SUBGRAPH

FINDINGS REPORT

Tella

May 5, 2023

Prepared for: Open Technology Fund

Subgraph Technologies, Inc.
642 Rue de Courcelle, Suite 309
Montreal, Quebec
https://subgraph.com

Contents

Executive Overview

Tella is an open source platform intended to be used by individuals and organizations to collect data on human rights violations in potentially dangerous environments. The Tella project is based primarily on two mobile apps, one for Android, and one for iOS, that support collection of photo, video, audio, and other types of data. Both Tella Android and Tella iOS support submission of data to remote servers, including servers running Tella Web.

Effort has been applied to make the apps safer to use. These efforts include data protection (encryption), camouflage, secondary authentication for the app, and more.

Subgraph was engaged in 2022 to perform security testing of the Tella applications and backend through the Open Technology Fund Red Team Labs program.

Tella for Android is based on the original code from the Whistler app originally developed by Build a Movement. The Whistler Android app and the web backend were pen-tested in the past, before Horizontal became custodians of the project. That report is available here.

Tella for iOS is written from scratch, in Swift. The codebase is entirely independent from the Android application.

Both apps and the Tella project as a whole are maintained by Horizontal.

Tella Web is an open source backend for the Tella clients written in Javascript, with a server-side component based on NestJS and the front-end based on NextJS. Both Tella Android and Tella iOS can submit data a Tella Web server that can be configured in the application. Tella Android can also submit reports and files to ODK Collect and Uwazi servers.

Scope

The scope of the test included latest versions of both applications. Both run-time testing and source code review were employed.

Versions examined

- *Tella Android app*
  - Version tested 2.0.9 (121) on Android mobile device running Android OS 9 and in emulator with API level 28
  - Source release 2.0.5 and 2.0.6 develop branch
- *Tella iOS app*
  - Develop branch reviewed, beta tested
  - Source release HEAD as of latest test date
- *TellaWeb NestJS*
  - Version examined was HEAD as of test date
- *TellaWeb Frontend*
  - Version examined was HEAD as of test date

Major functional components in scope: Android app

- User authentication and locking/unlocking
- Management of cryptographic keys
- Data encryption and decryption
- Data protection
- Forms
- Camouflage
- Submission of data to Tella Web, Uwazi, and ODK Collect

Major functional components in scope: iOS app

- User authentication and locking/unlocking
- Management of cryptographic keys
- Data encryption and decryption
- Data protection
- Submission of data to Tella Web

Due to platform constraints, camouflage on iOS is not implemented as a functional component of the app. Forms are not yet implemented.

Major functional components in scope: Tella Web

- TellaWeb NestJS Backend
- TellaWeb NextJS Frontend

Objectives

The key objectives of the mobile audit included the following:

- **OBJ-1:** Assess risk of casual unauthorized physical access to or confiscation of device by reviewing the app locking mechanisms
- **OBJ-2:** Assess risk of prolonged unauthorized access to, confiscation or, or tampering with of device by reviewing aspects of the application runtime lifecycle
- **OBJ-3:** Assess risk of prolonged unauthorized access to, confiscation or, or tampering with of device by reviewing aspects of timeouts
- **OBJ-4:** Assess risk of threat of confiscation and unauthorized access by reviewing the shutdown and quick delete features
- **OBJ-5:** Identification of any artifacts on the system while the application is in camouflage mode (Android)
- **OBJ-6:** Identification of any artifacts left on the filesystem after uninstall (Android)
- **OBJ-7:** Provide observations related to effectiveness of camouflage
- **OBJ-8:** Summarizing privacy risk in analytics data collection
- **OBJ-9:** Assess risk in interaction with remote servers

Meeting standard security review criteria for mobile applications are included as objectives in addition to the above:

- **OBJ-10:** Ensuring local authentication cannot be bypassed
- **OBJ-11:** Enforcing authorization as designed
- **OBJ-12:** Secure defaults
- **OBJ-13:** Correct function of data protection features, such as encryption
- **OBJ-14:** Sanitization of input where required
- **OBJ-15:** Third-party dependency risk
- **OBJ-16:** Other authentication

The audit of the Tella web implementation followed that of a standard web-based application/API backend with focus on the following:

- Authentication mechanism
- Role-based access control
- Input sanitization
- Serialization
- Session management
- Direct object access
- Middleware configuration
- Use of cryptography

Methodology

A high level summary of the activities performed for each objective are listed in the table below. Android and iOS have very different runtime testing characteristics, with iOS having many more constraints, making some tests challenging, or limited.

Objective	Android	iOS
OBJ-1	Device test, debugger, code review	Device test, code review
OBJ-2	Device test, debugger, code review	Device test, code review
OBJ-3	Device test, debugger, code review	Device test, code review
OBJ-4	Device test, filesystem inspection	Not tested
OBJ-5	Device test	Not applicable
OBJ-6	Filesystem inspection	Not tested
OBJ-7	Device test	Not tested
OBJ-8	Code review	Not applicable
OBJ-9	Device test, code review	Code review
OBJ-10	Device test, debugger, code review	Device test, code review
OBJ-11	Device test, code review	Device test, code review
OBJ-12	Code review	Code review
OBJ-13	Code review, filesystem inspection	Code review
OBJ-14	Code review, device test, network	Not applicable
OBJ-15	Code review	Code review
OBJ-16	Code review, network	Not applicable

- **Device test** Interacting with the application as a user on a physical device or in an emulator
- **Filesystem access** Accessing the filesystem through an attached debugger while the app is running, or after it has been installed, or after it has been removed
- **Debugger** Sending messages to the application while it is running, inspecting its runtime in various ways
- **Code review** Reading and documentation of source code
- **Network** Interacting with a simulated malicious or breached server

Tools Used

- adb
- Android emulator
- Standard Linux tools
- nginx for ODK server simulation
- Physical device running iOS
- Physical device running Android
- Burpsuite Pro

Tella Web

The Tella web components were audited using a mixture of limited runtime testing and tactical code review. Client interactions were audited in several ways:

- Manual tests against simulated malicious server endpoints (Uwazi, ODK Collect, Tella Web)
- Tactical code review

Observations and Recommendations

Observations on Android Analytics and Crash Data Collection

Tella Android includes integration of CleanInsights for collection of usage measurements. CleanInsights is designed to minimize data collection, yet allow for the developers to gain insight into how the application is being used by users. It is opt-in, though users are encouraged to participate through the Tella UI at and after the time of installation. The server ingesting data from apps appears to be hosted by Horizontal, and not a third party.

The Android app also uses Google Firebase Crashlytics. This is opt-out, at least in versions tested by Subgraph. This data collection should be considered for opt-in rather than opt-out, as the data is sensitive: exceptions are sent to a server run by a third-party. Some of these may be sensitive, and there may be jurisdictional concerns related to the data collection.

See *mobile/src/main/java/rs/readahead/washington/mobile/data/sharedpref/Preferences.java*:

```
public static boolean isSubmittingCrashReports() {
    return getBoolean(SharedPrefs.SUBMIT_CRASH_REPORTS, true);
}
```

Observations on Authorization: Mobile

Both versions of the Tella app keep data in locations expected to be inaccessible through the default user filesystem interface provided by the mobile OS. Modern devices enforce this reasonably well. Adversaries with root access on the device will be able to retrieve files, which is why local file encryption is a core design feature, and, in the case of Android, the database is also encrypted.

Files can be moved out of the Tella internal store and into the user exposed filesystem. Subgraph recommends that the implications of this be considered:

If a file is moved out of Tella, should it remain visible within Tella?

What if a user moves a file out of Tella, and then deletes the app, forgetting to remove the exported file?

Is this a plausible scenario, and it is a reasonable expectation that the user remembers to remove the copy?

Also worth considering that the app itself has very limited control outside of its domain of authorization within the filesystem.

There is no clear recommendation for this observation apparent at the time of writing. Without a submission system, it seems like this export feature would be necessary.

The developers should study the risk based on user behavior and design defensively accordingly.

Observations on Authorization: Tella Web

Tella Web implements a fairly simple role-based access control model where a user can belong to one of several roles:

- admin
- editor
- reporter
- viewer

Each of these roles can perform read or update operations on data objects, with some role-based limitations. For example, users with a non-admin role can update their own profile but cannot change their role. Role rules are applied at various controllers in a consistent manner throughout the API using NestJS decorators:

```
@AuthController('file', [RolesUser.ADMIN, RolesUser.EDITOR])
export class DeleteFileController {
  constructor(
    @Inject(TYPES.applications.IDeleteFileApplication)
    private readonly deleteFileApplication: IDeleteFileApplication,
  ) {}

  @Delete(':reportId/:fileId')
  async handler(
    @Param('fileId')
    fileId: string,
  ) {
    const deleted = await this.deleteFileApplication.execute(fileId);
    return { deleted };
  }
}
```

The *AuthController* decorator uses Guards to match the designated roles against the role of the user, which is stored as a column in the user table. Overall, the RBAC model is very simple and appeared to be effective in partitioning the API functionality into basic roles.

See tellaweb-nestjs/src/modules/user/guard/roles.user.guard.ts

Observations on Data Encryption: Android

Data collected on the device is written to temporary files and encrypted as soon as possible. This means that some media files (audio, video) will persist in cleartext temporarily and could potentially be recovered if an adversary has physical access to the device and root access.

Tella uses encryption in three different ways:

- Encryption of the main key (AES-128 in GCM mode)
- Encryption of SQLite databases
- Encryption of media files

The main key is a long-term key generated when the application is initialized. It is used for the following:

- Encryption of the Tella and Tella Vault databases
- Derivation of keys used for encrypting media files on the filesystem

The main key must be protected as it is critical and persists through the lifetime of the application. This is accomplished through wrapping with another key that is related to the method selected to unlock the application.

Main Key

The main AES key is 256-bits in length and generated in the Tella Keys package using AES software key generation provided by platform. This key is generated at the first attempt to retrieve it, if it does not exist, e.g.:

TellaKeysUI.getMainKeyStore().store(generateOrGetMainKey(), config.wrapper,...

This is the *main key*, used to encrypt the database and to derive keys for file encryption. It will persist and not be changed through the life of the application while installed on a device. The key is wrapped using another key that is derived from the authentication method selected by the user or generated by the Android keystore if device credentials are available.

If device credentials are not available, the key used to wrap the main key is derived from either an unlock pattern sequence, a PIN, or a password. The key derivation that is performed begins with conversion of the unlock credential to a password string:

PBEKeySpec(password.toCharArray()) in the case of a password unlock configuration,

PBEKeySpec(PatternUtils.patternToSha1String(pattern).toCharArray()) in the case of a pattern sequence unlock configuration,

or *PBEKeySpec(pin.toCharArray())* in the case of a PIN unlock configuration.

This string is then used to derive the AES wrapping key using Java native secret key generation using PBKDF2WithHmacSHA1, a 16-byte random salt sourced from SecureRandom() and 10,000 iterations.

Android Keystore Wrapper

The Android Keystore wrapper is used when device credentials (i.e. biometric) are supported by available hardware. This links accessibility of the key to device authentication with these other factors, and uses a wrapping key that is generated by internal OS and hardware mechanisms. The parameters for the key are as follows:

- AES-GCM

- No padding
- Authentication required to perform cryptographic operations using the key
- Authentication validity duration of 10 seconds

See *tella-keys/src/main/java/org/hzontal/tella/keys/wrapper/AndroidKeyStoreHelper.java*.

The use of this option should be encouraged, as it is the strongest method of protecting the main key currently available.

Persistence of Wrapped Key

Once wrapped, the encrypted main key is stored in Shared Preferences data store along with metadata about how it was wrapped, the IV, the wrapping key salt, and the wrapping key iteration count.

Vault

The Tella Vault is a SQLCipher encrypted database that contains metadata about the files that are stored on the filesystem. That metadata includes:

- The type of the file
- Its path location within a hierarchical virtual “filesystem” structure

File Encryption

Media files such as photos, videos, and audio recordings, are saved on the filesystem individually. Each of these files is encrypted with AES in CTR mode using a unique key which is derived deterministically from the master secret key using the PBKDF2 key derivation algorithm. The salt is the file name and there are 1000 iterations. Horizontal may wish to evaluate an increase in the number of iterations for these keys.

Encryption and decryption of media files is performed at the I/O level using an abstraction called the EncryptedFileProvider, as well as similarly within an extended abstraction called the Vault. Keys are generated deterministically, derived with 1000 iterations of PBKDF2 and with the salt from the filename and the main key. When a file is written to the filesystem, the following procedure is performed by class provided by the *EncryptedFileProvider* or the *Vault* that provides an I/O stream interface with encryption and decryption internally implemented:

1. The main key is fetched
2. A secret key is created deterministically using the main key and the filename and PBKDF2
3. An IV is created using 16 random bytes source from *SecureRandom()*
4. The IV is written to offset 0 of the new file
5. The file contents are even encrypted and the ciphertext is written to the file output stream immediately following the IV

Decryption is performed with a similar, reverse set of steps:

1. The main key is fetched
2. The secret key for the file is recreated using the main key and the filename and PBKDF2
3. The IV bytes are read from offset 0 of the encrypted file
4. The secret key and the IV are used to decrypt the data while it is being read, with cleartext bytes written to consumer

See *mobile/src/main/java/rs/readahead/washington/mobile/data/provider/EncryptedFileProvider.java*. See *tella-vault/src/main/java/com/hzontal/tella_vault/BaseVault.java*. See *tella-vault/src/main/java/com/hzontal/tella_vault/CipherStreamUtils.java*.

The code duplication across these is worth noting and indicative of the project evolution. Subgraph recommends that this security sensitive functionality be refactored to reduce attack surface and reduce the likelihood of inconsistency. An example of where such a risk could manifest is the *createSecretKey()* method which has 3 nearly identical implementations in the Tella source tree. Correspondingly, the constant **PBKDF2_ITERATIONS** is defined 3 different times:

```
[user@localhost Tella-Android]$ grep -ir PBKDF2_ITERATIONS .

./tella-vault/src/main/java/com/hzontal/provider/VaultProvider.java:
private static final int PBKDF2_ITERATIONS = 1000;

./tella-vault/src/main/java/com/hzontal/tella_vault/CipherStreamUtils.java:
private static final int PBKDF2_ITERATIONS = 1000;

./mobile/src/main/java/rs/readahead/washington
/mobile/data/provider/EncryptedFileProvider.java:
private static final int PBKDF2_ITERATIONS = 1000;
```

Database

Tella Android uses SQLite as a local database. These databases are intended to store metadata about files, as well as report content. The databases are located at:

/data/data/org.hzontal.tella/databases/tella.db

/data/data/org.hzontal.tella/databases/tella-vault.db

The databases are encrypted using SQLCipher which encrypts the entirety of the database file using AES-256 operating in CBC mode. The encryption is transparent; decryption is performed at the time the database is opened for access.

The SQLCipher raw key format is used. The main key is used as the encryption key in the following manner:

asyncSubject.onNext(DataSource.getInstance(this.context, MyApplication.getMainKeyHolder().get().getKey().getEncoded()));

See *mobile/src/main/java/rs/readahead/washington/mobile/data/database/KeyDataSource.java*.

The database is created with the SQLite method getWriteableDatabase() if it does not already exist.

See *mobile/src/main/java/rs/readahead/washington/mobile/data/database/CipherOpenHelper.java* and *tella-vault/src/main/java/com/hzontal/tella_vault/database/CipherOpenHelper.java*.

Observations on Data Encryption: iOS

Meta and Regular Keypairs

Tella uses two main key pairs: the meta keypair and the regular keypair.

- Meta keypair, used to protect the file encryption private key. The meta private key is stored in hardware secure enclave and protected by device authentication capabilities.
- Regular keypair, used to encrypt data on the filesystem. The private key is encrypted with the meta key pair.

The meta private key is created by and stored within the Secure Enclave when the application is initialized for the first time. The meta private key is also rotated when the authentication is updated. The meta public key is derived from the meta private key and stored on the device filesystem at a fixed location.

The regular private key is encrypted with the meta public key and stored on the filesystem, along with the regular public key. The regular keypair is created when the application is initialized for the first time. The regular keypair persists through the life of the application while it is installed.

See *Scenes/Authentication/Views/Unlock/UnlockPinView.swift* and *Scenes/Authentication/Views/Unlock/UnlockPasswordView.swift* with keypair creation/update and rotation in *VaultManager/CryptoManager.swift*.

The meta key is created with the following access control flags and attributes:

```
let access = SecAccessControlCreateWithFlags(
        kCFAllocatorDefault,
        kSecAttrAccessibleWhenUnlockedThisDeviceOnly,
        [.privateKeyUsage, type.toFlag()],
        nil)!
let attributes: [String: Any] = [
    kSecAttrKeyType as String: kSecAttrKeyTypeECSECPrimeRandom,
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPrivateKeyAttrs as String: [
        kSecAttrIsPermanent as String: true,
        kSecAttrApplicationTag as String: appTag,
        kSecAttrAccessControl as String: access
    ]
]
var error: Unmanaged<CFError>?
```

Data Encryption

Each file is encrypted using the regular public key read from the filesystem. The algorithm used is AES-GCM with a 128-bit key derived using ANSI X9.63 KDF using SHA256. Use of the Integrated Encryption Scheme is the recommended way to use asymmetric encryption to encrypt and decrypt data provided by the iOS native cryptographic framework.

At the time that files are saved, the public key is read from the filesystem and passed to the iOS library function *SecKeyCreateEncryptedData()* with the regular public key as the key argument and the algorithm parameter set to .eciesEncryptionCofactorX963SHA256AESGCM.

See *Filemanager.swift* and *CryptoManager.swift*.

Temporary Cleartext Files

Camera and audio media data is written to temporary location *NSTemporaryDirectory()* in cleartext at various points, such as while being recorded or are being played. This location is proactively cleared using methods such as *VaultManager.clearTmpDirectory()*. It was observed that this does not appear to be the case for audio files (see V-003), though other calls to clear the directory are made at other points in the application lifecycle.

For example, a proactive clearing of the temporary directory is included in *Scenes/Camera/ViewModel/CameraViewModel.swift*.

However, at the location where one might expect it, no such invocation exists in the corresponding *Scenes/Audio/AudioPlayer/Views/AudioPlayerView.swift*.

Subgraph recommends that this be investigated by the iOS app developers.

Observations on Authentication: Android

Application Unlocking

There are four methods of unlocking the application:

- Pattern lock
- Min 6 digit PIN
- Password
- Device credential (i.e. biometric)

Each of these authentication methods is linked to protection of the main secret key used to encrypt the database and from which file encryption keys are derived. The value of the authentication credential is used to wrap the main key in the cases of the pattern lock, the PIN, and the password authentication methods. A secret key generated by the Android keystore is used if the device credential authentication method is used.

The device credential is the most secure due to the use of a wrapping key that is generated by and stored in the OS keystore.

See *tella-keys/src/main/java/org/hzontal/tella/keys/wrapper/AndroidKeyStoreHelper.java*.

The least secure option is the pattern lock. The pattern sequence is used as the secret for creation of the key used for encryption of the main key.

See *tella-locking-ui/src/main/java/com/hzontal/tella_locking_ui/ui/pattern/PatternSetConfirmActivity.kt*.

The user is presented with guidance that informs them of this when they are setting or changing the unlocking method. It is the opinion of Subgraph that this guidance is potentially confusing. The difference in strength between the PIN and the unlock pattern is probably not close to the difference between a good password and the PIN, yet they are presented to the user on a scale that might be interpreted as consisting of equidistant tiers:

- Password: most secure
- PIN: moderately secure
- Pattern: weakest

The real scale may be something like:

- Device credential: Best
- Password: OK if good password used
- PIN and Pattern: Very weak

Tella should consider communication value of the scale presentation, as this credential is used for key derivation and data protection.

Authentication Bypass via Direct Activity Invocation

In an older version of the Whistler app, it was possible to bypass locking by using physical interaction with the Activity Manager via *adb* to invoke activities directly. Subgraph attempted this with the latest publicly available version of Tella for Android, e.g.:

```
adb shell am start -a "action.intent.action.VIEW" -n
"org.hzontal.tella/rs.readahead.washington.mobile.views
.activity.ServersSettingsActivity"
```

No activities were found to be exploitable to bypass unlocking. The mechanism to enforce locking was found to be in the base class *BaseLockActivity* which activities such as *ServersSettingsActivity* extend.

Other Authentication: Remote Peers

The Tella Android app dispatches HTTP clients to query for forms, to download forms, to download form parts, and to upload structured information. Subgraph verified the following behaviour when interacting with OpenDataKit (ODK) servers, Uwazi servers, and Tella Web servers:

- HTTP server targets are not accessed, a non-specific error is reported to the user when an attempt is made
- X.509 certificate validation failures cause a non-specific error to be reported to the user
- HTTP targets in ODK form definitions are not accessed, a non-specific error is reported to the user
- Upload targets in Uwazi templates are not used; rather the server URL is the target
- The upload target for Tella Web is the Tella Web server

Observations on Authentication: iOS

Tella for iOS authenticates the user to the application using two methods:

1. Password of 6 characters of more
2. PIN of 6 digits or more

The password option is identified to the user as being the most secure option, with recommendation being made not to choose easily guessed values such as *password*. The password *password* is an acceptable password, however, as the application does not enforce its guidance beyond requiring a minimum length of 6 characters.

Enforcement of App Unlock

The App forces a reset of the lock state when a scene changes to the *.background* state.

See *Application/TellaApp.swift*.

Observations on Authentication: Tella Web

Users authenticate to Tella Web with a username and password. The password is hashed prior to being stored in the database using bcrypt. The number of iterations to generate a salt is configurable, with the default being 2^10 (~10 hashes/sec on a 2GHz core).

See *src/environment/bcrypt-salt.environment.ts*.

Observations on Session Management: Tella Web

An authenticated session is maintained using a session token. The encoded access token is sent to clients as the response payload and in a cookie, with the expectation that it be transmitted by clients via the Authorization header. The token payload contains the userid and is valid for 23 hours from the time it is signed using a server-side secret defined in *src/.env*.

See *src/modules/auth/services/generate-token.auth.service.ts*.

Observations on Input Validation

Tella Android: Forms

Testing of handling of XML documents retrieved from remote servers was performed. This was done in part because of the possible presence of CVE-2017-1000190. No attack was confirmed possible. The tests consisted of various types of malformed / hostile XML documents were served from a server on the Internet to clients (Tella Android apps) that were configured to consume them.

Input validation testing was also performed against the SQL operations related to forms and form content. No exploitable vulnerabilities were discovered.

Tella Android will download JSON template definitions from Uwazi servers so that these can be rendered as forms. Subgraph did not find any exploitable vulnerability in this process.

Tella iOS

N/A

Tella Web

Tella Web persists data to an SQLite database and uses TypeORM for application interactions with it. The use of TypeORM in Tella Web was consistent with what the documentation recommends for avoiding SQL injection issues, i.e., through the use of prepared statements. The Tella database stores data about users, reports, remote configurations, and projects. File metadata is stored in the database, with data stored outside.

Client uploaded files such as images, videos, etc, are stored on the server side on the server filesystem. This is understood to be a temporary limitation as there is an abstraction implemented over files that are uploaded by clients. Each file is assigned a logical entity, with a UUID as the primary identifier. These IDs used as identifiers for buckets, which are the containers for file data. Currently, buckets are paths in the *data* directory on the server, however it is likely that this will be cloud object storage in the future. Each bucket contains a *full* folder and a *partial* folder for incomplete uploads. As mentioned above, file entity metadata, including the original filename, is stored in the database: only the server generated bucket ID is used on the file system.

See src/modules/file/handlers/storage/storage.file.handler.ts.

Observations on Camouflage

Both Tella apps offer a method to hide their presence from casual detection. They have different approaches due to different platform considerations.

Android App Camouflage

It is possible for Android apps to change identifying visual characteristics, such as their name and launch icon. This capability is employed to implement a camouflage capability where the Tella Android app can pretend to be something else.

This was observed to function with reasonable effectiveness in its most basic mode of operation (more on the functioning calculator is below). Some thought has clearly been put into the icons available for the simple camouflage mode: the icons are designed to not attract attention if an adversary is inspecting a phone by flipping through the installed apps.

A second camouflage mode is supported on Android: a functioning decoy calculator application. The functioning calculator camouflage is a clever idea that has one crucial drawback in its implementation within Tella: it requires the PIN unlock credential to be used. Subgraph does not feel like it needs to be this way. The PIN password is likely the one with the least strength, and forcing users to use it for camouflage undermines the strength of the data protection. Subgraph recommends that the calculator de-obfuscation be triggered with another, simpler code (e.g. 111111), allowing the user to use a more secure unlock credential, such as a complex password or device credentials.

There are many indications that Tella is installed when camouflage is enabled. For example, the OS identifies it as *Tella* in the list of installed applications, or when searching for installed applications, or when uninstalling it. This is likely unavoidable and a good reminder that the camouflage feature is only useful against casual inspection. The cost of its use should not be weaker data protection.

iOS App Camouflage

iOS users are offered an alternate application for camouflage. This is a solution that offers a great deal more flexibility in obfuscation capabilities. The cost is that the distribution is more complex. There is no real way around this due to Apple's control of the platform and iOS app experience.

Observations on Third-Party Dependencies

Android App

The third-party dependencies were reviewed and two potentially risky packages were identified:

- Butterknife: deprecated
- simplexml-safe 2.6.1: Associated with a contested vulnerability, CVE-2017-1000190.

iOS App

The iOS app was not found to have any risky third-party dependencies.

Forensic Review

Observations of Tella Android Artifacts after Uninstall

Both manual app removal via the OS interface and the emergency uninstall feature were used to remove Tella from the OS. We then searched the filesystem for traces of the app and found many. This simple search method used from the adb shell (as root) yields many positive results:

```
grep -ir hzontal / -print 2>/dev/null
```

Some of the findings are provided below:

- /data/system/batterystats-daily.xml
- /data/system/graphicsstats/1656633600000/org.hzontal.tella/121/total
- /data/user_de/0/com.google.android.gms/files/fcm_package_info.ldb/000020.log
- /data/data/com.google.android.apps.nexuslauncher/shared_prefs/reflection.private.properties.xml
- /data/user/0/com.google.android.apps/nexuslauncher/shared_prefs
- /data/data/com.google.android.gms/cache/autofill_infinite_data_cache/infinite_data_cache
- /data/data/com.google.android.gms/files/clearcut/0/LATIN_IME/1531368078/1658133101923.NONE
- /data/data/com.google.android.gms/files/clearcut/2/AUTOFILL_WITH_GOOGLE/1109721557/1658133129886.NONE
- /data/data/com.google.android.gms/files/AppDataSearch/main/cur/ds.docs
- /data/data/com.google.android.gms/files/AppDataSearch/main/rq/0000000000000003
- /data/data/com.google.android.gms/files/icing_apps_corpus_entries.bin
- /data/data/com.google.android.gms/databases/gass.db
- /data/data/com.google.android.apps.nexuslauncher/files/reflection.engine
- /data/data/com.google.android.apps.nexuslauncher/databases/reflection.events
- /data/data/com.google.android.apps.nexuslauncher/databases/reflection.events-wal
- /data/data/com.google.android.apps.nexuslauncher/databases/app_icons.db-wal
- /data/system/users/0/app_idle_stats.xml
- /data/system/usagestats/0/daily/n
- /data/system/usagestats/0/weekly/n
- /data/system/usagestats/0/monthly/n
- /data/system/usagestats/0/yearly/n
- /data/misc_ce/0/storaged/storaged.proto

iOS

The iOS app was not tested for forensic traces on the filesystem due to platform limitations.

General Observations on Codebase

The Android app code is fairly complex and not commented extensively, though not difficult to read and follow. This is understandable due to the change of custodianship to Horizontal, and later development staff changes at Horizontal.

The iOS app is commented reasonably well and was much simpler and therefore easier to understand.

It was required to review the development branches of both apps, and both apps show apparent evolution and change, such as with the transition to the Vault abstraction. It seemed partially complete in both apps, and that may be the reason for some duplicated functionality observed:

tella-vault/src/main/java/com/hzontal/tella_vault/CipherStreamUtils.java and *tella-vault/src/main/java/com/hzontal/provider/VaultProvider.java* in the Android app

and

CryptoManager.swift and *VaultManager/CryptoManager.swift* in the iOS app.

Subgraph recommends that this re-use be examined and considered for refactoring to reduce the attack surface and ensure consistency.

Summary

No.	Title	Severity	Remediation
V-001	Unrestricted Unlock Attempts	Medium	In Progress
V-002	Android Cipher Stream I/O Key PBKDF2 Iterations	Low	Resolved
V-003	Tella iOS Cleartext Audio Data may Persist Longer than Required	Low	Resolved
V-004	Tella Android Outdated Retrofit2 Dependency	Low	Resolved
V-005	Tella Android Deprecated Dependency: Butterknife	Informational	In Progress

Details

V-001: Unrestricted Unlock Attempts

Severity	Remediation
Medium	In Progress

Note: Remediation of this issue is in progress, with release scheduled for the coming months.

Discussion

Both applications permit unlimited authentication attempts, which increase the risk of unauthorized access through successful guessing of the unlock credential. This risk is most emphasized with the weaker authentication methods, such as PIN or pattern sequence on Android.

Impact Analysis

This is heavily mitigated by the requirement that device mobile OS be unlocked. The app locking is intended to be a secondary line of defense.

Remediation Recommendations

Consider implementing a slowdown mechanism after repeated unlock failures. This may greatly increase attack cost with minimal user experience degradation.

Remediation Status

Not yet resolved.

Additional Information

N/A

V-002: Android Cipher Stream I/O Key PBKDF2 Iterations

Severity	Remediation
Low	Resolved

Note: The Tella development team has increased the PBKDF2 iteration count to 100000 in mobile/src/main/java/rs/readahead/washington/mobile/data/provider/EncryptedFileProvider.java

Discussion

The keys used for encrypting files on Android are derived from the main key using the PBKDF2 key derivation algorithm with the filename as the salt. One of the parameters to PBKDF2 implementations is the iteration count: the number of times the function is run before the output is the fully derived key. This is an important input parameter, as it adds computational time cost to a search of the key space by an adversary.

The number of iterations is currently set to 1000. This has been the value in Tella Android for some time, if not since inception of the project, and it may be prudent to reconsider it for the contemporary period and near future.

For comparison, NIST SP800-63b recommends a minimum of 10,000 iterations in guidance that was published in 2016. As of this writing, the current version of the 1Password product uses 100,000 iterations of PBKDF2, an increase from 10,000 in 2012. Note that this is not an endorsement of 1Password, mention of them is only included for comparison purposes as 1) they provide detailed information on their resistance to password cracking attacks, and 2) do so in mobile applications.

See the following snippet from *mobile/src/main/java/rs/readahead/washington/mobile/data/provider/EncryptedFileProvider.java*:

```
    private static final SecureRandom secureRandom = new SecureRandom();
    private static final String transformation2 = "AES/CTR/NoPadding";
    private static final int HASH_BYTE_SIZE = 128;
    private static final int PBKDF2_ITERATIONS = 1000;

[..]

    private static SecretKeySpec createSecretKey(byte[] key, String
    ↪  fileName) throws
    NoSuchAlgorithmException, InvalidKeySpecException,
    ↪  UnsupportedEncodingException {

        byte[] salt = fileName.getBytes(Charset.forName("UTF-8"));
        char[] password = new String(key, "UTF-8").toCharArray();
```

```
        KeySpec keySpec = new PBEKeySpec(password, salt,
        ↪  PBKDF2_ITERATIONS, HASH_BYTE_SIZE);
        SecretKeyFactory factory =
        ↪  SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        SecretKey sk = factory.generateSecret(keySpec);

        return new SecretKeySpec(sk.getEncoded(), "AES");
    }
```

Impact Analysis

Consider the following common estimation of password strength.

With:

```
L = Password length
N = Number of distinct symbols

Entropy = L * log(N)/log(2)
```

The three unlock credential types based on user-supplied secrets have relatively low strength:

- Approximately 40 bits for a 6 character password
- Approximately 20 bits for a 6-digit pin
- Approximately and 18.5 bits for a 6 node sequence on a 3x3 grid

Cracking difficulty estimates based on this are ideal, assuming that the passwords are sampled from the key space uniformly randomly. In practice, they are not.

Using a key derivation algorithm such as PBKDF2 increases the time cost of brute force attacks on the limited key space and adds resistance to precomputed rainbow tables. However, this must be scaled with the reduction of cost and increase in speed of computation over time. The possibility of performance degradation must also be investigated, especially in a mobile application running on older hardware that must derive many keys (e.g. in a gallery view).

Remediation Recommendations

Conduct performance and usability testing and determine the feasibility of increasing the count of iterations to 100000 or 10000 from 1000.

Remediation Status

PBKDF2 iterations increased to 100,000 in *mobile/src/main/java/rs/readahead/washington/mobile/data/provider/EncryptedFileProvider.java*. It does not appear to have been implemented in the other locations noted; though this may be because those are to be deprecated.

Additional Information

- Wikipedia: Key derivation function
- Wikipedia: Key stretching
- Cryptosense: Parameter choice for PBKDF2

V-003: Tella iOS Cleartext Audio Data may Persist Longer than Required

Severity	Remediation
Low	Resolved

Note: The Tella development team has conducted a review of where temporary files may persist in Tella iOS and have implemented proactive clearing in: media file importation, audio recording viewing, and audio recording.

Discussion

Camera and audio media data is written to temporary location *NSTemporaryDirectory()* in cleartext at various points, such as while being recorded or are being played. This location is proactively cleared using methods such as *VaultManager.clearTmpDirectory()*.

Other calls to clear the directory are made at other points in the application lifecycle, when dealing with non-audio media files that exist in cleartext temporarily during writing or playback.

For example, a proactive clearing of the temporary directory is included in *Scenes/Camera/ViewModel/CameraViewModel.swift*:

```
mainAppModel.vaultManager.progress.progress.sink { value in
    if value == 1 {
        mainAppModel.vaultManager.clearTmpDirectory()
    }
}.store(in: &cancellable)
```

A similar directory clearing is performed in *Scenes/HomeView/Views/FileDetailView/Video/VideoViewer.swift*:

```
[..]
.onDisappear {
    appModel.vaultManager.clearTmpDirectory()
}
```

However, at the location where one might expect it, no such similar invocation exists in the corresponding *Scenes/Audio/AudioPlayer/Views/AudioPlayerView.swift*:

```
.onDisappear {
    self.viewModel.onStopPlaying()
}
```

Subgraph reviewed all of the views related to playing and saving audio files and did not see any proactive clearing of the temporary directory, even though *NSTemporaryDirectory()* is used during audio recording. It is recommended that Tella iOS developers investigate and determine whether or not the audio management should behave as do the other media management functionality within the app.

Impact Analysis

It may be possible for a privileged adversary with physical access to the device to recover cleartext media data if the app is very suddenly terminated during recording or playback, and not restarted. This will likely always be possible, but there may reduced likelihood if there is gratuitous and/or opportunistic removal of temporary (cleartext) files.

Remediation Recommendations

Subgraph recommends that Tella iOS developers evaluate the risk and implement proactive temporary file removal if deemed worthy to do so.

Remediation Status

Additional proactive temporary file removal has been implemented in three locations: media file import functionality, the audio recording viewer, and the audio recorder.

Additional Information

N/A

V-004: Tella Android Outdated Retrofit2 Dependency

Severity	Remediation
Low	Resolved

Discussion

Retrofit2 is a Tella Android third-party dependency used as an HTTP client in the forms functionality. The current version is 2.9.0, while the version included with Tella is 2.6.1.

The following was observed in the develop branch as of commit #36bd984:

```
./build.gradle:        retrofitVersion = '2.6.1'
```

Impact Analysis

There is at least one outstanding CVE in a dependency of Retrofit 2.6.1:

- CVE-2017-1000190: This is an alleged vulnerability affecting simplexml-safe. The vulnerability is contested, and it is not clear that this issue is exploitable in Tella. Subgraph attempted to reference external entities in hosted form documents but this was not successfully accomplished.

Remediation Recommendations

Consider an upgrade to the latest version of the Retrofit2 bundle. Evaluate possibility of exploitable vulnerability and use of “patched” simplexml-safe.

Remediation Status

Retrofit upgraded to version 2.9.0 in Tella Android v2.0.10.

Additional Information

N/A

V-005: Tella Android Deprecated Dependency: Butterknife

Severity	Remediation
Informational	In Progress

Note: Remediation of this issue is in progress, with views migrated to ViewBinding over coming releases.

Discussion

One of the dependencies was identified as deprecated. *Bufferknife* annotations are used throughout the Tella Android codebase. The repository README states that there will not be updates or bug fixes:

```
Attention: This tool is now deprecated. Please switch to view binding.
Existing versions will continue to work, obviously,
but only critical bug fixes for integration with AGP will be
considered. Feature development and general bug fixes have stopped.
```

It is not clear what type of security vulnerability could exist; it does not even seem probable. However, as it is officially deprecated third-party code, it should be scheduled for transition out of the Tella Android source tree.

Impact Analysis

Security or bug fixes may not be made available. Latent vulnerabilities may be exploited with no fixes available.

Remediation Recommendations

The README recommends a transition to Android native View Binding.

Remediation Status

Not yet resolved.

Additional Information

N/A

Appendix

Methodology

Our approach to testing is designed to understand the design, behavior, and security considerations of the assets being tested. This helps us to achieve the best coverage over the duration of the test.

To accomplish this, Subgraph employs automated, manual and custom testing methods. We conduct our automated tests using the industry standard security tools. This may include using multiple tools to test for the same types of issues. We perform manual tests in cases where the automated tools are not adequate or reveal behavior that must be tested manually. Where required, we also develop custom tools to perform tests or reproduce test findings.

The goals of our testing methodology are to:

- Understand the expected behavior and business logic of the assets being tested
- Map out the attack surface
- Understand how authentication, authorization, and other security controls are implemented
- Test for flaws in the security controls based on our understanding
- Test every point of input against a large number of variables and observe the resulting behavior
- Reproduce and re-test findings
- Gather enough supporting information about findings to enable us to classify, report, and suggest remediations

Description of testing activities

Depending on the type and scope of the engagement, our methodology may include any of the following testing activities:

1. **Information Gathering:** Information will be gathered from publicly availble sources to help increase the success of attacks or discover new vulnerabilities
2. **Network discovery:** The networks in scope will be scanned for active, reachable hosts that could be vulnerable to compromise
3. **Host Vulnerability Assessment:** Hosts applications and services will be assessed for known or possible vulnerabilities
4. **Application Exploration:** The application will be explored using manual and automated methods to better understand the attack surface and expected behavior
5. **Session Management:** Session management in web applications will be tested for security flaws that may allow unauthorized access
6. **Authentication System Review:** The authentication system will be reviewed to determine if it can be bypassed
7. **Privilege Escalation:** Privilege escalation checks will be performed to determine if it is possible for an authenticated user to gain access to the privileges assigned to another role or administrator

8. **Input Validation:** Input validation tests will be performed on all endpoints and fields within scope, including tests for injection vulnerabilities (SQL injection, cross-site scripting, command injection, etc.)
9. **Business Logic Review:** Business logic will be reviewed, including attempts to subvert the intended design to cause unexpected behavior or bypass security controls

---

Reporting

Findings reports are peer-reviewed within Subgraph to produce the highest quality findings. The report includes an itemized list of findings, classified by their severity and remediation status.

Severity ratings

Severity ratings are a metric to help organizations prioritize security findings. The severity ratings we provide are simple by design so that at a high-level they can be understood by different audiences. In lieu of a complex rating system, we quantify the various factors and considerations in the body of the security findings. For example, if there are mitigating factors that would reduce the severity of a vulnerability, the finding will include a description of those mitigations and our reasoning for adjusting the rating.

At an organization's request, we will also provide third-party ratings and classifications. For example, we can analyze the findings to produce *Common Vulnerability Scoring System* (CVSS)[1] scores or *OWASP Top 10*[2] classifications.

The following is a list of the severity ratings we use with some example impacts:

Critical
Exploitation could compromise hosts or highly sensitive information

Critical Exploitation could compromise hosts or highly sensitive information

High
Exploitation could compromise the application or moderately sensitive information

High Exploitation could compromise the application or moderately sensitive information

Medium
Exploitation compromises multiple security properties (confidentiality, integrity, or availability)

Medium Exploitation compromises multiple security properties (confidentiality, integrity, or availability)

[1] https://www.first.org/cvss/
[2] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Low

Exploitation compromises a single security property (confidentiality, integrity, or availability)

Low Exploitation compromises a single security property (confidentiality, integrity, or availability)

Info

Finding does not directly pose a security risk but merits further investigation

Info Finding does not directly pose a security risk but merits further investigation

The severity of a finding is often a product of the impact to general security properties of an application, host, network, or other information system.

The properties that can be impacted are:

Confidentiality Exploitation results in authorized access to data

Integrity Exploitation results in the unauthorized modification of data or state

Availability Exploitation results in a degradation of performance or an inability to access resources

The actual severity of a finding may be higher or lower depending on a number of other factors that may mitigate or exacerbate it. These include the context of the finding in relation to the organization as well as the likelihood of exploitation. These are described in further detail below.

Contextual factors

Confidentiality, integrity, and availability are one dimension of the potential risk of a security finding. In some cases, we must also consider contextual factors that are unique to the organization and the assets tested.

The following is a list of those factors:

Financial Exploitation may result in financial losses

Reputation Exploitation may result in damage to the reputation of the organization

Regulatory Exploitation may expose the organization to regulatory liability (e.g. make them non-compliant)

Organizational Exploitation may disrupt the operations of the organization

Likelihood

Likelihood measures how probable it is that an attacker exploit a finding.

This is determined by numerous factors, the most influential of which are listed below:

Authentication Whether or not the attack must be authenticated

Privileges Whether or not an authenticated attacker requires special privileges

Public exploit Whether or not exploit code is publicly available

Public knowledge Whether or not the finding is publicly known

Exploit complexity How complex it is for a skilled attacker to exploit the finding

Local vs. remote Whether or not the finding is exposed to the network

Accessibility Whether or not the affected asset is exposed on the public Internet

Discoverability How easy it is for the finding to be discovered by an attacker

Dependencies Whether or not exploitation is dependant on other findings such as information leaks

Remediation status

As part of our reporting, remediation recommendations are provided to the client. To help track the issues, we also provide a remediation status rating in the findings report.

In some cases, the organization may be confident to remediate the issue and test it internally. In other cases, Subgraph works with the organization to re-test the findings, resulting in a subsequent report reflecting remediation status updates.

If requested to re-test findings, we determine the remediation status based on our ability to reproduce the finding. This is based on our understanding of the finding and our awareness of potential variants at that time. To reproduce the results, the re-test environment should be as close to the original test environment as possible.

Security findings are often due to unexpected or unanticipated behavior that is not always understood by the testers or the developers. Therefore, it is possible that a finding or variations of the finding may still be present even if it is not reproducible during a re-test. While we will do our best to work with the organization to avoid this, it is still possible.

The findings report includes the following remediation status information:

Resolved
Finding is believed to be remediated, we can no longer reproduce it

Resolved Finding is believed to be remediation, we can no longer reproduce it

In progress
Finding is in the process of being remediated

In progress Finding is in the process of being remediated

Unresolved
Finding is unresolved – used in initial report or when the organization chooses not to resolve

Unresolved Finding is unresolved – used in initial report or when the organization chooses not to resolve

Not applicable
There is nothing to resolve, this may be the case with informational findings