MOBILE APPS SECURITY

Prof. Dr. Bernhard Tellenbach

Content

- Introduction to security in mobile apps
- Discussion of security-relevant aspects of mobile apps focusing on six different categories
  - Data Storage and Data Leakage
  - Secure Communication
  - Authentication and Authorization
  - Inter-App Communication
  - Client-Side Injection
  - Reverse Engineering
- For all security categories, we look at examples what can go wrong and describe best practices that should be followed to develop secure mobile apps

Goals

- You know and understand what can go wrong with respect to security when developing mobile apps
- You are capable of analyzing mobile apps to find vulnerabilities and can exploit the discovered vulnerabilities (see also security lab)
- You know and understand best practices to develop secure mobile apps

Introduction to Security in Mobile Apps

Introduction (1)

- Just like with other types of applications, developers of mobile apps often make security-critical mistakes, which result in vulnerabilities
- With web applications, we have a good de-facto standard to categorize the most critical security problems → OWASP Top 10 project (see SWS1)
- A similar project was established to categorize the most critical vulnerabilities in mobile apps → OWASP Mobile Top 10 project

M1 – Improper Platform Usage	M3 – Insecure Communication	M5 – Insufficient Cryptography	M7 – Client Code Quality	M9 – Reverse Engineering
M2 – Insecure Data Storage	M4 – Insecure Authentication	M6 – Insecure Authorization	M8 – Code Tampering	M10 – Extraneous Functionality

OWASP

- OWASP = Open Web Application Security Project (*http://www.owasp.org*)
- Develops and provides guidelines (best practices) and tools
- Original focus on web applications, but this has been extended
- The OWASP Top 10 list has established itself as a de-facto standard

OWASP Mobile Top 10 - Real-World Study

According to the blog post referenced below, NowSecure tested apps from the Apple app store and the Google Play store and found out that 85% of apps violate at least one top 10 risk.

Of these apps 50% have insecure data storage and almost the same number of apps use insecure communication. For a more detailed statistics, check out

Source: https://www.nowsecure.com/blog/2018/07/11/a-decade-in-how-safe-are-your-ios-and-android-apps/

Introduction (2)

- The OWASP Mobile Top 10 project is not as mature as the OWASP Top 10 one
  - There's redundancy among the categories and it's questionable whether the right security categories are included
- Therefore, we are using an adapted version in this chapter
  - Still based on the OWASP Mobile Top 10 project
  - But the categories are reorganized, we include some security problems which are not on the current list, and we ignore some security problems from the list
- The security categories discussed in this chapter:

- OWASP Mobile Top 10 wasn't Updated since 2016 -> Latest Release https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10

Data Storage and Data Leakage

Data Storage and Data Leakage

- This category deals with storing sensitive data on the mobile device in a secure way and preventing that sensitive data is leaked such that it may be accessible by an attacker
  - Data stored in the file system (files, databases, SD cards,...)
  - Data stored in a buffer of the OS (during copy/paste)
  - Screenshots taken by the user or the OS
  - Data written to the system-wide log
  - Data sent to 3rd party services (e.g. for analytics)
  - ...
- Here we look at the following in detail
  - Storing sensitive data in the file system
  - Preventing data leakage via copy/paste, screenshots, and logs

Storing Credentials – Exercise

- Scenario:
  - You must develop an app for an e-shop
  - To access the own account, users must provide their credentials (username & password)
  - Entering the credentials on a mobile device is cumbersome, so the app should store the credentials after having entered them the first time

- How would you solve this? Is your approach secure enough?

Options to Store Data on the Mobile Device

- 1. Option: Store the data within the app-specific area in the file system, e.g., in folder *Documents* on iOS or *shared_prefs* on Android
  - Other apps on the same device cannot access this area (sandboxing)
  - Requires root rights (services running as root, jailbroken devices, privilege escalation)
  - This data is also included in backups, which exposes it to other attacks
- 2. Option: Store the data in a location that is intended for sensitive data, e.g., the iOS keychain or SecureEnclave or Android's keystore
  - Keychain/keystore: Never included in unencrypted backups, but malware / user / apps with root rights* can still access the content
  - SecureEnclave would be best but works with public-key crypto only
- 3. Option: Store the data in a «shared place», e.g., using the MediaStore API on Android (*or directly on /sdcard, which is unprotected, old Android versions only*)
  - That's obviously a bad choice unless the data should be shared

* **Even with root it can be possible to secure key-material** if you use user authentication for key use. **User Authentication can be coupled with key-material**. More specifically, when generating or importing a key into the AndroidKeyStore you can specify that the key is only authorized to be used if the user has been authenticated. The user is authenticated using a subset of their secure lock screen credentials (pattern/PIN/password, fingerprint).

This is an advanced security feature which is generally useful only if your requirements are that a compromise of your application process after key generation/import (but not before or during) cannot bypass the requirement for the user to be authenticated to use the key. See https://developer.android.com/training/articles/keystore for more details on the two available modes of operation.

SecureEnclave

Keeping a private key in a keychain is a great way to secure it. The key data is encrypted on disk and accessible only to your app or the apps you authorize. However, to use the key, you must briefly copy a plain-text version of it into system memory. While this presents a reasonably small attack surface, there's still the chance that if your app is compromised, the key could also become compromised. As an added layer of protection, you can store a private key in the Secure Enclave.

The Secure Enclave is a hardware-based key manager that's isolated from the main processor to provide an extra layer of security. When you store a private key in the Secure Enclave, you never actually handle the key, making it difficult for the key to become compromised. Instead, you instruct the Secure Enclave to create the key, securely store it, and perform operations with it. You receive only the output of these operations, such as encrypted data or a cryptographic signature verification outcome.

Hence, this approach only works, if authentication is done using a public-key crypto approach. For example, by registering a public key with the web shop and then proving possession of the private key by having it send a challenge to the app and the app then asks the secure enclave to sign the challenge and send it back. So it does not work for our scenario with username/password authentication.

Source:

https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave

/sdcard

Usually, Android devices provide a storage area which is identified as *external storage*, which is typically reached via the directory */sdcard*. This external storage can correspond to a removable storage media (such as an SD card) or to an internal (non-removable) storage. This external storage is used e.g., for downloads or pictures taken by the camera.

On newer Android versions, using */sdcard* with *getExternalStorageDirectory()* directly is deprecated or not possible anymore. To share data among apps, data-specific APIs should be used, for example the MediaStore API for storing and sharing media files. See:

https://developer.android.com/training/data-storage/shared

for details. In general, starting with Android 10, app developers are forced to think more about where and how to store data. This is especially the case when data should be written to an external storage media in a way that other devices can read the data too.

SD Cards as Internal / External Storage

When using an SD card, there are two options to configure it in an Android device. One is to use it as "portable storage". In this case, it's used as external storage, it's (usually) mapped to */sdcard*, and it's accessible by all apps, as described above (except subdirectory */sdcard/Android*). The other option is to use it as "internal storage" to increase the already available internal storage. In this case, it's not mapped to */sdcard* and it's not accessible by all apps.

Adopt external Storage: https://source.android.com/devices/storage/adoptable

It is possible to adopt external storage like sd-cards. When external storage media is adopted, it's formatted and encrypted to only work with a single Android device at a time. Because the media is strongly tied to the Android device that adopted it, it can safely store both apps and private data for all users.

Android Keystore

Android has introduced a keychain with Android Pie:
https://developer.android.com/training/articles/keystore

Example: Android Shared Preferences

- Android Shared Preferences are often used to store sensitive data


```
public void saveCredentials(View paramView)
{
  SharedPreferences.Editor localEditor = PreferenceManager.getDefaultSharedPreferences(this).edit();
  EditText localEditText1 = (EditText)findViewById(2131493000);
  EditText localEditText2 = (EditText)findViewById(2131493001);
  localEditor.putString("user", localEditText1.getText().toString());
  localEditor.putString("password", localEditText2.getText().toString());
  localEditor.commit();
  Toast.makeText(this, "3rd party credentials saved successfully!", 0).show();
}
```

```
root@vbox86p:/data/data/jakhar.aseem.diva/shared_prefs # ls -l
-rw-rw---- u0_a60   u0_a60        169 2016-02-29 06:08 jakhar.aseem.d
ore jakhar.aseem.diva_preferences.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="password">eufQ=&amp;53GlekURb</string>
    <string name="user">superman</string>
</map>
```

Other apps cannot access this file due to the Android sandboxing model

- But it can be accessed with root rights (e.g., malware with root rights and – on jailbroken / rooted devices – by the root user and apps running with root rights)
- And it's included in backups

Storing Sensitive Data – Paranoid Best Practices

- From a pure security point of view, we should therefore recommend the following:
  - Only store non-sensitive data on the device but do not store anything else
  - Don't you ever dare to store credentials on the device!
  - This should guarantee that no sensitive information can be leaked via backups or even to malware with root rights using the 'storage' channel

- But that's simply not realistic given the fact that users expect to use mobile devices and apps in a convenient way
  - Would you like to enter the password whenever checking your e-mail, accessing Facebook, using your TV or newspaper app,… ?
- Also: If users cannot store passwords, the risk that they pick weak (easy-to-type / remember) passwords increases!

Storing Sensitive Data – Realistic Best Practices (1)

- Don't allow the users to store passwords for highly critical services
  - E.g., e-banking, bitcoin wallets, online password stores,…
- With most apps, it's reasonable to store credentials (or other sensitive data) in the file system as the sandboxing model makes it difficult to access this data
  - For this, use e.g., the *Documents* (iOS) or *shared_prefs* (Android) folder
  - Use platform-specific security features such as the iOS or Android keychain
  - On Android, don't store sensitive data on shared storage (e.g., */sdcard*)
- If you like, you can try to further increase protection of this data
  - By encrypting it with a secret that is hardcoded in the app – but this won't increase protection from a determined attacker (→ reverse engineering)
  - By encrypting it with a secret that is supplied by the user at startup – but that's not really user-friendly

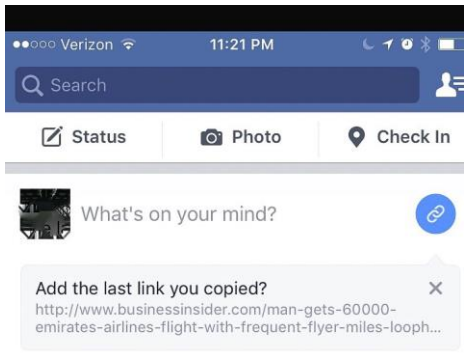
Storing Sensitive Data – Realistic Best Practices (2)

- Nevertheless, it's important that you are aware of the risks of locally storing sensitive data so you can make the right decisions:
  - Malware with root rights can always access all local files – but against such a threat we are doomed anyway*
  - Data is exposed in (unencrypted) backups stored on you PC or in the cloud
  - If the device is jailbroken / rooted, the overall risk increases (many platform security measures no longer apply)
  - When briefly lending a phone to someone else, they can use the apps with stored credentials
    - It may even be possible to root it, extract the credentials, and unroot it again in a matter of minutes
- In practice, usability (convenience) and security must be balanced
  - Yes, there are threats that may allow the attacker to get access to the sensitive data, but in many cases the risks are probably acceptable

*However, if critical key material can't be extracted but for example, the secure enclave has to be used to encrypt/decrypt data, an attacker has to be "online" and persistent on your device to be able to encrypt/decrypt stuff and use the "credential/key" in question in general.

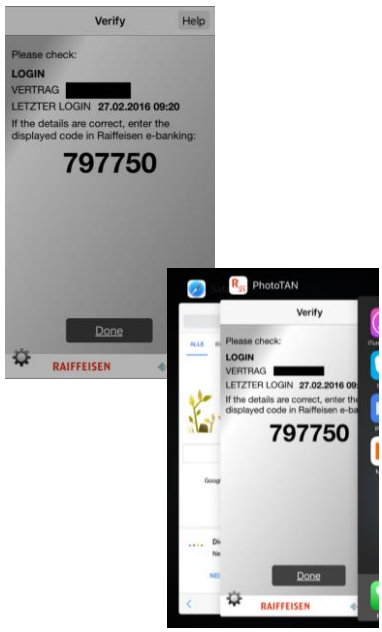
Data Leakage with Copy/Paste

- When data is copied in one app, this data is available to all other apps
  - Otherwise, the copy/paste feature would be much less valuable
- This allows to write a Trojan app that reads data that is currently in the «copy/paste buffer» and that sends this data to the attacker
  - If the attacker is lucky, he gets access to copied passwords or the like, e.g., when the user copies passwords from a password manager app
- Some apps even used this «as a feature»
  - The Facebook app was doing this to «friendly ask» you whether you want to post a copied link
  - Can include data such as credit card numbers, passwords,… in GET parameters



Data Leakage with Screenshots

- Screenshots by the user
  - iOS: Screenshot are stored in the photo library, where they are accessible by other apps (that have the necessary permission)
  - Android: The location where they are stored may vary; they are accessible by other apps (that have the necessary permission)
  - → If a screenshot is made of a login screen or a screen that shows credit card details, this information «may be stolen» by other apps (e.g., by a Trojan app)
- Screenshots by the OS
  - When an app enters the background, the OS takes a screenshot to show it in the list of running apps
  - In contrast to screenshots taken by the user, they are not accessible by other apps, but possible sensitive information is exposed in the file system nevertheless



Android:

On certain devices that use modified Android; the button combination and the storage location can vary.

Taking a Screenshot programmatically: https://stackoverflow.com/questions/2661536/how-to-programmatically-take-a-screenshot

It is possible to take a screenshot if you have access to the view object of your app's activity. From the view object is a BitMap created, which can be saved as screenshot.

Screenshot Prevention:
https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE

To prevent that an activity can be screenshotted by the App itself or any other app you can set the "FLAG_SECURE" -> Then the content of the window is not appearing in screenshots.

getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE, WindowManager.LayoutParams.FLAG_SECURE);

Data Leakage with Logs

- iOS/Android provide logging facilities that can be written by all apps
  - The logs can't be read by apps, but if you get access to an unlocked device, access them with *Xcode* (iOS) or *adb* (Android)
    - Also possible by malware with root rights and on jailbroken / rooted devices
  - → If an app writes sensitive data to the logs, leakage to an attacker is possible
- Example: Mobile banking app logging the credentials (iOS and Android)
  - Forgot to turn it off for the release version

```
<DxiRequest>
        <Header>
                <VendorApp>[redacted]</VendorApp>
                <VendorAppVersion>1371</VendorAppVersion>
                <Language>en</Language>
        </Header>
        <RequestContent>
                <ClientTimestamp>[redacted] 08:04:58</ClientTimestamp>
                <TransactionID>78989078908908</TransactionID>
                <Transaction>DxiLogin</Transaction>
                <DxiLoginRequest>
                        <IdentID>1351421</IdentID>
                        <Password>bWFnYXJvbmk=</Password>
                </DxiLoginRequest>
        </RequestContent>
</DxiRequest>
```

```
06:02:41</ClientTimestamp><TransactionID>78989078908908</TransactionID><Transaction>DxiLo
gin</Transaction><DxiLoginRequest><IdentID>1351421</IdentID><Password>bWFnYXJvbmk=</Passw
ord></DxiLoginRequest></RequestContent></DxiRequest>, method=POST,
authDelegateObject=[redacted]plugins.system.login.ui.LoginResponseHandler@43561648}
```

Preventing Data Leakage – Best Practices

- Disable copying from views used for sensitive data (passwords, credit card,…)
  - Can be done programatically in Android and iOS
- Prevent the OS from taking screenshots for the list of running apps
  - For security-critical apps only
  - Can be done programatically in Android and iOS; as a result, a blank image is used in the list
- Deactive that users can take screenshots in security-critical apps
  - Can be done programatically in Android, impossible in iOS
- Make sure not to write any sensitive data to the logs
  - Don't forget to turn off any logging you enabled during development
- Check what 3rd party libraries you are using are doing
  - Maybe they are logging sensitive data?
  - Maybe they are sending sensitive data to 3rd party services for analytics?

Copy/Paste – Preventing Leakage

To avoid leaking sensitive data from the clipboard, you should make sure that sensitive data cannot end up there at all.

From the point of view of a specific application, data can end up there in two ways:

- Sensitive data is copied from another application and then pasted to your application

- Sensitive data is copied from your application and then pasted to another application

In the first case, there is nothing your app can do to prevent that the data ends up in the clipboard. An app has no way to control copy operations performed by other apps.

In the second case, the copying of sensitive data can be prevented. The way to go here is to prevent copy/cut operations from Views (TextView, EditText etc.). If there are no copy/cut functions, the data can't be extracted and end up in the clipboard. There are multiple ways how this can be done in Android. One way is to disable "long press" on a View. Another one is to delete the copy/cut items from the menu shown when selecting a character string.

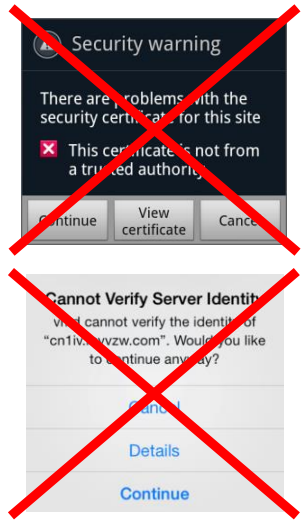
Secure Communication

Secure Communication

- If mobile apps exchange sensitive data with backend servers, the communication should obviously be secured
- As mobile apps usually use HTTP to communicate with servers, the best way to secure the communication is by using TLS (→ HTTPS)
  - With Android 9 Pie and above, HTTPS is enforced, if you want to use HTTP you have to explicitly enable it
  - For apps in the Apple app store, proper exceptions must be included in the information property list file (Info.plist) since January 2017
- In most cases, using TLS is easy as it is supported by all relevant programming languages for mobile apps (Java, Swift, Objective C,...) and by all common server platforms / technologies

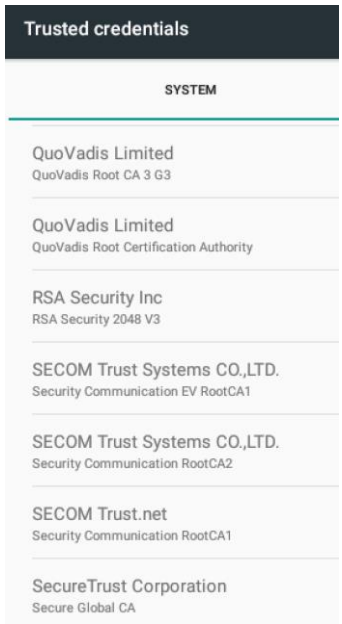
TLS – Best Practices

- Make sure insecure versions (SSL, TLS 1.0,…) and cipher suites that use weak algorithms (DES, RC4, MD5,...) are disabled
  - If you use recent versions of programming languages and runtime environments, this is usually secure per default
- Use server certificates from established certification authorities that are trusted by the mobile platforms
  - As a result of this, there shouldn't be any problems when validating the certificates in the mobile app
- Don't allow to override failed certificate checks
  - Because a failed certificate check can be an indication of an attack, especially if the server uses a certificate from an established certification authority



Certificate Validation in Apps

- To check the validity of a received certificate, mobile OS include a trust store containing certificates of trusted CAs
- An attacker who wants to get a valid certificate for a server used by an app just has to find one CA which is vulnerable (provides a certificate for this server)
  - Allows impersonation of the server or MITM attack
- Question: How can we prevent that the app accepts such a certificate? Does this also introduce problems?



Certificate Pinning (1)

- Option 1: Pinning of the official CA certificate
  - Include the certificate of the CA or a hash thereof in the code
  - After a TLS connection has been set up, additionally compare the certificate that signed the server certificate with the hard-coded CA certificate

- Advantages:
  - To do a MITM attack, the attacker must get a fraudulent certificate from the same CA (not just any CA)
  - If the server gets a new certificate from the same CA, the app code does not have to be updated
- Disadvantage:
  - Requires a new app version if the CA certificate is changed (happens rarely)

Certificate Pinning

For a detailed discussion and code samples see *https://infinum.co/the-capsized-eight/how-to-make-your-ios-apps-more-secure-with-ssl-pinning*

Blacklisting CA's: https://developer.android.com/training/articles/security-ssl

Since Android 4.2 every Android Device has as well a CA's blacklist. The list is dynamically updated in case of a CA breach.

Certificate Pinning (2)

- Option 2: Pinning of the server certificate
  - Include the certificate of the server or a hash thereof into the code
  - Compare the received server certificate with the hardcoded one at connection setup
  - Advantage:
    - The attacker has to get the server's private key or a certificate with the same hash for MITM
  - Disadvantage:
    - Requires a new app version if the server certificate is changed (can happen very often, if for example let's encrypt certificates are used (every 90 days))
- Option 3: Completely rely on an own certificate checking approach
  - Do not perform any standard certificate checking, use a self-signed certificate for the server
  - Include this certificate or a hash thereof in the code
  - Compare the received certificate with the hardcoded certificate
  - This way, you control everything, but cannot rely on any external checks

Which one to use?

All three variants are used in practice. From a security perspective, options one and two are to be preferred over option 3 unless there are very specific reasons for using option 3. Option 3 is discouraged because if you leverage (low-level) TLS and certificate parsing APIs yourself, there is quite some potential for doing it wrong. Security-wise, the second option is probably the best option. It involves two security checks (first with the trust store and then with the pinned certificate) and everything (especially the change of server certificate) is under control of the app provider.

One case where option 3 might be considered (but there are other solutions to that too) is when an app is company-internal only, not published n a public app-store, the server has no connectivity to the Internet and the company does not want to leak any information such as the hostname to a CA and CT log.

Authentication and Authorization

- Assume a cooking app provides innovative recipes for free and corresponding videos as a premium service for a monthly fee
  - To access the premium service, users have to enter username & password
  - The app sends the credentials to the backend server, which checks the correctness and whether the user has paid for the premium service
  - If successful, the app presents «the screen» where the videos can be selected
  - When selecting a video, the video file is requested from the server without including any authentication / authorization information in the request
- Obviously, that's totally insecure, getting access to all videos for free is easy
- But such things happen in practice for different reasons
  - Lack of knowledge on how to do access control on backend (e.g. RESTful web service)
  - Lack of understanding that «not getting access to an app screen doesn't mean the user cannot use the corresponding functionality»

Weak Authentication in Mobile Apps

- Often, mobile apps have less strict authentication requirements e.g.:
  - Password requirements are relaxed to make entering passwords on mobile devices easier (shorter minimal length, no special characters,...)
  - 2nd authentication factors are used less frequently as using them may be cumbersome (e.g. copying a code from an SMS message)
- → Increases the risk that attackers can steal the identities of the users
  - E.g. by guessing a password or by cracking them offline (hashes stolen from the server)
- To prevent this, mobile apps should have similar authentication requirements as what we typically use with web application
  - With passwords, it's much better to require equally strong passwords with mobile apps and store the password for usability reasons (unless they are highly sensitive, e.g. e-banking)

2nd Authentication Factors on Mobile Devices – Exercise

- Access to an e-banking system is by browser or mobile app
  - In both cases, a password and mTAN (login code received by SMS) are used
- Assume Alice always uses the browser (on a personal computer) and Bob always uses the mobile app – which approach is more secure?
  - Assume the goal of the attacker is to get the password and get access to the login codes so he can repeatedly login as Alice / Bob

* Note that there are also arguments why it is more secure to use the mobile device for this.

First, if an attacker places a «Man-in-the-Browser» malware on a victims personal computer, it could just wait until the user authenticated itself. Afterwards, it could make transactions on behalf of the user. Since the malware is in the browser (or MitM), it can manipulate what the user sees – hence, the user might see the expected amount of money on his/her account and this amount is modified correctly depending on the transactions of the user.

However, in contrast to a hack on the mobile device, an attacker cannot access the e-banking system when he/she wants. Access is only possible when the victim itself initiates a session (and enters the mTAN).

To prevent this, transactions should be verified using a second channel (e.g., using transaction signing).

Offline Authentication and Authorization (1)

- A mobile game gives you the first three levels for free and 47 more for money
  - To buy a user-specific access code, you must create an account on the company's website
  - To unlock the levels, username and access code are entered in the app
- To keep the app simple and to make sure it can be used everywhere, the app does not communicate with any server, which implies:
  - The access-restricted content must be available somewhere within the app (in the code, in separate files on the file system,…)
  - The logic to check username+access code to grant access must be «somewhere» in the code
- This is a typical case of offline authentication and authorization
- In general, it's very difficult (probably impossible) to implement offline authentication and authorization in a secure way

Offline Authentication and Authorization (2)

- Fundamental problem: User (attacker) has access to the entire app
  - Code and additional files can be read, analysed, and manipulated; which usually allows to circumvent access control checks
- Example: Assume the access code is build by computing a HMAC over the username of the user: AC = $HMAC_{key}$(username)
  - To check the validity of the access code, the app computes the HMAC over the provided username and compares it with the provided access code
    - This requires the app contains code to do the check and the key
  - The attacker can defeat this in (at least) two different ways:
    - By extracting the key from the code, which allows to create valid access codes
    - By adapting the code (binary patching) to make sure checking the access code is always successful
- Anti-reverse engineering techniques will make the attack more difficult, but this most likely will not stop a committed attacker

Authentication and Authorization – Best Practices (1)

- Use a secure access control mechanism when communicating with the server
  - E.g. RESTful web service: After authentication, send the app an access token that is included in every subsequent request
  - The server uses this access token to identify the user and to check whether he is permitted to execute the corresponding action
- In mobile apps, use similar authentication standards as in web applications
  - Don't allow weaker passwords
  - For increased security, use a 2nd authentication factor, even if the overall security is lower than when using a browser on a personal computer
- Don't use offline authentication and authorization, unless the value of the protected data or functions is low
  - Most likely, local attacks can circumvent all safeguards
  - Use server-side authentication/authorization and provide restricted content from the server

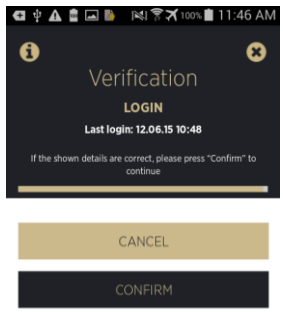
OAUTH2 Example for Secure Authentication on Android for REST API's:
https://developers.google.com/android/guides/http-auth

So far, we looked at mTAN as a 2nd authentication factor on mobile devices, but there are also alternatives:

- Shared secret between app and backend server (low-cost solution)
  - During registration, a secret is stored within the app and on the server
  - During login, username, password, and the shared secret are sent to the server
  - The server verifies username, password, and shared secret with the locally stored secret
  - For increased security, the server sends a new shared secret for the next login
- How does the security of this compare to mTAN?
  - To get the 2nd authentication factor (shared secret), the attacker must have access to the device (physically or via malware)
  - This is comparable to mTAN where device-access also allows them to get the mTAN codes

Authentication and Authorization – Best Practices (3)

- Using a separate authentication app (more expensive option that is used by some e-banking systems)
  - After checking username & password, the server sends the banking app a challenge, which can only be solved by the authentication app
  - This requires that this app has been personalized before (e.g. with a key)
  - The banking app sends the challenge to the authentication app (with inter-app communication), where the action is confirmed by the user
  - This computes the response, which is forwarded via banking app to the server



- How does the security of this compare to mTAN?
  - To get the 2nd authentication factor (use the app or get the key), the attacker must again have access to the device (physically or via malware)
  - This is comparable to mTAN where device-access also allows them to get the mTAN codes

If we look at it in a bit more details, there are some clear advantages of the app aproach over the mTAN approach:

- Usability is improved, no copying of content from the SMS to the e-banking app
- A vulnerability in the e-banking app that allows to hijack it is not enough to carry out malicious transactions without the user noticeing

Inter-App Communication

Inter-App Communication

- Both iOS and Android support inter-app communication between different apps
  - Both support URL-based communication, e.g. zhaw://this-is-great-data-i-want-to-send-to-you
  - Android also allows to explicitly call an Activity in another app
- Inter-app communication communication has the following main risks:
  - The receiving application processes the received data without validating it (see also category client-side injection)
  - The app offers inter-app communication «accidentally», which may allow other apps and the user to access the app in non-intended ways
    - Unlikely in iOS, as the recipient app has to contain code to handle the incoming communication
    - More likely in Android, as only «a little configuration» must be added to *AndroidManifest.xml* and no specific code is required
  - Malicious apps can hijack inter-app communication, which may be critical if sensitive data is exchanged

Inter-App Communication in Android (1)

- Activity, Service or Content Provider must be registered in *AndroidManifest.xml*
- To allow a component to be called by other apps, one does either:
  - Mark them in den Android Manifest with android:exported="true"
    - Other apps can then call the Activity or Service via an explicit intent, using a package- or class-name of the Service or Activity, e.g. com.android.insecurebankv2.PostLogin
  - Add an intent-filter to the Activity or Service
    - The intent-filter defines a URL scheme or an action (e.g. jakhar.aseem.diva.action.VIEW_CREDS); both can be used by other apps to call the activity / service via an implicit intent.

```
<activity android:exported="true" android:label="@string/title_activity_post_login"
    android:name="com.android.insecurebankv2.PostLogin"/>

<activity android:label="@string/d8" android:name="jakhar.aseem.diva.InputValidation2URISchemeActivity"/>
<activity android:label="@string/d9" android:name="jakhar.aseem.diva.AccessControl1Activity"/>
<activity android:label="@string/apic_label" android:name="jakhar.aseem.diva.APICredsActivity">
    <intent-filter>
        <action android:name="jakhar.aseem.diva.action.VIEW_CREDS"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

An Intent is an object that is passed from the calling to the called component

Inter-App Communication in Android (2)

- If «the wrong» Activity or Service is made available to other apps, this can have negative security implications
- Example: An e-banking app requires login before the banking functions can be used

- But due to a copy-paste mistake, the developer has marked the Activity which follows successful login with android:exported="true"

```
<activity android:exported="true" android:label="@string/title_activity_post_login"
    android:name="com.android.insecurebankv2.PostLogin"/>
```

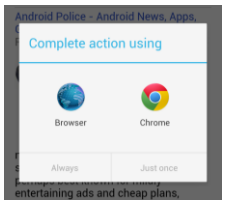
- This means that other apps or users (via *adb*, see security lab) can directly call this Activity, which may give access to app functionality that should be available to authenticated users only

Command to start the exported Activity:

am start -n com.android.insecurebankv2/.PostLogin

Inter-App Communication – Best Practices

- Always validate data received by inter-app communication before processing it
- In Android, only use an intent-filter or *android:exported="true"* if you want to make available the Activity to other apps
- Be careful when exchanging sensitive data with inter-app communica-tion as a malicious app can hijack the data
  - The malicious apps registers for the same URL scheme (e.g. zhaw:///) as the app for which it wants to get data
  - In Android, this opens a dialogue where the user can pick the app; in iOS, the first app registering for a scheme gets the data (since iOS 11)
  - On Android, explicit Intents might be used to avoid the pop-up



- If only specific apps are allowed to communicate with an app, make sure that the receiving app checks the identity of the calling app
  - This is possible with Intents in Android and URL schemes in iOS

iOS inter-app communication

iOS allows one single URL Scheme to be claimed by multiple apps. For instance, Sample:// can be used by two completely separate apps in their implementation of URL Schemes. This is how some malicious apps can take advantage of the URL Scheme and compromise users.

Apple addressed the issue in later iOS versions (iOS 11), where the first-come-first-served principle applies, and only the prior installed app using the URL Scheme will be launched. However, there may still be vulnerabilities - some ways how this can be done and the underlying problems/vulnerabilities are described here:

Source: https://blog.trendmicro.com/trendlabs-security-intelligence/ios-url-scheme-susceptible-to-hijacking/

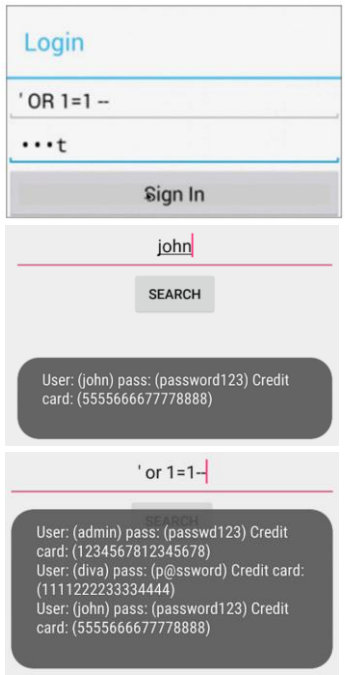
Client-Side Injection

Client-Side Injection

- Number one on the OWASP Top 10 list for web applications are injection attacks on the server-side
  - SQL injection, OS command injection, XML injection,…
- Client-side injection is basically the same – but within the app, on the local device
- The attacks can be performed in two ways:
  - By the user that is directly using the app
  - By other apps that can interact with the vulnerable app (inter-app communication)
- We look a two examples here: SQL injection and local file inclusion

Local SQL Injection (1)

- Both Android and iOS provide functionality to store data in local SQLite databases
  - Databases are usually stored in the app-specific area in the file system and cannot be accessed by other apps
- If the user can specify parts of the SQL query, he may be able to adapt the semantics of the generated query → SQL injection
- Example 1: SQL injection to circumvent a local login
- Example 2: SQL injection to read the entire content of a database



SQLite

SQLite is a relational database management system contained in a C programming library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program. Source: *https://en.wikipedia.org/wiki/SQLite*

Local SQL Injection (2)

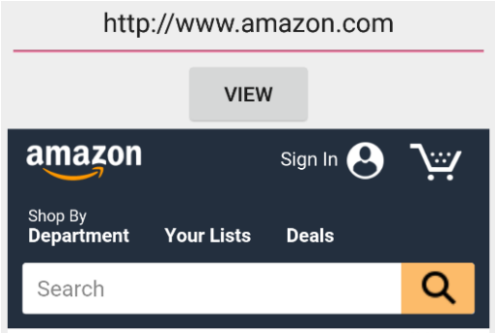
- In general, the risk of local SQL injection is significantly smaller than with server-side SQL injection vulnerabilities
  - In many cases, local databases only contain information known to the user anyway
  - Database content on own device is accessible also in different ways (e.g. with adb)
- But there are also cases where local SQL injection is a security threat
  - If the database is used for local authentication → if you temporarily hand over the phone to someone else, a local SQLi attack might enable circumvention of the authentication
  - If the app accepts requests from other apps (inter-app communication) that result in querrying the database → the other app may use SQLi to access arbitrary data
    - E.g. an app where you maintain all your health-related data and that allows other apps to get some basic medical information about you

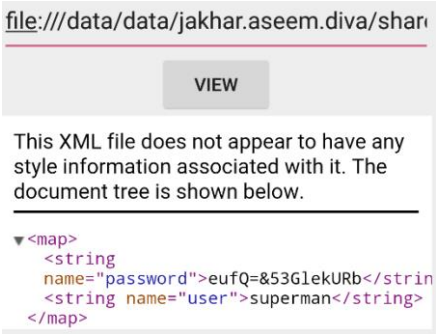
Local File Inclusion (1)

- Local file inclusion may allow the user (or another app) to access local files he should not be allowed to access
- Example:
  - Assume that the user can create files in an app and give them a name
  - As a result, such a file, e.g. *myfile.txt*, is stored below *files/* in the data directory of the app
  - To open the file again at a later time, the user must enter the filename
  - If the filename is not validated, opening files other than those in files/ might be accessed
    - E.g. by entering an absolute or relative path such as ../shared_prefs/com.zhaw.myapp_preferences.xml
    - Assuming the app runs with normal user rights, the user is restricted to the sandbox and "public" storage locations (if any) of the app
- Just like with local SQL injection, the user can access these files anyway, but it gets more critical if this is used with inter-app communication

Local File Inclusion (2)

- Local file inclusion can also happen if an app processes URLs
- Assume a user can enter a http URL and the app gets and displays the content
- If the app does not validate the URL, the attacker can use a file URL to access local files, e.g., the shared preferences
- Again, this is mainly critical in the context of inter-app communication





Preventing Client-Side Injection – Best Practices

- The countermeasures to prevent client-side injection are the same as with server-side injection attacks
- With SQL statements, use prepared statements instead of using string concatenation to generate the queries
- Perform input validation, ideally using a whitelisting approach
  - E.g., for a username: allow at most 10 characters and only lower-case letters
  - E.g., for a file name: allow at most 20 characters, consisting of lower- and uppercase letters and one single dot character
  - With URLs, make sure that only the expected URL schemes (e.g., http and https) are accepted

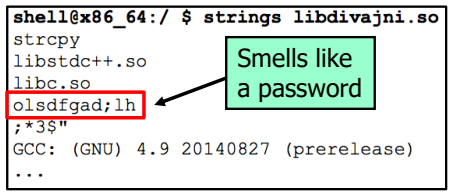
Reverse Engineering

Reverse Engineering

- The goal of reverse engineering is to get a better understanding how the app works internally and to extract hidden information from it
- What's the motivation for an attacker to do reverse engineering?
  - Find hardcoded secrets such as passwords and cryptographic keys (e.g., to crack encrypted files that are stored locally)
  - Find hidden functionality (e.g. one that provides administrative access to the server)
  - Find the location of security functions in the code and determine the functionality with the goal to circumvent them (e.g., offline access control)
  - Gain knowledge of intellectual property (e.g., an ingenious algorithm used by the app)
  - Easily develop an app with similar functionality based on large amounts of code of the original app (can also be a malicious Trojan app)
  - ...

Reverse Engineering – Attacker's Possibilities

- Extract strings that contain printable characters from any file (e.g. a binary)
  - E.g. with the command line tool *strings*
  - Allows to easily identify potential passwords, URLs, commands,...



- Static analysis using decompilers or disassemblers
  - Android Studio, JD-GUI, Hopper, IDA-Pro,...
  - Allows to analyze the internal working, identify relevant functions,...
- Dynamic analysis using debuggers
  - gdb, jdb,...
  - Helps to get a better understanding what happens during program execution, often used in combination with decompilers / disassemblers
- Once the attacker has understood the relevant parts, he can e.g., manipulate the app so that it behaves according to his wishes
  - E.g., «patch» access control or jailbreak / rooted detection code with no-op
  - E.g., change the behavior of methods during runtime (e.g., cycript on iOS, Frida on Android)

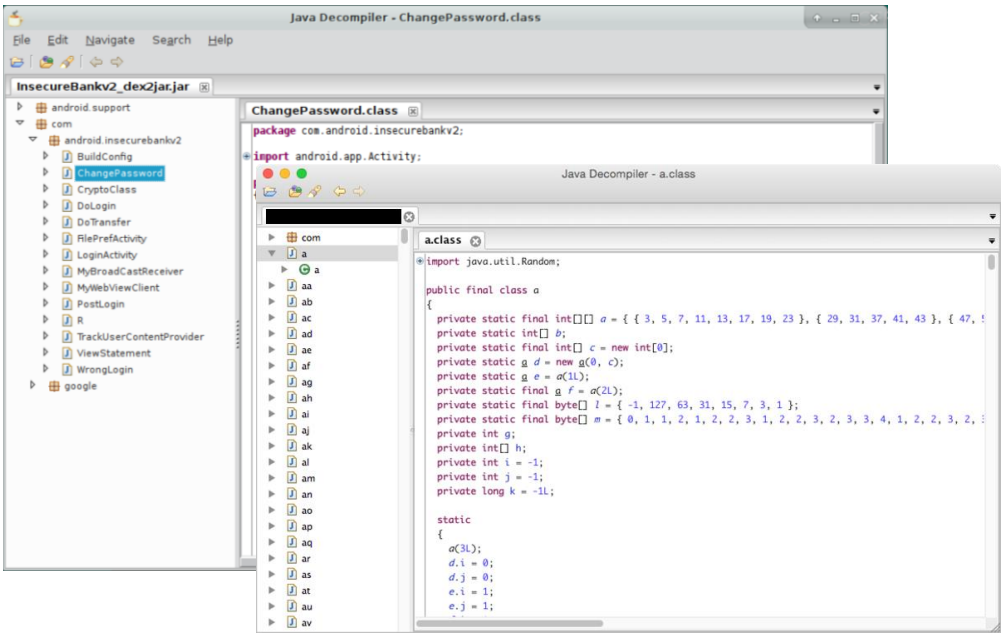
Reverse Engineering – Best Practices (1)

- Disclaimer: No matter what anti-reverse engineering techniques you are using in an app, it will most likely just slow down the attacker and make his life «more miserable» – but a determined attacker will most likely achieve his goals
- You can hardcode cryptographic secrets and passwords in the code, but be aware that they are likely detected by an attacker
  - So they shouldn't be used for really security-critical parts of the app
  - To encrypt some local files to increase their protection «a bit» → OK
  - To protect access to administrative functions on the server → NOK
- For apps that have «a certain value», consider code obfuscation, which makes decompiling and dissassembling significantly harder
  - Especially for Android, as decompiling Dalvik bytecode is easy
  - Objective C / Swift code is harder to decompile due to lots of optimization by the compiler
  - Obfuscators: Android: ProGuard, DexGuard; iOS: iXGuard, PreEmptive ...

Reverse Engineering and Countermeasures Links

- iOS: See e.g. *http://www.raywenderlich.com/45645/ios-app-security-analysis-part-1* and *http://www.raywenderlich.com/46223/ios-app-security-analysis-part-2*
- Android: *https://slides.night-labs.de/AndroidREnDefenses201305.pdf*

Reverse Engineering – Best Practices (2)



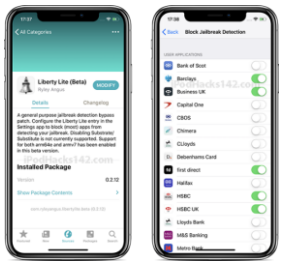
- Decompiling an Android app – Non-obfuscated vs. obfuscated

- To further improve protection, include anti-debugging techniques into the code, which increases the complexity of dynamic analysis
  - iOS: e.g., with macro *SEC_IS_BEING_DEBUGGED_RETURN_NIL()*
  - Android: e.g., with method *isDebuggerConnected()* in class *Debug*
  - Note: A determined attacker can probably remove this with binary patching
- Example: Trying to attach gdb to an app using anti-debugging techniques (iOS)

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=arm-apple-darwin9 --target=".
(gdb) attach Banking
Attaching to process 966.
Segmentation fault: 11
```

Reverse Engineering – Best Practices (4)

- Many techniques can only be used on jailbroken / rooted devices
  - gdb, cycrypt, tools that hook into the app to monitor file system accesses (e.g. Introspy) or to bypass certificate pinning (e.g. SSL kill switch)
- Preventing that an app can be run on a jailbroken / rooted device raises the bar
  - Done by integrating code to detect if the device is jailbroken / rooted, which e.g. checks the file system for the presence of typical files
    - E.g. */Applications/Cydia.app* (iOS), */system/app/Superuser.apk* (Android)
  - There exist free libraries that provide such functionality
- But again, this can usually be circumvented
  - Binary patching of the app code
  - With apps like Liberty Lite (iOS)/RootCloak (Android)
  - The tools hook into the running apps, intercept typical calls that are used to detect jailbroken / rooted devices, and deliver a response expected from a «clean» device



Root resp. jailbreak detection and evading that detection is a cat-and-mouse game.

Both sides have to keep up. However, despite the effort to keep evasion tools up to date, there are still some free options around.

For iOS 12 (and also 13), Liberty Lite is a good option (as of 05/2020). A blog post about that tool (from 2019) can be found here.

This is also the source of the image on the slide.

Source: https://www.ipodhacks142.com/how-to-bypass-jailbreak-detection-on-uk-banking-applications-ios-12/

Summary

- There's a wide spectrum of security-relevant mistakes that can be made in
- Some of them are similar to typical security problems that happen in web applications and can be prevented with similar mechanisms
  - Secure communication, authentication and authorization, client-side injection
- Other security problems are app-specific and therefore require different security measures
  - Data storage and data leakage, inter-app communication, reverse engineering
- In general, its important to realize that any security-measures within the app can usually be circumvented by a determined attacker, e.g.
  - Offline authentication and authorization
  - Encrypted data based on hardcoded keys
  - Anti-reverse engineering techniques will slow down attacks, but won't prevent them

Additional Security Measures?

Note that besides the best practices discussed in this chapter, the server-side must of course be secure as well. So make sure that all the server-side attacks such as injection, XSS, CSRF, server-side access control issues and so on are prevented. This was discussed in detail in module SWS1.