A Brief Introduction to Mobile App Security

Lecture 8

Goal for today

- Hack-proofing mobile apps
- Understand limitations of locking phones
- Understand Android's permission system
- How to use or avoid cloak & dagger attacks

Content

- Facts about online security
- App markets
- Threat types
- Mobile device management
- Defining, using, handling permissions

Introduction

More malware is being launched every day than ever before: 230,000 new malware samples/day

Total cost of cybercrime in expressed in million $ (in the US):

- 2013 – $11.56 M, 2014 – $12.69 M, 2015 - $15.42 M
- 2017 - $21.22 M, 2018 – $27.37 M
- Today – going up (2025 predicted at $10 T)

Also high for Germany, Japan, UK, Brazil, Australia, Russia.

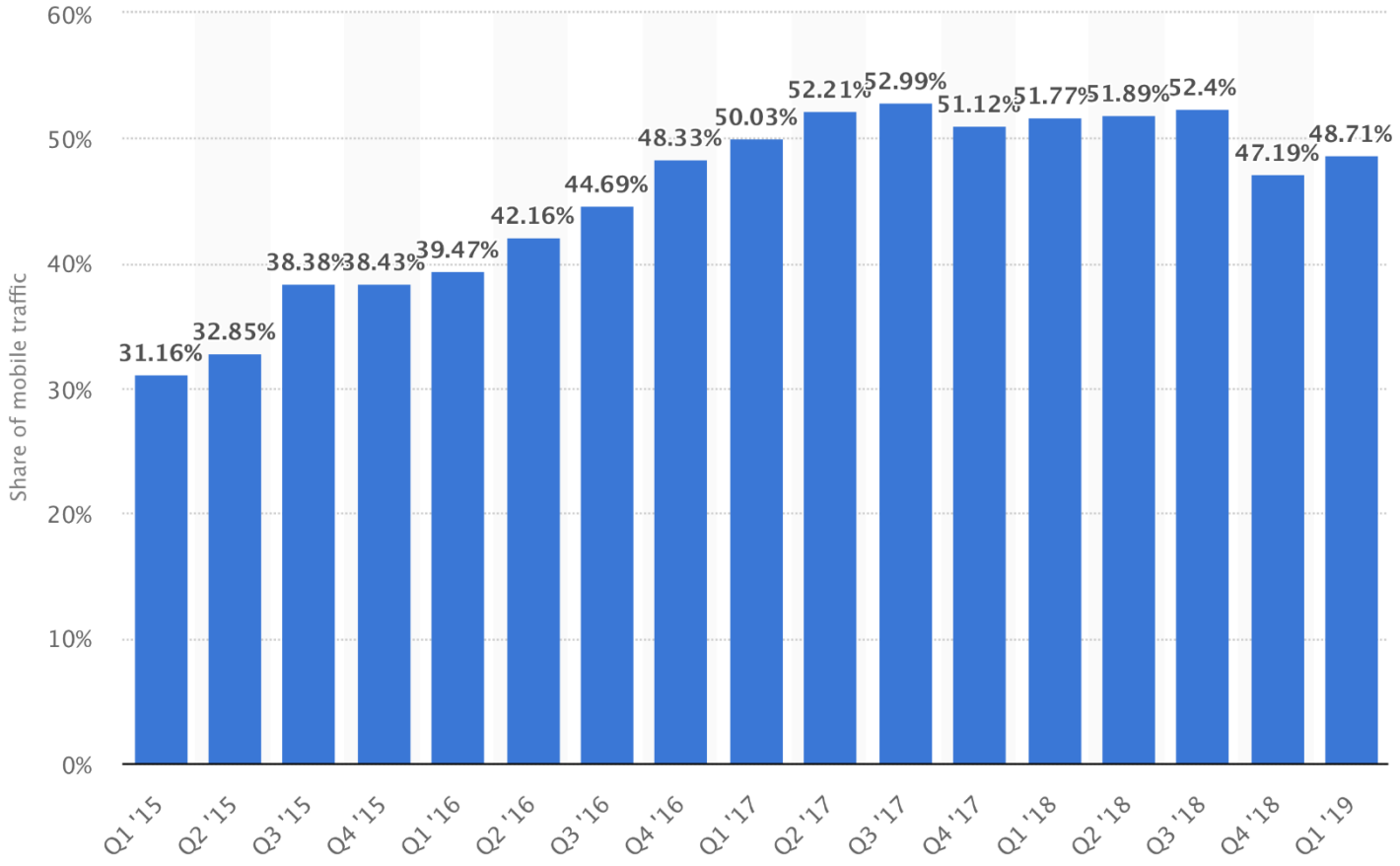
Introduction

Why is mobile security an issue?

- Increased reliance on person device
  - Communication, personal data, banking, work
  - Data security, authentication increasingly important
- From enterprise perspective: BYOD
  - Mobile device management (MDM) to protect enterprise
- Reliance on cloud: iCloud attack risks, etc.
- Progress from web use to mobile device UI
  - Apps provide custom interface, but limited screen size

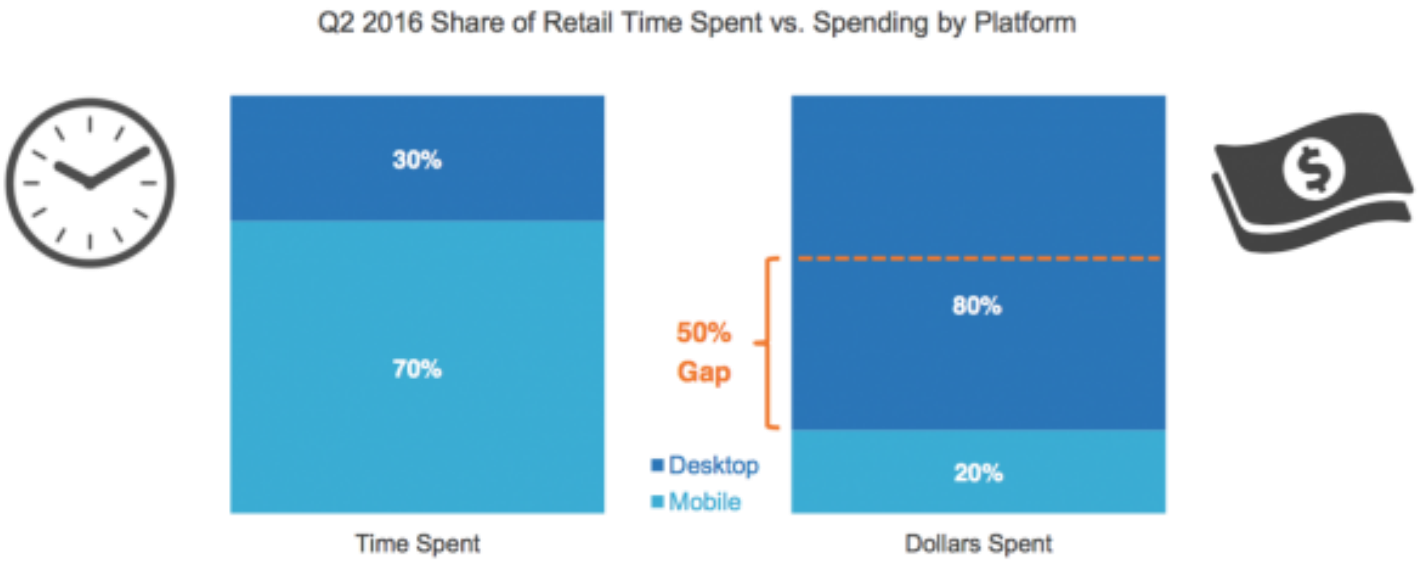
Introduction

World mobile app traffic



Introduction

Desktop usage versus mobile phone usage in terms of $$$



The App Marketplace

- Better protection & isolation than laptop install
- App review before distribution
  - iOS: Apple manual and automated vetting
  - Android: easier to get app placed on market, transparent automated scanning, removal via Bouncer
- App isolation and protection
  - Sandboxing and restricted permission
  - Android: permission model, defense against circumvention

What's on your phone?

- Contact list?
- Email, messaging, social networking?
- Banking, financial apps?
- Pictures, video?
- Music, movies, shows?
- Location information and history?
- Access to cloud data and services?

Q: what would happen if someone picked up your unlocked phone?

Top reasons leading to attacks

M1: Improper Platform Usage

M2: Insecure Data

M3: Insecure Communication

M4: Insecure Authentication

M5: Insufficient Cryptography

M6: Insecure Authorization

M7: Client Code Quality Issues

M8: Code Tampering

M9: Reverse Engineering

M10: Extraneous Functionality

Mobile platform threat models

Attacker with physical access

- Try to unlock phone
- Exploit vulnerabilities to circumvent locking

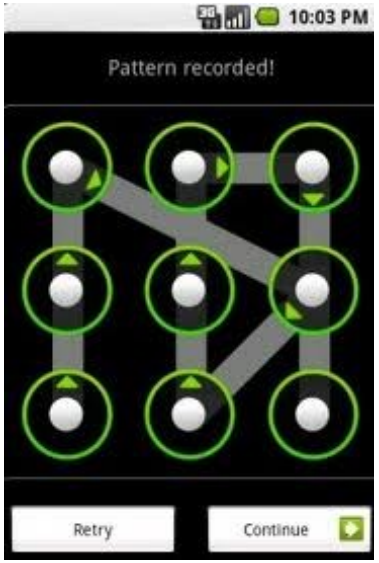
System attacks

- Exploit vulnerabilities in mobile platform via drive-by web downloads, malformed data, etc.

App attacks

- Use malicious app to steal data, misuse system, hijack other apps

Physical attacks



Need PIN or pattern to unlock device

→Once unlocked all apps are accessible

Plot twist 1: set a PIN or pattern per app (per photo, video)

→Protect settings, market, Gmail even if phone is unlocked.

→Examples: App Protector Pro, Seal, Smart lock

Plot twist 2:

→Front camera takes picture when wrong PIN entered

→Example: GotYa

Brute force password attacks

Traditional offline attack

1. Steal pwd file (Unix), or try "all" pwds
2. Usage of hashed pwds, cannot be reversed
3. Try dictionary attack, usage of salts

   hash(pwd, salt) - cannot be guessed

Online attack

→Can you try all passwords on a website?

→What does this mean for phone attacks?

How a brute force actually works

How can a hacker actually try all pwd permutations if a site/system blocks him after just a few attempts?

- Simple brute force – try all combinations, but get blocked by the system, either for a timeout or until account is unlocked.
  - Even without unlocks, the login process is slow and impractical
- Targeted brute force – try top 100-100K popular passwords ('12345', 'pass', 'hello') on a wide range of accounts
  - Can break a surprisingly large number of accounts
- Database attack – data breaches result in stealing DB(uid, pass) so that hackers can work offline on them using classic brute force attacks at very high speeds (e.g. GPU)
  - This is where password length and complexity come into play

Physical attacks

- Smudge attacks [2010]
  - Entering pattern leaves smudge that can be detected with proper lighting.
  - Smudge survives incidental contact with clothing.
- Potential defense [2011]
  - After entering pattern, require user to swipe across
- Entropy
  - People choose simple patterns – few strokes
  - At most 1600 patterns with <5 strokes

Biometric unlocking

- Biometric unlock
  - Fingerprint scanner
  - Requires backup PIN → not more secure than PIN
- Android 4.0: Face unlock
  - Raises “some” concerns about security
- Standard biometric security concerns
  - Not secret
  - Cannot be changed

Device lock and unlock on Android

- Similar PIN and fingerprint mechanics
- Fingerprint API (Android 23+) lets users:
  - Unlock device
  - Securely sign in to apps
  - Use Android Pay
  - Purchase on Play Store

Better device unlocking

More secure alternatives to unlocking:

- Unlock phone using a security token on body
- Wrist watch, glasses, clothing
- Cheap token, should not require charging

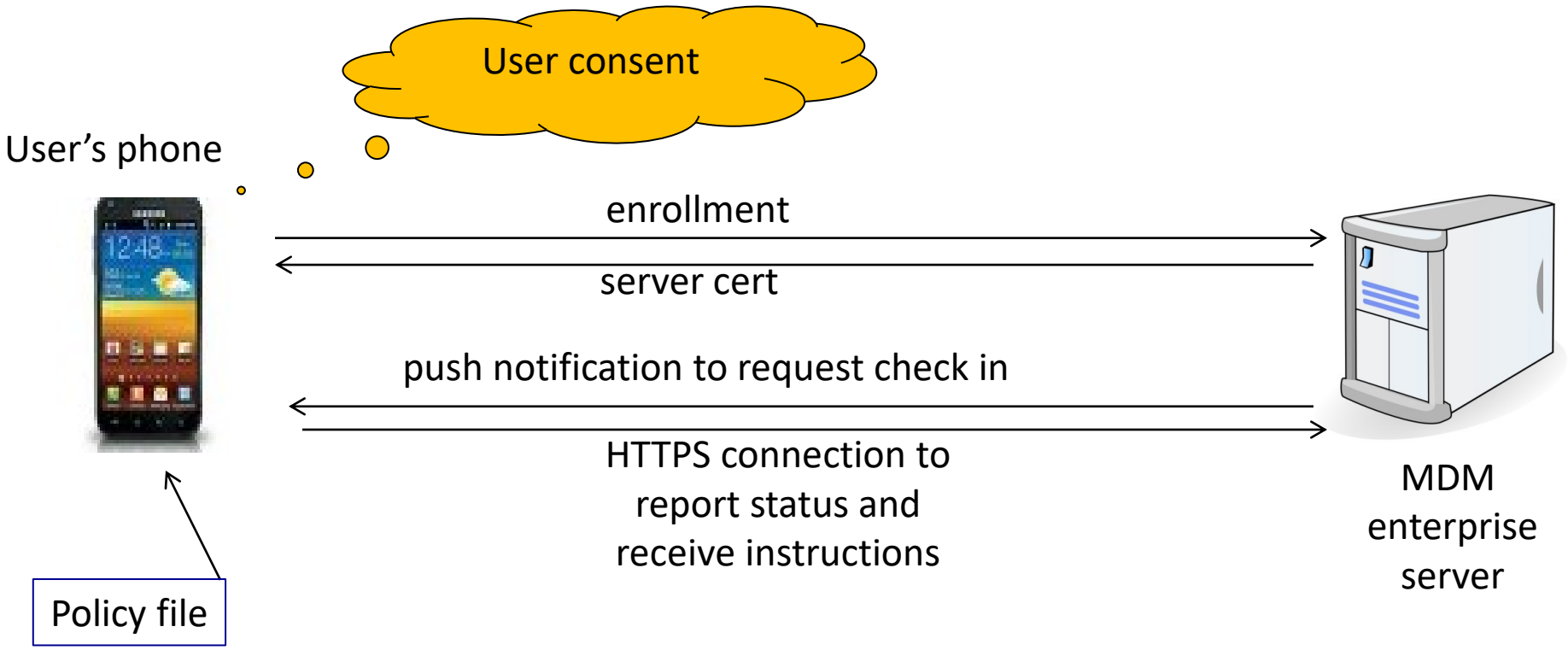
MDM: Mobile Device Management

- Manage mobile devices across organization
  - Consists of central server and client-side software

Functions:

- Diagnostics, repair, and update
- Backup/restore
- Policy enforcement (e.g. only allowed apps)
- Remote lock and wipe
- GPS tracking

MDM deployment



Summary: physical attack protection

Protect from thief:

- Authentication: PIN, swipe, biometric, … token?
- Phone locks if too many tries, unlock only through MDM
- GPS: where is my phone?
- Can phone destroy itself if too many tries?

Physical access can allow:

- Thief to jailbreak and crack pwd/pin
- Subject phone to other attacks

Next defense: erase phone when stolen (frequent backups)

The Android Platform

Android permissions

To maintain security, Android requires apps to request permission before they can use certain system data and features. Depending on how sensitive the area is, the system may grant the permission automatically, or it may ask the user to approve the request.

These permissions are Android permissions: they grant access to device features.

Android permissions

Android is a privilege-separated operating system

- Each app runs with a distinct system identity (Linux user ID and group ID).
- Parts of the system are also separated into distinct identities.
- Linux thereby isolates apps from each other and from the system.

App sandboxing

Each Android app operates in a process sandbox

- Apps must explicitly request access to resources and data outside their sandbox.
- They request this access by declaring the permissions they need for additional capabilities not provided by the basic sandbox.
- Depending on how sensitive the area is, the system may grant the permission automatically, or may prompt the user to approve or reject the request.

Using permissions

A basic Android app has no permissions associated with it.

- To make use of protected features of the device, use the <uses-permission> tags in the app manifest.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

Using permissions

Normal (*install-time*) permissions are automatically granted

Don't pose much risk to the user's privacy or the device's operation

Dangerous (*runtime*) permissions need user's explicit approval

Could potentially affect the user's privacy or the device's normal operation

The way Android makes the requests depends on the system version, **and** the **system version targeted** by your app:

- If system AND targetSdkVersion >=23 → runtime requests
- If system OR targetSdkVersion <=22 → requests at install time
- Else SecurityException in log

Normal and dangerous permissions

Normal – the app needs to access data or resources outside the app's sandbox.

E.g., set the time zone, access/change network/wifi state, Bluetooth, Internet, set alarm, vibrate, wake lock, use fingerprint

Dangerous – the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps. These are read/writes to user's:

calendar, camera, contacts, location, microphone, phone calls, sensors, sms, and storage.

Permission enforcement

The default system permissions are listed here.

Any app may define and enforce its own permissions.

A particular permission may be enforced at a number of places:

1. At the time of a system call, to prevent an app from executing certain functions.
2. When starting an activity, to prevent apps from launching activities of other apps.
3. Both sending and receiving broadcasts, to control who can receive your broadcast or who can send a broadcast to you.
4. When accessing and operating on a content provider.
5. Binding to or starting a service.

Permission groups (Android 23 / 6.0+)

All dangerous system permissions belong to permission groups.

If system **AND** targetSdkVersion >= 23:

If the app does not have any permissions in the permission group:

→A dialog box describes the permission group that the app wants access to. The dialog box does not describe the specific permission within that group. For example, if an app requests the READ_CONTACTS permission, the system dialog box just says the app needs access to the device's contacts.

→If the user grants approval, the system gives the app just the permission it requested.

If the app already has another permission in the permission group:

→The system immediately grants the permission without any interaction with the user. For example, if an app had previously requested and been granted the READ_CONTACTS permission, and it then requests WRITE_CONTACTS, the system immediately grants that permission.

Permission groups (<Android 6.0)

All dangerous system permissions belong to permission groups.

If system **OR** targetSdkVersion <= 22:

- The system asks the user to grant the permissions at install time.
- The system just tells the user what permission groups the app needs, not the individual permissions.

Defining permissions

→Declare them in AndroidManifest.xml using one or more <permission> elements.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp" >
    <permission
android:name="com.example.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-
group.COST_MONEY"
        android:protectionLevel="dangerous" />
    ...
</manifest>
```

Defining permissions

→Protection level – specifies how the user is informed of apps requiring the permission.

→Permission group – helps the system display permissions to the user.

→Label and description – are used by the system to inform the user about the permission

Why the permission model of Android 6.0+ ?

The new approach streamlines the app install process

→The user does not need to grant permissions when they install or update the app.

It gives the user more control over the app's functionality

→E.g., a user could choose to give a camera app access to the camera but not to the device location.

→App must handle these scenarios!

The user can revoke the permissions at any time, by going to the app's Settings screen.

Checking for permissions at runtime

Check for dangerous permissions.

The user is always free to revoke the permission.

```
// this refers to the current activity (app context)
int permissionCheck = (ContextCompat.)checkSelfPermission(this,
        Manifest.permission.WRITE_CALENDAR);
```

The method returns either:
PackageManager.PERMISSION_GRANTED or
PackageManager.PERMISSION_DENIED.

Requesting permissions at runtime

Call requestPermissions with (an array of) the permission name and an in-app integer request code:

```
if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.READ_CONTACTS)
        != PackageManager.PERMISSION_GRANTED) {

        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.READ_CONTACTS},
                        PERM_REQ_READ_CONTACTS);
}
```

Handle permissions request response

The system presents a dialog box to the user. When the user responds, the system invokes onRequestPermissionsResult, passing the user's response. The same request code is received.

```
public void onRequestPermissionsResult(int requestCode, String
permissions[], int[] grantResults) {
    switch (requestCode) {
        case PERM_REQ_READ_CONTACTS: {
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0 && grantResults[0] ==
                 PackageManager.PERMISSION_GRANTED) {
                // permission was granted …
            } else {
                // permission denied, disable functionality                }
            return;
        }
         case : … // other permission checks
    }
}
```

Handle permissions request response

- The system dialog box describes the permission group only, and does not list the specific permission.

  E.g., if you request the READ_CONTACTS permission, the system dialog box just says your app needs access to the device's contacts.

- The user only needs to grant permission once for each permission group.

  If your app requests any other permissions in that group (that are listed in your app manifest), the system automatically grants them.

- The system calls **onRequestPermissionsResult** and passes PERMISSION_GRANTED automatically

  … the same way it would if the user had explicitly granted that request through the system dialog box.

Handle permissions request response

- If the user denies a permission request, your app should take appropriate action.

  E.g., the app might show a dialog explaining why it could not perform the user's requested action that needs that permission.

- The user has the option of telling the system not to ask for that permission again.

  In that case, any time an app uses requestPermissions, the system immediately denies the request.

- The system calls onRequestPermissionsResult and passes PERMISSION_DENIED automatically

  … this means that when you call requestPermissions, you cannot assume that any direct interaction with the user has taken place.

Minimizing user frustration (1)

It's easy for an app to overwhelm a user with permission requests.

Using an intent to designate another app to perform an action:

- You do not have to design the UI, the app that handles the intent provides the UI.
- This means you have no control over the user experience. The user could be interacting with an app you've never seen.
- If the user does not have a default app for the operation, the system prompts the user to choose an app (an extra dialog)

Minimizing user frustration (2)

It's easy for an app to overwhelm a user with permission requests.

Using permissions:

- Your app has full control over the user experience when you perform the operation.
- Adds to the complexity of your task, since you need to design an appropriate UI.
- The user is prompted to give permission once, either at run time or at install time.
- If the user doesn't grant the permission (or revokes it later on), your app becomes unable to perform the operation at all.

Minimizing user frustration (3)

In some cases, one or more permissions might be absolutely essential to your app.

Ask for all of those permissions as soon as the app launches for the first time.

> E.g., if you make a photo app, the app would need access to the device camera – no surprise.
>
> If the same app also has a sharing feature, don't ask for READ_CONTACTS immediately –wait until that feature is used.

If your app provides a tutorial, it may make sense to request the app's essential permissions at the end of the tutorial sequence.

Cloak & Dagger attacks

A new class of potential attacks affecting Android devices.

- A malicious app may control the UI feedback loop and take over the device — without giving the user a chance to notice the malicious activity.
- Require only two permissions that the user does not need to explicitly grant and for which she is not even notified:
  SYSTEM_ALERT_WINDOW (“draw on top”)
  BIND_ACCESSIBILITY_SERVICE (“a11y”)
- These attacks affect all recent versions of Android (5-7).
- Not fixed (June 2017).

Cloak & Dagger attacks demos

- Invisible Grid Attack (video)
- Context-aware/hiding Clickjacking + Silent God-mode Install Attack (video)
- Stealthy Phishing Attack (video)

Courtesy of http://cloak-and-dagger.org/

Other reading resources

- Cyber security facts
- Better mobile dev
- Hack-proof your smartphone
- Cloak & dagger attacks
- Read Secure mobile development