Previous a researcher in the usable security group at ICSI/UCB

Now a lead research engineer at Two Six Labs working on DARPA Brandeis

This talk goes over work from UCB
Install-time permissions

Accept all or don’t use the app at all. No obvious hints for why an app is requesting a certain privilege. Can be overwhelming to end-users.
I joined the usable security group at UC Berkeley shortly after they published results showing over 1 in 3 attempts to access sensitive data are unwanted by the user under the install-time model.

This work motivated further research into better aligning permissions systems with user privacy preferences.
Run-time permissions were introduced to Android in version 6 “Marshmallow,” released in October 2015. Asks for permission on the first time an app tries to access the protected resource (i.e., “ask-on-first-use” or AOFU).

An improvement to install-time permissions. This provides contextual clues to the user: in this example, Facebook needs to read photos and videos for its Camera Roll feature.
AOFU is an improvement, but only captures user privacy preferences in one context: the first time a permission is exercised. It naively applies that decision to all future contexts.

A user might be OK with Uber collecting their information in requesting a ride, but not for continuous location tracking.
AOFU has shortcomings, so how can we improve it?

We can naively ask on every use. But unusable.

How about we ask on some uses? Let’s prototype evaluate it. This requires modifying Android.
Why is a lot of mobile security/privacy research on Android? What about iOS?

Closed source: Can’t modify.
Encrypted app packages (.ipa files): Requires jailbroken phone to decrypt; lots of hoops to jump through.
However, can still install root cert to MITM traffic; only mildly annoying to do.
Going forward, all methods and results in this talk are most relevant to Android.
Android apps operate at the top of the software stack: Apps call functions exposed by the Android framework. For example, functions to manipulate on-screen UI elements or read sensitive user data (e.g., location, contact information, etc.).

The Android framework is the highest level of abstraction, acting as a front-end to the underlying software/hardware stack. This makes it easy to write one app that works for a broad set of Android devices.
Just as you can write Android apps, you can write your own fork of the Android platform too.
Some cool things you can do up and down the stack:

- Framework: Custom permissions system
- Native libraries: Capture unencrypted TLS traffic
- HAL/HIDL: Get raw touchscreen input data
- Linux kernel: Log all file operations
Modifying, deploying, and testing Android source code has a lot of little quirks and details associated with it. Will only go over how to get started with it. More detailed documentation at source.android.com

My goal is to give you enough to be curious and ask questions, so feel free to contact me. It took me a while to get comfortable with it myself.

Twitter: @irwinreyes.com
Email: irwin.reyes@twosixlabs.com OR ioreyes@icsi.berkeley.edu OR email@irwinreyes.com
To build Android, you’ll need:

1. Modern Linux build environment. Ubuntu Server 19 LTS generally works out of the box. Might need to install gcc and openjdk.
2. Lots of hard disk space. The Android 9 source tree takes up about 150 GBs. Compiling it for a phone will result in about 250 GBs of output.
3. Building Android can be done in parallel. More CPU cores = faster (but is eventually disk-bound).
4. A smartphone compatible with the version of `Android you’re developing. Nexus 5/5X/6P recommended for Android 6 through 8. Pixel series recommended for Android 9 and 10. Can develop using VMs, but VMs are slow and unreliable.
5. Building Android can take a long time. Debugging is done by using log lines. No runtime debugger for the OS. Long turnaround between building and installing on phones.
On a 40-core (80 logical) server with SSDs, building Android from scratch takes about 25 minutes.

Luckily, you can do incremental builds afterwards.
The Android source tree is made up of several hundred Git repositories. The “repo” tool manages those Git projects; initialize build environment, pull code, check for outstanding changes, etc.

Each Git project roughly corresponds to a particular part of Android: device-specific code, the Linux kernel, preinstalled apps, etc.
The `repo` tool can also manage branches and tags. When first initializing the build environment, you have to pick a tag corresponding to the version of Android you want and what device you’re targeting.

As mentioned before, Pixel phones are highly recommended for modern Android development. Older releases target all the Nexus phones.
Use the repo tool to select the Android version. Version tags available at https://source.android.com/setup/start/build-numbers
Use the lunch tool to select the target device. Target device codenames available at https://developers.google.com/android/images Taimen is the codename for the Pixel 2XL.

Fastboot commands assume the phone bootloader has already been unlocked. How to do this is left as an exercise to the reader.

Incremental builds only need the steps on this slide. 4 – 7 for a new session, and only 6&7 for an existing session.
The source tree is huge and hard to navigate. Android Xref is a really useful resource for searching through the code.
Most modifications will touch the frameworks/base project. This is where nearly all API functions used by apps are implemented.

Useful terminology: Managers are app-space code that are front-ends to system-space ManagerServices that actually talk to the underlying HAL.

For example, LocationManager (app-space) and LocationManagerService (system-service implementation).
Coincidentally, frameworks/base/ also has a PermissionManagerService.

“Manages all permissions and handles permissions related tasks.” Hmm…
we modified the PermissionManagerService to use context and predict user preferences

Permission requests go through the PermissionManagerService.
Normally, when an app requests sensitive data (e.g., location), it goes through the corresponding manager.

The manager talks to the backing service, which requests a permission check.

The PermissionManagerService checks if the app has declared the appropriate permission in the manifest and if the user has approved it under AOFU. Approves the access if so.
We modified this flow to include context in the request, which is used by an additional step called by the PermissionManagerService:

The context is used to predict user preferences based on a prebuilt bootstrapped classifier model. It has a training phase for personalization. See Oakland paper for more details.
In practice, this works very similarly to the existing AOFU model.

But the user is prompted when either the classifier is in training mode (i.e., when device is first used) or when the classifier produces low-confidence results.
In practice though, the classifier isn’t perfect. It will still produce unwanted outcomes sometimes.
users need to **review and correct** the classifier’s results

How can users control this without being overwhelmed?
Existing configuration tools for permissions are insufficient: They only offer blanket on/off toggles, and they don’t give any information about the circumstances in which permissions were exercised.
We developed a front-end configuration tool to support users in contextual permissions systems and tested them.
In the initial TurtleGuard study, we iterated through designs for these controls and evaluated interactive mock-ups of them with 598 participants. 580 produced complete responses, from which the results were drawn.

The final design looked something like this: Have a history of all recently allowed/denied permissions, plus per-app settings.
In evaluating these designs, we split the participants into a control group (presented with the stock settings) and an experimental group (presented with TurtleGuard).

Four tasks:
1. Determine the app that most recently accessed location
2. Determine what permissions are granted to a given app
3. Determine if a given app could access location in the background
4. Prohibit app from accessing location in the background

Tasks 3 and 4 take context (app visibility) into account. TurtleGuard fares much better.
are dynamic permissions and controls actually usable on real mobile devices?
The TurtleGuard study steps us through the design of the controls.

We eventually implemented them into the Android platform as part of the system settings. We also implemented a live permissions model for this to control.
field testing with real users

37 participants from Bay Area

1 week using custom Android

usage/error logging and exit interviews
<table>
<thead>
<tr>
<th></th>
<th><strong>STOCK ANDROID</strong></th>
<th><strong>DYNAMIC PERMISSIONS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Median permission error rate</td>
<td>Under AOFU: 20.0%</td>
<td>Classifier: 5.26%</td>
</tr>
<tr>
<td>Median permission prompts</td>
<td>Under AOFU: 15</td>
<td>Classifier: 13</td>
</tr>
<tr>
<td>SecurityExceptions thrown</td>
<td>N/A</td>
<td>Classifier: 3/day</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(out of 2000 denials/day)</td>
</tr>
</tbody>
</table>
8% expressed *annoyance* at the prompts

70% *surprised at the frequency* of app access to sensitive data

40% *opened the permission manager* ("TurtleGuard")

**GET_ACCOUNTS** permission most likely to be **restricted** by users

**READ_CONTACTS** permission most likely to be **permitted** by users
personal note: realistic user studies are very **time consuming**!
Because we owned the operating system, we had a very privileged view on how apps interact with user data.
Apps are able to request access to private user data and sensitive device resources.

In their app store listings (such as this one from the Google Play Store), apps disclose their capabilities. However, these disclosures don’t tell the full story. Do apps actually use these privileges? With whom do they share sensitive data?
We developed a fully automated platform to analyze how apps actually collect and share sensitive data.

We instrumented the Android operating system and used advanced network traffic monitoring tools. Apps are run and evaluated without any human interaction. Technical details in the paper.
Custom Android 6 ROM for observing access to sensitive resources.

Lumen Privacy Monitor to see who gets that info.
We run any Android app in this environment and observe its behavior.

Not enough to just launch the app. Solution: explore with monkey. It’s dumb!

Monkey did as well as undergrads 60% of the time in children’s games. Results are a lower bound.
Our system observes when apps access and share personal information, as well as unique persistent identifiers that can be used to track users over time and across services.

<table>
<thead>
<tr>
<th>PERSONAL INFORMATION</th>
<th>PERSISTENT IDENTIFIERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Email Address</td>
<td>Hardware Serial Number</td>
</tr>
<tr>
<td>Phone Number</td>
<td>IMEI</td>
</tr>
<tr>
<td>GPS Latitude/Longitude</td>
<td>Wi-Fi MAC</td>
</tr>
<tr>
<td>Wi-Fi Router BSSID (MAC)</td>
<td>Android ID</td>
</tr>
<tr>
<td>Wi-Fi Router SSID (Name)</td>
<td>SIM Card ID</td>
</tr>
<tr>
<td></td>
<td>Google Services Framework (GSF) ID</td>
</tr>
<tr>
<td></td>
<td>Android Advertising ID (AAID)</td>
</tr>
</tbody>
</table>
COPPA is one of the few comprehensive privacy laws in the US. It covers online services (like apps) that have users under 13 years of age.

Verifiable parental consent: Can take on the form of out-of-band methods like credit card verification or a phone call. Our system is fully automated with no direct human input, so observed data collection did not have consent.

Note that our analysis system is not specific to COPPA. It can be adapted to other regulatory measures such as GDPR and California’s new online privacy law.
What apps does this law apply to? We looked at the “Family” category in the Google Play Store.
Those are apps that have opted into the Designed for Families Program, or DFF for short.

DFF is opt-in. Participation is the dev saying kids are in the target audience. Google can reject or remove DFF apps not relevant to children.

DFF’s requires devs to represent their apps **and bundled services** are COPPA compliant. For example, graphics, communications, analytics, and ads.
Apps collected between November 2016 and March 2018

Average 750K installs

Representing nearly 1900 developers
The majority of our corpus was seen to be in potential violation of COPPA, in that they:

- Accessing and collecting email addresses, phone numbers, and fine geolocation
- Potentially enabling behavioral advertising through persistent identifiers
- Sharing user data and identifiers with SDKs that are themselves potentially non-compliant
- Not using standard security technologies

Note that some apps were observed engaging in more than one of these behaviors, so the percentages will add up to more than 57%.
We attributed most of these violations to various third-party services bundled with apps.

These services allow developers to expedite production by offering drop-in functionality, whether for graphics, communications, advertising, or analytics, among others.
We believe that these violations are prevalent because the gatekeepers in the mobile app space are not enforcing their own terms meant to protect end-users. (recall DFF requirements)

Google controls the Android operating system and the Play Store, which is the primary app distribution channel for Android. They are in an excellent position to conduct analysis similar to ours on all apps submitted to the Play Store, as well as secure the operating system to prevent potential abuses.
For example, COPPA prohibits behavioral advertising for children. Behavioral advertising uses persistent identifiers to build profiles of users by tracking individuals over time and across services.

Google has recognized the privacy implications of persistent identifiers, and in 2013 introduced the resettable Android Advertising ID (AAID) to give users (or parents) control over how advertisers track them. Since 2014, Google requires developers and advertisers to use this in lieu of non-resettable device identifiers like the IMEI and Wi-Fi MAC address.
However, a large chunk of children’s apps were seen sharing the AAID with another non-resettable identifier to the same destination, which defeats the purpose of the AAID. Although Google requires the use of the AAID, non-resettable identifiers remain available to apps.
We found adherence to this AAID-only policy to vary among third-party ad networks. From nearly constant violation with Chartboost to nearly full compliance with Doubleclick (which is a Google company).

Full table in paper.
Not all third party services are appropriate for children, as claimed by those services themselves. We found nearly 1 in 5 DFF apps sharing personal information or identifiers with third-party services whose own terms of use prohibit their deployment in children’s apps.

Recall that the apps we studied were opted into the Designed for Families program, indicating that the developers intended to include children in their apps’ audience. Still, these same developers were found including these prohibited services.
Presumably, these services prohibit their use in children’s apps because these services may engage in non-COPPA-compliant data collection and processing.
Crashlytics is a crash reporting service that allows developers to receive usage information about their apps in the wild. Crashlytics terms prohibit its use in children’s apps.
Google owns Crashlytics, Android, and the Play Store. Google should be able to detect when its own service is integrated with children's apps, then take necessary steps to address that.
Potential COPPA violations are widespread, but the reality is regulatory agencies like the FTC have finite enforcement capability. COPPA, however, allows for industry self-regulation in the form of review and certification from designated safe harbor certifying bodies.
However, we found that apps certified by safe harbors fared no better than DFF apps as a whole.
In fact, they were in some cases were worse.

There’s a large body of economics research into adverse selection, in which bad actors are the ones most likely to participate in positive signaling activities.

We suspect safe harbors have had the unintended consequence of allowing potentially non-compliant apps to signal that they are indeed COPPA compliant.
Our study has had an impact in industry and enforcement since its release last April.

I’ll close this presentation with an example of such impact.
In our study, we named Tiny Lab Productions’s games as a popular example of the collection of personal information from children without verifiable consent.

Their game Fun Kid Racing has over 10M installs, and was seen collecting and sharing geolocation data with advertisers. Of Tiny Lab Production’s 82 DFF games, we observed this behavior in 81 of them.

In response to our findings, Tiny Lab Productions stated to CNET that their games are not necessarily for children.
RACING GAME FOR KIDS
CHILDREN LOVE IT!

Fun Kid Racing is one of the best racing games for kids — this free game features simple controls that children love and can quickly master. Take control of a huge variety of vehicles and race them to the finish line, optionally collect coins along the way, and do fun stunts on the many varied environments.

The levels are designed specifically for children, and will keep them entertained for hours!
We have identified that 2,667 apps are potentially incorrectly listed as directed to “mixed audiences,” and “not primarily directed to children,” corresponding to ~51% of Designed for Families (DFF) apps from our original sample which are still listed on DFF. Developers seem to have an incentive to miscategorize their apps as “not primarily directed to children” so they will be able to engage in defective “age gating,” thereby very likely causing children under 13 to enter ages over 13, allowing COPPA-prohibited behavioral advertising.

We reported Tiny Labs to Google, along with our results identifying all other DFF apps potentially violating COPPA and failing to meet Google’s own standards for DFF apps
Google responded to us saying that there was no way to detect these issues at scale, and that it was unclear that Tiny Labs was offering child-directed apps.

1) This was exactly the technology we developed and deployed in the course of this research
2) Definitely *not* for kids
In September, the New Mexico Attorney General filed a suit, with Tiny Lab Productions and Google as co-defendants for violating children’s privacy law.
After facing scrutiny from the New York Times and the New Mexico AG’s office, Google recently took a more aggressive stance towards Tiny Labs, taking down their apps after Tiny Labs failed to address the various privacy issues we identified in those products.
In the course of developing and refining this app testing infrastructure, we encountered a “critical bug” that turned out to be something more interesting
One day as a sanity check, I asked our database of app behaviors, “give me all that apps that sent location data but never declared permissions to access the phone’s location.”

This intersection should be null.
Instead, the database turned up over 1300 apps that match this criteria.

I panicked for a second because
Despite the failures of permission systems they serve an important purpose. At the very least, if an app is denied permission, it must not access resources protected by the permission.
The Android permissions system can be circumvented, often through the permissions system itself.
This “bug” resulted in a USENIX paper
Example side channels: EXIF data; /proc/net
Example covert channel: App 1 holds appropriate permissions, writes sensitive data to shared storage, App 2 doesn’t have permissions but can read from storage.
THANKS!

https://irwinreyes.com

@irwinreyescom

irwin.reyes@twosixlabs.com

ioreyes@icsi.berkeley.edu