation: https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#iterate-int-java.util.function.IntUnaryOperator-

```
public static String retrieveRandomDigits(int length) {
try {
SecureRandom secureRandom = SecureRandom.getInstance(ALGORITHM,
PROVIDER);
return IntStream.iterate(0, i -> secureRandom.nextInt(10))
.limit(length)
.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append)
.toString();

} catch (NoSuchAlgorithmException | NoSuchProviderException e) {
log.error("Error occurred during a generation of random digits. Error:
{}", e.getMessage());
throw new RuntimeException("Error occurred during a generation of random
digits.");
}
}
```

The code is valid only for 30 seconds.

Solution

Set the `seed` value to `secureRandom.nextInt(10)`.

4.7.2. Hardcoded secrets (Android, iOS, API)

Risk level	Impact	Likelihood
Medium	High	Low

Finding

Vulnerable. **iOS**

File `vexl-ios-devel/vexl/Resources/Constants.swift`:

```
static let contactsHashingPassword = "VexlVexl"
    static let localEncryptionPassowrd = "tmpPassword"
```

On iOS hardcoded encryption password is used for private key encryption:

```
extension ManagedKeyPair {
    var privateKey: String? {
        get { try? encryptedPrivateKey?.aes.decrypt(password:
Constants.localEncryptionPassowrd) }
        set { encryptedPrivateKey = try? newValue?.aes.encrypt(password:
Constants.localEncryptionPassowrd) }
    }

    var keys: ECCKeys? {
        guard let pubK = publicKey else {
            return nil
        }
        return ECCKeys(pubKey: pubK, privKey: privateKey)
    }
}
```

Android

File `vexl-android-main/cryptography/src/main/java/com/cleevio/vexl/cryptography/HmacCryptoLib.kt`:

```
const val HMAC_PASSWORD = "VexlVexl"`
```

For both platforms HMAC password is used for computing HMAC of user's phon number used as hash.

In general using shared and hardcoded secret does not provide additional security. They can be easily leaked by reverse engineering techniques.

API

We identified multiple secrets in the code repositories:

- https://gitlab.cleevio.cz/backend/vexl/vexl-chat-ms/blob/47776ebd0746482bc832bd801901e3d96d6f830d/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-chat-ms/blob/99da4382bc12a2ff78235a38c66cd5fec31d850a/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-chat-ms/blob/9af6ee9aa1eb6594ff613200b29d61712d7147ed/src/main/resources/firebase.json
- https://gitlab.cleevio.cz/backend/vexl/vexl-chat-ms/blob/fa7c5c2d6e334365706641b38d82cc648b4b1f3b/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-contact-ms/blob/24b53b897da2c716d825fb115ad6913b854a6cea/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-contact-ms/blob/661d4bfaf7a644d46d4a414bc68a73aba277d3dc/src/main/resources/firebase.json
- https://gitlab.cleevio.cz/backend/vexl/vexl-contact-ms/blob/8c3d1fe1f3b114776f540db37c630ed276e867e1/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-contact-ms/blob/d9f017263d7c674807b206dc93b28aba29014961/src/main/resources/application-stage.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-offer-ms/blob/52580f701b385218526567d32c495d6d2150a247/src/main/resources/application-stage.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-offer-ms/blob/59fc8fcaa59db1f6b577a0b86ae5245808f4c739/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-offer-ms/blob/c9f782bc612cb1ac2b3685535eed25e90bb95d3a/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-offer-ms/blob/cf1c5a395d86bbaec33afc38a7f440c750528984/src/main/resources/application-dev.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-user-ms/blob/486635a18c1f7940fb836c5358e2351b628e2571/src/main/resources/application-dev.properties

- https://gitlab.cleevio.cz/backend/vexl/vexl-user-ms/blob/c50a5fbad31e7904c98811d91ab1cf7a4945b552/src/main/resources/application.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-user-ms/blob/c50a5fbad31e7904c98811d91ab1cf7a4945b552/src/test/resources/application-test.properties
- https://gitlab.cleevio.cz/backend/vexl/vexl-user-ms/blob/d7037f7d4e01e181ae988ecf46602c945883d650/src/main/resources/application-stage.properties
- https://gitlab.cleevio.cz/cleevio-dev-ios/SwiftInstagram/blob/c204f547d4a8039c7055de925c7d561fb2211d90/Cartfile.resolved
- https://gitlab.cleevio.cz/clients/vexl-ios/blob/de299f3156f8d47233daa0f541b7eb3e4102aa6a/ServiceAccount.json

Some of them are not critical or does not represent serious security issues, but in general it is not recommended to store secrets and keys within code repositories. The most serious ones are secrets for:

- PostreSQL databases
- AES encryption keys
- gcloud service accounts - (may be false positive)

Solution

iOS provides a mechanism for working with encryption keys, referred to as Secure Enclave. Similarly to Android, any interactions with these keys are only possible via programming interfaces - not by reading files directly.

You can also generate keys and store them in the Keychain yourself.

If iOS apps use keys stored in Keychain, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The secure enclave provides functionality, where signing can be performed in this secured environment. It means that private key never leaves secure enclave processor. For more information see: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave.

Android offers an interface known as Keystore. Apps can only obtain encryption keys via the programming interface, not by reading files directly. The files containing the encryption keys are themselves stored in `/data/misc/keystore/` and belong to the keystore user.

If Android apps encrypt all sensitive data using keys from Keystore, it will not be possible to decrypt the data unless there are vulnerabilities permitting arbitrary code execution. Arbitrary code execution is a critical but comparatively uncommon vulnerability. But other, less critical vulnerabilities can reduce the risk of sensitive data stored in the file system being leaked to zero.

The same can be applied for singing.

Do not used HMAC with hardcoded secret. It is same as the salted hash function.

For code repository do not store secrets within source code files. Best solution is to use Key Vault like service to avoid storing secrets on filesystem. Further Gitlab CI/CD provides also solution for storing secrets outside source code files - CI/CD variables.

We recommend using tools like detect-secrets or trufflehog in you CI/CD pipelines, or as a pre-commit hook.

4.7.3. Account delete - storage not cleared (Android)

Risk level	Impact	Likelihood
Low	Medium	Low

Finding

Vulnerable. The application provided functionality for deleting user account, like red button. but on Android device no data were deleted after account delete.

Solution

Properly clear whole application storage and server data on user delete action.

4.7.4. File path manipulation (code, API)

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. The `ImageService` functionality stores user avatar. We identified that extension of the file name is taken from user input and it is not validated. in case of NTFS like filesystem it would be possible to perform path traversal. But currently, there is not this file system in use. Traversing from not exsisting file is not allowed.

File: `vexl-contact-ms-master/src/main/java/com/cleevio/vexl/module/file/service/ImageService.java`

```
public String save(@NotNull ImageRequest imageRequest) throws
 FileWriteException {
         try {
             final File dir = new File(this.contentPath);

             if (!dir.exists() && !dir.mkdirs()) {
                 throw new FileWriteException();
             }
```

```
            final String fileName = UUID.randomUUID().toString();

            final File destination = new File(dir, fileName + "." +
imageRequest.getExtension());

            try (FileOutputStream stream = new
FileOutputStream(destination)) {

stream.write(Base64.getDecoder().decode(imageRequest.getData()));
            }

            log.info("created file {}", destination.getPath());

            return String.join("", urlPath, destination.getName());
        } catch (IOException e) {
            log.error(e.getMessage());
            throw new FileWriteException();
        }
    }
```

Request:

```
POST /api/v1/user HTTP/2
Host: user.vexl.staging.cleevio.io
Content-Type: application/json
Public-Key:
LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUZZd0VBWUhLb1pJemowQ0FRWUZLNEVFQUFvRF
FnQUUyTXpXNVdsZkpabUVmUjc0Q28ydU1UeUprUmlMRExZdgpMa2Z5TjZvNHJpZGUvTGlyTVky
MGFmblFhS0pDV1RwOVU0VjdKZXQ5bWFENEdHemgrSVpCUGc9PQotLS0tLUVORCBQVUJMSUMgS0
VZLS0tLS0K
Signature:
MD4CHQCYo9DvqwQju3gTbvNFDjihV67W88juWZNg3b/4Ah0AsenvDK+4E7FNAMGRRuNnBg/8q3
LgGMXCNPbaNw==
Hash: uFJJ59YGqD+mhcu3WeXQLBUFWEPBt7rIYjL7Li00yIo=
Content-Length: 92
X-Platform: ios

{"username":"<@random(10)>123123<@/random>","avatar":{"data":"","extension
":"../../../xxx"}}
```

Response:

```
HTTP/2 500 Internal Server Error
Date: Thu, 11 Aug 2022 15:26:31 GMT
Content-Type: application/json
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
```

```
X-Content-Type-Options: nosniff
X-Xss-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Strict-Transport-Security: max-age=15724800; includeSubDomains
X-Frame-Options: DENY

{"message":["Issue with store of file."],"code":"102100"}
```

Solution

Do not enter user entered data into file path without proper validation - whitelisting.

4.8. Code Tampering

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. The applications do not check if it is running on rooted or jailbroken device. It is still possible to bypass this checks, but it is kind of mitigation against common code tampering attacks. The application should not run if root or jailbreak is identified. In a case of rooted/jailbroken device the malicious application running under privileged user is able to modify or read any application data, memory or code.

Solution

For production release on iOS device we recommend to follow at least these principles:

- https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06j-testing-resiliency-against-reverse-engineering
- https://duo.com/blog/jailbreak-detector-detector
- https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/jailbreak-detection-methods/

There is a open source project:

- https://github.com/securing/IOSSecuritySuite
- https://github.com/securing/IOSSecuritySuite/blob/master/IOSSecuritySuite/JailbreakChecker.swift

4.9. Reverse Engineering

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. It is still possible to reverse engineer the application code if the attacker is not time limited. But in case of not obfuscated code it is much easier. We were given the testing version of the applications so they were not obfuscated.

Solution

For Android application use `ProGuard` or for better results `DexGuard` solution.

iOS application are in some cases denied by App Store policy. Therefore use only allowed obfuscation methods. In general it is recommended obfuscating your application to make it harder to reverse engineer.

We recommend removing all debug information and unused application code from the application before production deploy and stripping the binary. To strip symbols in your app go to "Build Settings > Deployment > Strip Swift Symbols" and setting the option to Yes.

4.10. Extraneous Functionality

Risk level	Impact	Likelihood
Note	Low	Low

Finding

Vulnerable. We identified following endpoint, which is used only for testing purposes:

- https://chat.vexl.staging.cleevio.io/api/v1/temp

File `vexl-chat-ms-master/src/main/java/com/cleevio/vexl/module/temp/TemController.java`:

```
@Tag(name = "TEMP - only for testing purposes")
@RestController
@RequestMapping("/api/v1/temp")
@RequiredArgsConstructor
public class TemController {

    @PostMapping
    @Operation(summary = "ECDSA verifying", description = "Only for
```

```
testing purposes")
    public TempResponse signItTemp(@RequestBody TempRequest request) {
        return new
TempResponse(CLibrary.CRYPTO_LIB.ecdsa_sign(request.publicKey(),
request.privateKey(), request.challenge(),
request.challenge().length()));
    }
}
```

Although it does not represent a security issues, in general unreferenced files and not used components should be removed from production release.

Android source code function definition - `vexl-android-main/repository/src/main/java/cz/cleevio/repository/repository/contact/ContactRepositoryImpl.kt`:

```
override suspend fun generateContactsTmp(): Resource<Unit> = tryOnline(
request = {
contactApi.generateContactsTmp(
hash = encryptedPreference.hash,
signature = encryptedPreference.signature
)
},
mapper = { }
)

override suspend fun generateFacebookContactsTmp(): Resource<Unit> =
tryOnline(
request = {
contactApi.generateContactsTmp(
hash = encryptedPreference.facebookHash,
signature = encryptedPreference.facebookSignature
)
},
mapper = { }
)
```

File `vexl-android-main/network/src/main/java/cz/cleevio/network/api/ContactApi.kt`:

```
@POST("../../temp")
suspend fun generateContactsTmp(
@Header(AuthInterceptor.HEADER_HASH) hash: String? = null,
@Header(AuthInterceptor.HEADER_SIGNATURE) signature: String? = null,
): Response<ResponseBody>
```

Solution

Check for libraries that are communicating with the listed servers and remove those that are not needed.

5. WHY TO RE-TEST

The motivation for penetration testing is the protection of personal, financial or otherwise sensitive data, the reputation of your company and its trade secrets. Regular penetration testing can prevent the unpleasant situation of a hack by an external attacker or a rogue employee. Ethical hacking is an ideal way to test your security in practice.

The often omitted part of a pentest is the re-test of the application after a fix for the found vulnerability has been implemented. The idea of a re-test is to ensure that the implementation of the fix is correct, and the vulnerability is no longer exploitable.

Integration of penetration testing into the System Development Life Cycle (SDLC) can yield dramatic results to the overall quality of the code developed. Consider it as one of the layers in a defense-in-depth approach to application security. The idea of integrating a phase into your SLDC may sound daunting, yet another layer of complexity or an additional cost, but in the long term and in today's cyber landscape it is cost effective, reputation building, and in the best interest of any business to do so.

6. DISCLAIMER

All findings described in this report refer only to the application version used during the testing and reflect the status of the application during the testing period defined in section "2. ASSESSMENT SCOPE AND DETAILS." Other installed versions/build, past and future releases of the application may not be affected by the mentioned vulnerabilities or may be affected by different vulnerabilities.

Please keep in mind that the negative results of the tests ("Not vulnerable.") DO NOT represent 100% accuracy that the application is not vulnerable to this type of attack. Attack methods and attack vectors evolve rapidly and therefore attackers may improve the existing attacks or find new innovative ways to exploit vulnerabilities currently present, but undetected, in the application. This represents a strong argument for integrating penetration tests into the application Software Development Life Cycle on a periodic schedule.