Fejlessz biztonságos alkalmazást

programozási minták fejlesztőknek

Attila Polacsek
Senior Android Developer | Supercharge

Csaba Kozák
Android Tech Lead | Supercharge

Security techniques

API protection

SUPER
CHARGE

API protection - Simple app id key

- Sent in every request
- URL-s are often logged
  - Never put your key in the url
  - `Authorization: key some-client-id`
- Vulnerable to MITM attacks
- Unique per app, hard to replace

SUPER
CHARGE

Security techniques

Secure the communication channel

SUPER CHARGE

Secure communication - HTTP vs HTTPS

- HTTP
  - Plain text
  - Easy to obtain and view the data by third party
- HTTPS
  - Stands for HTTP Secure
  - Used with SSL / TLS
  - TCP socket channel is encrypted

Secure communication - HTTPS

- SSL
  - Secure Socket Layer
  - SSLv3.0 21 years old
  - v2.0 was prohibited in 2011 by RFC 6176 and v3.0 followed in 2015
- TLS
  - Transport Layer Security
  - Successor of SSL, basically TLSv1.0 is SSLv3.1
  - Use the latest version to maximize security
    - TLSv1.0 supported since Android 1 and iPhone OS 1
    - TLSv1.1, TLSv1.2 supported since Android 5 Lollipop and iOS 5

SUPER CHARGE

Secure communication - OkHttp

- Powers HttpUrlConnection since Android 4.4
- Use MODERN_TLS connection spec (it’s the default)
- It has a COMPATIBLE_TLS fallback
- SSLv3.0 is not supported since OkHttp 2.2

https://twitter.com/jakewharton/status/482563299511250944

```java
// create a custom connection spec (TlsVersion.TLS_1_2 requires Android 5+)

ConnectionSpec spec = new ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS)
        .tlsVersions(TlsVersion.TLS_1_2)
        .cipherSuites(
                CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
                CipherSuite.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
                CipherSuite.TLS_DHE_RSA_WITH_AES_128_GCM_SHA256)
        .build();

OkHttpClient client = new OkHttpClient.Builder()
        .connectionSpecs(Collections.singletonList(spec))
        .build();
```

Secure communication - MITM

- Technique to read HTTP or plain socket communication
- Attacker can view, redirect or repeat the requests and responses
- 4 common ways to intercept network traffic
  - Fake WiFi or cell tower
  - ARP (Address Resolution Protocol) spoofing
  - Hostile proxies / SSL bump
  - Malicious VPN
- Burp suite, mitmproxy

Secure communication - mitmproxy

- We will use mitmproxy in transparent
  - Transparent mode: monitors traffic at network level
  - Not all apps can use global proxy settings on Android
- How
  - Enable TCP forwarding on the host machine
  - Route web ports through 8080 which is our default port
  - Start up mitmproxy in web mode: `sudo mitmweb -T --host`

SUPER CHARGE

Demo

mitmproxy

SUPER CHARGE

Demo

mitmproxy

Demo

mitmproxy

SUPER
CHARGE

Secure communication - CERT Pinning

- Leaf certificate
- Intermediate certificate
- Root certificate

SUPER CHARGE

```java
CertificatePinner certPinner = new CertificatePinner.Builder()
       .add("api.github.com", "sha256/VRtYBz1boKOXjChfZYssN1AeNZCjywl77l2RTl/v380=")
       .build();

OkHttpClient client = new OkHttpClient.Builder()
       .certificatePinner(certPinner)
       .build();
```

Demo

mitmproxy

Security techniques

API protection

API protection - Prevent API call tampering

- Shard API key to an ID and a shared secret
- App ID is in every request
- Sign request with the shared secret
  - Compute a message authentication code (MAC) with eg. HMAC SHA-256 algorithm
- Send MAC in every request
  - `Authorization: HMAC-SHA256 my-api-id my-hmac`

SUPER CHARGE

API protection - Prevent API call tampering

- Secrets are static constants
- Use code obfuscator to make it harder to locate and extract
- Encode it with some computationally simple way
- Distribute it around the binary
- Reassemble if needed
- Never save it in persistent storage

SUPER CHARGE

```java
// Somewhere in the code
byte[] encodedSecret = {'S', 'e', 'c', 'r', 'e', 't'};

// Somewhere else in the code
byte[] decodingKey = {'K', 'e', 'y'};

// Just before using the secret
byte[] clearSecret = decode(encodedSecret, decodingKey)

// Use the secret key to generate the signature for the API request
String signature = HMAC(clearSecret, message);
```

API protection - Handle User credentials

- Client sends credentials
- Server validates and sends back a session key
- If session last longer than the app instance, persist it
  - Keychain Services on iOS
  - SharedPreferences on Android

API protection - Handle User credentials

- Resource owner (aka the User)
- Resource server (aka the API server)
- Client
- Authorization server
- Grant types
  - Client credentials
  - Authorization code
  - Refresh token

SUPER CHARGE

API protection - Switch to Authorization Token

- Return access token instead of a session key
- They look similar and used the same way, but the content differ
- Access token is represented as JSON Web Token (JWT)
  - Common claims
    - "iss" - identifies who issued the token
    - "sub" - the principal subject of the claims, often the User
    - "aud" - the intended audience for the claims, often the Server
    - "exp" - the expiration timestamp of the claims
  - Also called bearer token and passed with every API call

API protection - Shorten token lifetimes

- Customizable expiration time
- Can be replaced with refresh token

SUPER
CHARGE

API protection - Authenticate the App, not just the User

- Authorization is split into two steps
- Resource owner authorization
  - Authorization code is returned
- Client authorization
  - Authorization code and client secret are exchanged for tokens

API protection - Remove the Client Secret

- Client secret is statically stored, like the app key was
- We can remove it just like we removed the signing secret
- Client authorization step
  - Send a request with the app's unique characteristics
  - Receive the client secret from the server in the response

SUPER
CHARGE

API protection - SLA

- Multi factor authentication
- Receive an SMS or use an RSA type token
- Authorization step
  - Send credentials and receive authorization code
  - Ask for the second, one time pass
  - Send a request with the code, the OTP and the client secret
  - Receive the token

API protection - Token storage

- AccountManager service
- Encrypted SharedPreferences

SUPER CHARGE

API protection - Encrypted token

- Encrypt with
  - Users PIN (with PBKDF2)
  - Android Keystore entry (from API 18)
  - Users fingerprint (from API 21)
    - Use only the official SDK provided by the Android Framework
    - Others eg. Samsung Pass are not secure

SUPER CHARGE

API protection - Encrypted storage

- Realm
  - 64 byte key with AES-256 encryption
  - Encryption key must be provided by us
- SqlCipher
  - 64 byte key with AES-256 encryption
  - Key is derived from a passphrase provided by us

SUPER CHARGE

```java
// key is a 64 item long byte array
RealmConfiguration realmConfiguration = new RealmConfiguration.Builder()
        .encryptionKey(key)
        .build();

Realm realm = Realm.getInstance(realmConfiguration);
```

```java
FlowManager.init(new FlowConfig.Builder(this)
        .addDatabaseConfig(new DatabaseConfig.Builder(ExampleDatabase.class)
                .openHelper((databaseDefinition, helperListener) ->
                        new SQLCipherOpenHelper(databaseDefinition, helperListener) {
                            @Override
                            protected String getCipherSecret() {
                                return "passphrase";
                            }
                        })
                .build())
        .build());
```

Security techniques

Storage

Storage - Intro

- Most secure is to not store anything :)
- Most apps need to store data
- Multiple ways to store data on Android

SUPER
CHARGE

Storage - Internal vs. external

- The naming is rather confusing
- Does not mean device storage vs. SD card

- Internal storage: only the owner application can access it
- External storage: all apps can access itt

- Internal storage can be on the SD card
- External storage can be on the device storage

SUPER CHARGE

Storage - Sandbox

- Android apps run in a sandbox
- Does not access data / services outside its sandbox
- To do so, it must require permissions from the user
- This means other apps cannot access our app’s data
- Unix file permission to enforce this

SUPER
CHARGE

Storage - Sandbox cont'd

- Each app has its own unix user group
- The group is created during app installation

```
# cat /data/system/packages.list | grep supercharge
io.supercharge.securityworkshop 10085 1
/data/user/0/io.supercharge.securityworkshop default:targetSdkVersion=26 3003

# ls -lha | grep grep supercharge
drwx------   5 u0_a85 u0_a85 4.0K io.supercharge.securityworkshop
```

SUPER CHARGE

Storage - Internal storage

- The path is something like this: `/data/data/io.supercharge.securityworkshop/files`
- To retrieve: `context.getFilesDir()`
- Only the application can access these files
- Even the user does not access these
- Uninstalling the app deletes it

Storage - Internal storage cont'd

- Debug mode allows accessing it

```
$ run-as io.supercharge.securityworkshop cat
/data/data/io.supercharge.securityworkshop/files/hello
Hello world
```

- But this is not possible in release apps:

```
$ run-as io.supercharge.securityworkshop cat
/data/data/io.supercharge.securityworkshop/files/hello
run-as: package not debuggable:
io.supercharge.securityworkshop
```

- Never publish debuggable app!

SUPER CHARGE

Storage - External storage (private)

- External storage is not always accessible
- `Environment.getExternalStorageState()`
- Path is something like this: `/storage/emulated/0/Android/data/io.supercharge.security workshop/files`
- To retrieve it: `context.getExternalFilesDir(null)`
- These file should be private to the application
- Deleted during app uninstallation
- No security restriction
- Do not store sensitive data here!

SUPER CHARGE

Storage - External storage (public)

- Path is something like this:
  `/storage/emulated/0/Download`
- To retrieve it:
  `Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)`
- Shared files
- Stays after app uninstallation
- These files could be from anyone
- We should perform input validation
- Verify loading dynamic libraries

Storage - SharedPreferences

- To store key-value pairs
- You should only store simple data
- The was a world readable option before
- Now it is deprecated, use `Context.MODE_PRIVATE`

SUPER
CHARGE

Storage - SharedPreferences cont'd

- Path is something like this:
  `/data/data/io.supercharge.securityworkshop/shared_prefs`
- It is under internal storage
- But these are plain text files!

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
   <string name="key">sensitive</string>
</map>
```

- There are encrypted alternatives
- Only effective, if uses external password

SUPER CHARGE

Storage - File system encryption

- From Android 5.0, the system encrypts the files by default
- Full Disk Encryption (FDE)
- Prompts for password before boot

- New technique since 7.0
- File-based Encryption (FBE)
- Direct Boot
- Credential Encrypted Storage - available after first password
- Device Encrypted Storage
  - useful for example for phone, alarm, etc.

Storage - File system encryption cont’d

- Only available if users set passcode
- Encryption keys are claimed during first passcode prompt
- Stays in the RAM until reboot
- Lock screen does not evict the encryption keys
- You have to implement it manually using KeyStore

SUPER
CHARGE

Coffee break

See you in 15 minutes

SUPER
CHARGE

Security techniques

Binary protection

SUPER
CHARGE

Binary protection - intro

- Our APK can be retrieved by third party
- Google Play does not provide the APK
- But there are several ways to get it
- Google Play crawling
- apkmirror.com , apkpure.com
- Some countries does not even have Google Play

SUPER
CHARGE

Binary protection - intro cont’d

- We should know how the APKs are built, to protect them
- Android app binaries are APK files
- Actually these are simple zip files
- Anybody can explode them

Binary protection - AndroidManifest.xml

- Contains app meta-data
- App package name
- Activity, Services, ContentProviders
- Permissions
- Is the app debuggable?

```
$ aapt dump xmltree my-app.apk  AndroidManifest.xml
$ aapt dump badging my-app.apk
```

SUPER CHARGE

Binary protection - res

- res folder
- All Android resource files
- JPG, PNG files
- XML resources - in binary form
- XML drawables
- Layout files

SUPER
CHARGE

Binary protection - resources.arsc

- Basically a big table
- Value resources are being put here
- Color
- Dimen
- ID
- Integer
- String

SUPER
CHARGE

Binary protection - classes.dex

- The actual source code can be found here
- Dalvik bytecode format
- Program code and Java all libraries
- Multi-dex -> classesN.dex

SUPER CHARGE

Binary protection - APK signature

- Identifies the developer
- APK integrity
- JAR signing v1 scheme
- APK Signature Scheme v2 (v2 scheme)
- Since Android 7.0
- Backwards compatibility

SUPER CHARGE

Binary protection - analyzing APK

- Android Studio
- Build → Analyze APK

SUPER CHARGE

Demo

Android Studio APK analyzer

Binary protection - jadx

- https://github.com/skylot/jadx
- GUI tool
- Decompiles bytecode to human-readable Java code
- Also decompiles resources

SUPER CHARGE

Demo

jadx

Binary protection - apktool

- https://github.com/iBotPeaches/Apktool
- APK reverse engineering tool
- Disassembly APK
- Decompiles Dalvik bytecode to Smali code

```
$ java -jar apktool_2.3.0.jar d workshop.apk
```

SUPER
CHARGE

Demo

apktool

Binary protection - rebuilding APK

- `$ java -jar apktool_2.3.0.jar b workshop`

- `$ adb install workshop.apk`
  `Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES]`

- APK must be signed
- JAR signing v1 scheme
  ```
  $ jarsigner -sigalg SHA1withRSA -digestalg SHA1
  -keystore release.keystore workshop.apk alias_name
  ```
- APK Signature Scheme v2
  ```
  $ apksigner sign --ks release.keystore --out
  workshop-signed.apk workshop.apk
  ```

SUPER CHARGE

Demo

rebuilding APK

SUPER CHARGE

Binary protection - obfuscation

- As we can see, source code can be easily reverse-engineered
- And also easily modified
- We could make this harder, by introducing obfuscation tools
- Multiple options on Android

Binary protection - ProGuard

- Default code obfuscation tool
- Comes with the Android Gradle Plugin
- Must be configured
- Also contains optimizer and byte code preverifier
- Does not touch resources
- Mapping should be retained to retrace later

SUPER CHARGE

Binary protection - ProGuard configuration

- Configuration in proguard.cfg
- Libraries: consumerProguardFiles
- Developers really hate this tool
- Reflectively accessed code must be kept
- We should keep the smallest numbers of classes

Binary protection - ProGuard directives

- `-keep`
- `-keepclassmembers`
- `-keepnames`
- `-keepclassmembernames`
- `-keepclasseswithmembers`
- `-keepclasseswithmembernames`

SUPER CHARGE

Demo

disassembly obfuscated code

SUPER
CHARGE

Binary protection - APK integrity checks

- Check if APK debuggable

```
boolean debuggable = 0 != (getApplicationInfo().flags &
ApplicationInfo.FLAG_DEBUGGABLE);
```

- Check APK signatures

```
PackageManager pm = getPackageManager();
PackageInfo info = pm.getPackageInfo(getPackageName(),
PackageManager.GET_SIGNATURES);

for (Signature sig : info.signatures) {
   if (!sha256(sig.toByteArray()).equals(SIGNATURE) {
       // stop the app
   }
}
```

SUPER CHARGE

Binary protection - Other tools

- https://www.guardsquare.com/en/dexguard
- https://dexprotector.com/

- Not free - rather expensive
- Control flow obfuscation
- Class, resource encryption
- Runtime self-protection

SUPER CHARGE

Security techniques

Root protection

SUPER
CHARGE

Root protection - Rooting intro

- The Android operation system provides lots of security features
- Rooting enables the user to run as root user
- These of security features will not be available
- For example: internal storage is not private to the app anymore
- We can try to check whether the user is running on an unprotected environment

SUPER CHARGE

Root protection - Root checks

- There are simple libraries to indicate root
- https://github.com/scottyab/rootbeer

- Availability of cloaking apps
- Availability of apps with root access
- Availability of busybox
- Availability of su

- However, these checks can be easily defeated.

SUPER CHARGE

Root protection - SafetyNet

- Google's attestation API
- Comes with Google Play Services
- Cannot work on non-Google Play devices
- Updated automatically
- Free, but has quota

SUPER CHARGE

Root protection - SafetyNet internals

- `snet` service collects the data
- Sends back to Google
- `snet` is not in any APK
- Updated regularly
- It has lots of checks

Root protection - Using SafetyNet

1. The app requests a nonce from the trusted server
2. The app calls the SafetyNet
3. SafetyNet returns the result in JWS
4. The app should send this to the trusted server for verification
5. The server returns the final result
6. The app can resume its services

SUPER CHARGE

Root protection - SafetyNet results

- `ctsProfileMatch`:
  - Certified, genuine device that passes CTS
- `basicIntegrity`:
  - Certified device with unlocked bootloader
  - Genuine but uncertified device, such as when the manufacturer doesn't apply for certification
  - Device with custom ROM
- No `basicIntegrity`:
  - Emulator
  - Protocol emulator script
  - Signs of system integrity compromise, such as rooting
  - Signs of other active attacks, such as API hooking

Root protection - SafetyNet caveats

- Use the latest library
- Generate the nonce on server side
- Create big nonce, using secure random number generator
- Verify the results on the server, not in the app
- Do not use the test attestation verification service for production
- Check nonce, timestamp, APK name, and hashes

SUPER CHARGE

Demo

SafetyNet

Security techniques

Sensitive data in memory

Sensitive data in memory - intro

- Sensitive data should be in the memory in the smallest window
- Generally, passwords are used as `String` objects
- But `Strings` are immutable
- We cannot remove them from the memory
- Therefore we should use a mutable data structure with more control

SUPER CHARGE

Sensitive data in memory - EditText

```
int length = passwordView.length();
char[] password = new char[length];
passwordView.getText().getChars(0, length, pd, 0);

// use password

Arrays.fill(password, ' ');
```

SUPER
CHARGE

QA

Thanks for your attention!

Contact us!

Attila Polacsek
Senior Android Developer | Supercharge

Csaba Kozák
Android Tech Lead | Supercharge

SUPER
CHARGE