Principles of Android Platform Security

Stanford CS155 guest lecture

May 20, 2021

Ashwini Oruganti <ashfall@google.com>

Whoami

Ashwini Oruganti (ashfall@google.com)

Senior Software Engineer and one of the leads on the Android Platform Security team

Work on a wide range of things but my favorites are

- Making the platform APIs “secure by default”
- Eliminating entire classes of vulns by solving for “footgun” issues

My background

2012-2014: Core developer for twisted.python, an asynchronous event-driven networking library in Python

2014-2015: Worked on a TLS implementation in Python that used the crypto primitives from openssl but implemented the handshake itself as an explicit state machine in Python

2015-2019: Worked at Eventbrite, Docker, Apple

2019: Joined Google

Why am I here?

- Talk about high level concepts and guiding principles we think about daily
  - practical advice on how to build secure mobile applications
- Encourage you to look at defensive security work and to think about security at a design level
- What working in security in general and mobile security in particular is like

There’s lots of layers to security

But we’ll focus on the platform itself here since that is my main focus

Our goal: Keep people safe

“Strive to build systems so strong that we ourselves cannot even break into them, and so private that people can trust us with their most sensitive data.” — Nick Kralevich

Regardless if they’re paying $1000 or $10 for their phone

Regardless if they obsess about security as much as us or don't think about it at all

Regardless if they’re some ‘important’ person or just an average person

Android has over 2.3 billion 30 day active users, we want **all** of them to be safe.

Or more succinctly

Make things so secure we're not needed anymore.

That's way too high level though

Let's break it down into two parts

1) The OS model
2) Protecting the system to enforce that model

OS Model?

How interactions on the system are supposed to work

How components interact

Who should be allowed to do what and how

It’s the most important part: You can fix a bug but fixing design flaws in the model is significantly harder

The model is critical

What is working as intended and what is a vulnerability depends on your OS model

Before you can try and secure your OS you need to have a useful model to enforce.

The security of a system against exploits is moot if attackers can do everything they want within the OS's model

Example?

On older OSes it's perfectly acceptable for applications to inject code into other applications running on the same user.

This leads to massive pain for developers, new and exciting vulnerabilities injected into places that were safe, and limitations to what a developer can do without sparking security arms races.

On mobile OSes such a capability would be treated as a serious security vulnerability.

But first: It has to be a useful product

Regardless of what you're building If no one uses it it doesn't matter how secure it is, they'll be using something else (that could be far more insecure).

You have to build *useful* secure devices, not just big bricks, don't let the perfect be the enemy of the good.

But technical debt can be hard or impossible to fix, so there's a balancing act.

“Security is ~~the enemy of~~ usability”

This is a wrong statement

Security is opposed to **raw access**, not usability or availability.

The right kind of access can give you the same end utility with **massive security benefits**.

Example: Sharing a file

Scenario: You want to take a file you made in one application and view/modify it in another

Desktop: Save the file somewhere and then open it in the application

Mobile OSes: Share the file from the first app into the second

End result is the same utility, but in the older OS case any other application can view/modify that file too

Mobile OSes are fundamentally different than desktop

Mobile OSes don’t work the same way as the desktop OSes of the past. Why?

New OSes give a chance to learn from the mistakes of the past generation and try new things. Mobile was different enough that it allowed new operating systems and methods of interaction to come about.

Mutual Consent Model

Android operates on a three-party mutual consent model for all actions and data is owned by by the actor that created it.

What consent looks like depends on the specific action, sometimes it can be explicit and sometimes implied by the action itself.

What do you mean by consent?

When it comes to security we're very explicitly talking about **informed** consent.

The person that is making the decision (be it user or application developer) must understand the decision they're making, otherwise it's not meaningful consent.

Actors on Android

Android has three core actors

The User - You

The Application - The developer(s) of the application(s)

The System - Us, the adult in the room

The Application

Unlike older OSes mobile OSes treat the specific application, not the user account, as lowest level actor.

This means that applications own their data and control access to it.

Why?

Compartmentalization!

This might seem small or simply pro-DRM but it and the compartmentalization it implies and enables are one reason why mobile OSes are so much more secure.

Example: Why should a game installer be able to access Chrome's password db file?

Keep in mind

Android is a consumer OS, we assume that users and developers do not fully understand or focus on security.

Consent has to be meaningful for **real** people.

Don't just offload your job onto the user

There’s a bad habit in security to offload the hard decisions to the user under the guise of asking for permission.

It's not a user’s job to keep their device secure, it's our job.

Meaningful consent is non-trivial

“What if we just asked for everything” almost immediately leads to fatigue and blindness, removing your ability to get any kind of meaningful informed consent **for anything**.

You have to balance getting their informed consent with not asking questions they can’t be equipped to answer or ask so many questions they just blindly say yes to everything.

Some angles of attack (details follow)

1) Disallow unsafe behavior
2) Safe default behavior
3) Make intent obvious
4) Avoid prompt fatigue
5) Ask in an understandable way

Disallow unsafe behavior

If you can’t do it in a way that keeps the user safe or presents security decisions in a way an average usage can reason about, you probably shouldn’t do it in a consumer OS.

Safe By Default

“It should be easy to do the right thing, impossible to do the wrong thing”

You shouldn’t have to take an action to be secure, it should be the default behavior. Both for users and developers.

Make intent obvious

You can avoid asking directly for consent if interactions are structured in a way where it's unambiguous.

Example: With a file picker the user selects a file to share with the application via a UI, the application is then granted access to that specific file. There’s no need to ask again about giving the app the file because the user intent is clear from their actions.

Avoid prompt fatigue and blindness

If you ask too much users will very quickly become blind to the prompt and blindly hit yes.

- Avoid prompts blocking the user from what they want to do
- Make some higher risk actions/grants require active effort (e.g. go to settings and enable a special access permission for an application instead of showing it as a runtime).

Ask in an understandable way

Your questions should be narrowly scoped and clear as to what they allow

“Allow X to make changes to your system” — ?????

“Allow X to access your contacts” — understandable

Alright, so we have a model

If our model is perfect then the only way to do ‘bad’ things would be to exploit the system and escape the model.

So now we can finally talk a bit about hardening

Goals

Since our goal is to make it so vulnerabilities don’t happen

Fixing those that happen isn’t enough — we want to make it so they don’t exist. This is where hardening and good engineering comes in.

Some major themes (details follow)

- Isolation and Containment
  - Attack surface reduction
  - Principle of least privilege
  - Architectural decomposition
- Exploit mitigation and prevention
- Integrity
- Safe by default APIs and tools

Attack surface reduction

“If vulnerable code isn’t reachable does it matter?”

Attack surface reduction is all about limiting the area an attacker has to play with.

The less surface area the have the easier it is to understand and protect

Principle of least privilege

A component should only be able to do the things it needs to do, it shouldn't have more capabilities than it needs.

Since Android L, Android has used SELinux to constrain the entire system

We've also expanded what SELinux can do, for example in Android M we expanded it to cover filtering ioctls, which were a large source of security issues in kernel drivers.

There is no root on Android

Since L with SELinux there's no user with root-like privileges on Android.

A root/administrator user with all the capabilities is antithetical to a secure system.

Principle of least privilege

Thanks to this work serious bugs for other systems running on top of Linux were minor or not issues at all due to the attack surface being unreachable.

Example: local root CVE-2017-6074

Not compiled into our kernels

Even if it was unreachable due to SELinux

Principle of least privilege

This isn't a one-off, by restricting the privileges of all components on the system we've avoided being impacted by vulnerabilities we didn't even know existed at the time we made our hardening changes.

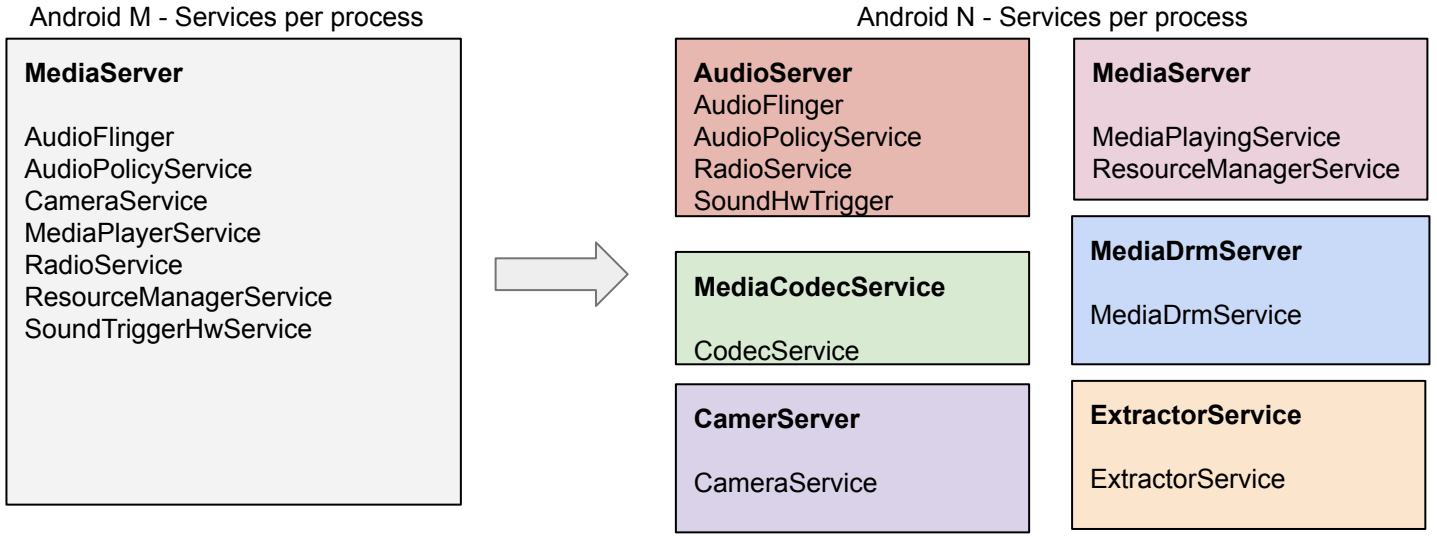
A large number of the bugs we see are mitigated by the restrictions we place due to least privilege.

Architectural decomposition

In order to best take advantage of the principle of least privilege the components need to be as small as possible to prevent capabilities becoming too large.

Decomposition also makes exploiting the system more difficult when a component is compromised due to the limited privileges and reachable attack surface from that component.

Example: Mediaserver



Exploit mitigation

Even if potentially vulnerable code is present, we want to mitigate its risk of being exploitable.

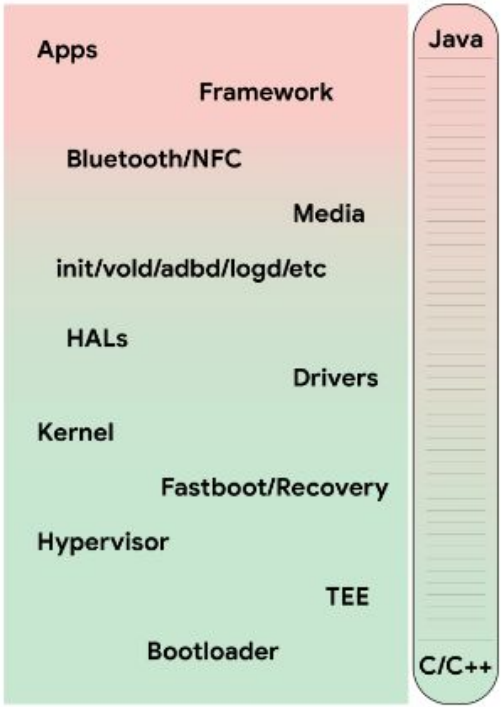
Ideally so that exploitation isn't possible at all and it becomes simply a bug, or by increasing the difficulty of exploitation.

Example: Unsigned Integer Overflows

A large number of bugs in media parsing was the result of incorrect bounds checking due to unsigned integer overflows

Instead of simply fixing each bug as they came in **we enabled sanitization** such that an integer overflow in this code would cause an abort(), leading to a crash of the media component instead of a vulnerability.

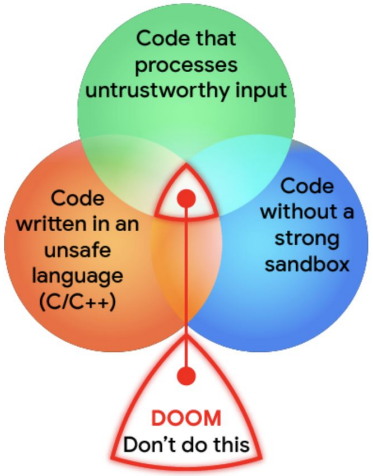
Rust!


Apps
Framework
Bluetooth/NFC
Media
init/vold/adbd/logd/etc
HALs
Drivers
Kernel
Fastboot/Recovery
Hypervisor
TEE
Bootloader
Java
C/C++

Rust!

Memory-safe languages like Rust help us overcome the limitations of sandboxing in two ways:

1. Lowers the density of bugs within our code, which increases the effectiveness of our current sandboxing.
2. Reduces our sandboxing needs, allowing introduction of new features that are both safer and lighter on resources.



Example: Removing execmem permissions

As part of improving the security of mediaserver the ability to mark memory executable was removed.

This greatly increases exploitation difficulty as most ROP based-attacks simply mark some shell code executable and then jump to it.

Example: Data in transit protection

My personal passion :)

Networks are not to be trusted, especially not on mobile.

Proper data in transit protection (aka TLS) on **everything** protects the device from snooping and content injection attacks

In Android P we added Private DNS (DoT) to also protect DNS requests

Integrity

We also need to **protect the OS itself from modification** or from an attacker from simply removing the flash and reading the data directly.

For the first we have dm-verity which uses a signed hash tree to detect modification of the system images and refuse to load those pages.

For the second we have disk encryption with hardware entanglement.

Integrity

Applications are signed by the developer as well so that they, and only they, can provide updates to their app.

This is because code is how the application expresses its consent for actions, being able to inject or replace code would bypass the model.

User Integrity

The same logic for wanting application consent to be from the true developer applies to the user -- Consent needs to come from the user and not J Random Attacker.

For this we have secure lockscreens .

Safe by default, again!

Like with user and developer decisions we want the platform and APIs to be safe by default and resilient to mistakes.

It shouldn’t be easy to introduce a vulnerability.

e.g. PendingIntents are no longer mutable by default starting Android S

Want to attack Android?

We pay. https://www.google.com/about/appsecurity/android-rewards/

Code execution reward amounts

Description	Maximum Reward
Pixel Titan M	Up to $1,000,000
Secure Element	Up to $250,000
Trusted Execution Environment	Up to $250,000
Kernel	Up to $250,000
Privileged Process	Up to $100,000

Data exfiltration reward amounts

Description	Maximum Reward
High value data secured by Pixel Titan M	Up to $500,000
High value data secured by a Secure Element	Up to $250,000

Lockscreen bypass reward amount

Description	Maximum Reward
Lockscreen bypass	Up to $100,000

Shameless plug

We’re hiring and take interns every year (too late for this summer though).

If you want to work on the security of the largest consumer OS in the world, do let us know :)

END

Questions? Rants? Comments?