Isolation and Confinement in Android

Lujo Bauer
18-732
Spring 2015

Slide credits: Limin Jia
Smartphones Store Sensitive Information
What Is There to Protect?

- **User data**
  - Personal information, sensitive input devices, device metadata

- **System resources**
  - Cost-sensitive APIs
    - Telephony, SMS, network, in-app billing, NFC, ...

- **Application isolation**
Android Architecture

Security Cornerstone: Sandboxing

- Each application has its own Linux user ID, runs as a separate process
- OS mediates inter-application communication
Android Permissions

- How to decide what an application may do?
- Sensitive resources are protected by permissions
  - Applications request permissions from the user at install time
Problem 1: Privilege Escalation

Shopping

Barcode scanner

Camera permission

-!-
Problem 2: Information Leakage

Editor

- Secret Files permission
- Internet permission

Secret File Manager

Secret Files

Internet
Proposed Solutions

- Identifying dangerous combinations of permissions [Enck et al. 2009]
- Data tainting at the variable level to prevent information leakage [Enck et al. 2010]
- Data shadowing [Hornyack et al. 2011]
- Domain isolation [Bugiel et al. 2011]
- Run-time monitoring of application interactions to mitigate privilege escalation [Bugiel et al. 2012, Dietz et al. 2011]

Disadvantages:
- Policies usually hard-coded
- Lack of formal guarantees
Two More Solutions

- **System 1 (Sorbet): Light-weight information-flow policy enforcement** [Fragkaki et al. 2012]
  - Leveraging, enhancing Android permissions

- **System 2: Enforcing distributed information-flow control (DIFC)** [Jia et al. 2013]
  - Inspired by systems that enforce DIFC at OS level
  - Enforcement mechanism
    - Associate each app with a set of information flow labels
    - A run-time monitor mediates communications based on labels
Light-weight Information-flow Protection

- Apps can specify flows that should be prohibited
  - $\text{disallow-flow}(P_1, P_2)$: an app that accessed (including indirectly) a component protected by $P_1$ cannot use $P_2$

- Apps can specify exceptions
  - $\text{allow-declassify}(P_1, P_2)$: an app can avoid the above information-flow restriction
Information-flow Protection

Permissions
- Internet permission
- Secret Files permission

Editor

Network API

Check call

Sorbet Reference Monitor
Information-flow Protection

disallow-flow (secretFiles, internet)
allow-declassify (secretFiles, internet)
Declassification

Secret File Manager

Email

Permissions
- Internet permission
- Secret Files permission
- Declassification

Email Compose Activity

Network API

Sorbet Reference Monitor

Check call
Complicated… Does It Really Work?

- How do we gain assurance that the design is sound?

- Prove it! (Formally.)
Noninterference

- A system with components containing secrets is observably equivalent (from the adversary’s point of view) to the system without these components

- … we’ll come back to this in a little bit
Enforcing DIFC

- **So far today: Poor man’s solution**
  - Allows only simple information-flow policies
    - A component cannot refuse a secret
    - A component cannot reliably prevent another component from getting its secret

- **Next: A full-fledged enforcement mechanism**
  - Inspired by systems that enforce DIFC on OS
  - Associates each app with a set of information-flow labels
  - A run-time monitor mediates communications based on labels
Information-flow Labels

A set of secrecy tags

A set of integrity tags

A set of declassification and endorsement tags

Partial order of labels decides whether data can flow from \((S_1, I_1, D_1)\) to \((S_2, I_2, D_2)\)

\[(S_1, I_1, D_1) \equiv (S_2, I_2, D_2) \text{ iff } S_1 \subseteq S_2 \text{ and } I_2 \subseteq I_1\]
Partial order of labels decides whether data can flow from \((S_1, I_1, D_1)\) to \((S_2, I_2, D_2)\)

\[(S_1, I_1, D_1) \sqsubseteq (S_2, I_2, D_2) \text{ iff } S_1 \subseteq S_2 \text{ and } I_2 \subseteq I_1\]
Information-flow Labels

Secret File Manager

- Secret Files
  - {FileSecret}, {FileWrite}, {-FileSecret}

Editor

- Edit Activity
  - {FileSecret}, { }, {}

Network API

- { }, {Internet}, { }

{FileSecret}, {FileWrite}, {-FileSecret}
Run-time Enforcement

Secret File Manager

{FileSecret}, {FileWrite}, {-FileSecret}

Editor

{FileSecret}, {}, {}

Check call

{FileSecret} ⊆ {FileSecret}
{FileWrite} ⊇ {}

Reference Monitor

{FileSecret} ⊆ {}?
{} ⊇ {Internet}?

Network API

{ }, {Internet}, {}
Floating Labels

Secret File Manager

Secret Files

{FileSecret}, {FileWrite}, {-FileSecret}

File Manager

Files

{ }, {FileWrite}, { }

Editor

Edit Activity

{FileSecret}, { }, { }

{ }, { }, { }
Floating Labels

Editor

Edit Activity

F{},
{}

{ }
Floating Labels

{FileSecret}, {FileWrite}, {-FileSecret}

F{} instantiated as {FileSecret}

{ } ⊆ {FileWrite}

Reference Monitor
Floating Labels (with Base Labels)

F{EditorSecret}, {Internet}, { }
Floating Labels (with Base Labels)

- **File Manager**
  - `{}`, `{Internet}`, `{}`

- **Editor**
  - `{EditorSecret}`, `{Internet}`, `{}`

- **Reference Monitor**
  - `{Internet} ⊆ {Internet}`
  - `{EditorSecret} ⊆ {Internet}`
  - `{Internet} ⊆ {Internet}`

- **Network API**
  - `{}, {Internet}, {}`
Static Labels vs Effective Labels

File Manager

{ }, {Internet}, {}

Editor

Edit Activity

F{EditorSecret}, {Internet}, {}

Run-time Instance of Edit Activity

{EditorSecret}, {Internet}, {}

Effective label

Static label

{ }, {Internet}, { }
Declassification

- Removes secrecy tags, adds integrity tags
  - Only succeeds if the declassification label allows it
  - E.g., declassify {FileSecret}
Raising

- Adds secrecy tags, removes integrity tags
  - Always succeeds
  - E.g., raise \{+contactSecret\}
Noninterference

- System with high components is observably equivalent to the system without high components.
- Label for components decides whether they are high or low:
  - High = attacker isn’t allowed to see
  - Low = attacker is allowed to see

Secret File Manager

- Secret Files
- {FileSecret}, {FileWrite}, {-FileSecret}

lower or equal to attacker’s label
Noninterference (Trace equivalence)

Noninterference:
- S is a valid configuration,
- $S_L$ contains all the low components of $S$,
- $S \Rightarrow (t_1) S'$ implies $\exists t_2$ such that $S_L \Rightarrow (t_2) S'_L$ and $t_1 \approx (L) t_2$,
- $S_L \Rightarrow (t_1) S'_L$ implies $\exists t_2$ such that $S \Rightarrow (t_2) S'$ and $t_1 \approx (L) t_2$.

What does this mean?
- No matter how the attacker manipulates the system, the attacker cannot learn any information about the secrets that the policy prohibits him from learning.
Prototype Implementation

App1

Package Manager
Android Perms

Static Data

Activity Manager

Reference Monitor

Instance Data

App1 App2

mediates inter-component communication to enforce additional policies

hooks in Linux kernel to mediate file system & network access

Java API

Kernel
Prototype Implementation

- System is implemented on Android v4.0.4 and tested on a Nexus S phone
- The impact on performance is unobservable to the user
Privilege Escalation

Information leakage
Today

- Application sandboxing
  + reference monitors in Android

- Hot research topic: How to specify and enforce policies that guard sensitive data and resources?
  - One option: Distributed information-flow control
  - Formal proofs support correctness of mechanisms

- Next two lectures:
  - A formal approach to ensuring control-flow integrity
  - A formal look at what policies are enforceable

- Later:
  - More about noninterference