Static Verification of Android Security

Michele Bugliesi
based on work with Stefano Calzavara and Alvise Spanò
appeared at FORTE/FMOODS Int. Conf. 2013

Università Ca’ Foscari Venezia
Dipartimento di Scienze Ambientali, Informatica e Statistica

ISACA VENICE – 27.9.2013
Mobile Devices All Over

Smartphones have become pervasive

- Increasingly widespread and involved in a variety of contexts:
  - Web browsing and social networking
  - Multimedia and entertainment
  - Navigation and Location Based Services
  - On-line payments
  - . . .

Smartphone security and robustness have therefore become critical
A variety of attack surfaces

- **Apple’s iOS**
  - Closed platform: secret vetting process for new applications
  - Largely (?) protected against malware and design flaws
  - Privacy concerns arising from closed nature of the platform

- **Google’s Android**
  - An open platform: loose control on flawed applications
  - Interesting case study for security research
  - Lots of design flaws and security attacks found over the last few years
Android Security

Main attack surfaces
- Information flow leaks
- Privilege escalation
- Coarse permissioning
- ICC mismatches

Countermeasures
- Data-flow analysis
- Runtime reference monitors
- OS enhancements
- Exception handling

We focus on privilege escalation and ICC mismatches
- Language-based security approach:
  - static verification, based on security typing
  - enforce certified secure design practices
  - assist robust application development
- Also effective for validation of existing applications
Android Architecture – in 1 slide

Applications consist of separate components

- **Activities**: associated with user interfaces
- **Services**: performing long-running background computations
- **Broadcast receivers**: acting as forwarders of system-wide broadcast messages to specific application components.
- **Content providers**: managing persistent application data.

Components communicate with intents

- **Explicit**: directed to a specific component
- **Implicit**: directed to any component supporting associated action

Exchanged asynchronously
Applications consist of separate components
- **Activities**: associated with user interfaces
- **Services**: performing long-running background computations
- **Broadcast receivers**: acting as forwarders of system-wide broadcast messages to specific application components.
- **Content providers**: managing persistent application data.

Components communicate with intents
- **explicit**: directed to a specific component
- **implicit**: directed to any component supporting associated action
- exchanged asynchronously
Android Security Model – in 1 slide

Sandboxing

- Applications mutually distrusted
- Application assigned unique IDs
- Applications may only access resources they own, or resources owned by others that are made publicly available
Android Security Model – in 1 slide

Sandboxing

- Applications mutually distrusted
- Application assigned unique IDs
- Applications may only access resources they own, or resources owned by others that are made publicly available

Permission system

- Sensitive components protected by permissions
  - PHONE_CALL, INTERNET, BLUETOOTH, ...
- Permissions required to run an application defined in the application’s manifest, and granted upon installation
- Runtime reference monitor to check permissions upon ICC calls
- Delegation via pending intents
Android insecurities – Privilege escalation

Intended ICC protection policy

- An application protected by a permission $P$ should only be invoked by applications owning $P$
Android insecurities – Privilege escalation

Intended ICC protection policy

- An application protected by a permission $P$ should only be invoked by applications owning $P$

```
App A
Granted: -
Requires: -
```
```
App B
Granted: $P$
Requires: -
```
```
App C
Granted: $P$
Requires: $P$
```

M. Bugliesi (UNIVE)

LINTENT

27.9.2013
Intended ICC protection policy

- An application protected by a permission $P$ should only be invoked by applications owning $P$

Security flaws in Android protection system

- Unprivileged caller may invoke a privileged callee, transitively acquiring privileges
- Pending intents may be employed to transfer permissions to unprivileged components
Android insecurities – Privilege escalation

A realistic threat [Felt et. al 2011]

“More than a third of the 872 surveyed Android applications request permissions for sensitive resources and also expose public interfaces; they are therefore at risk of privilege escalation”

“Found 15 permission re-delegation vulnerabilities in 5 core system applications”

Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]
inspect privileges in the ICC call chain, downgrade components as needed

Our approach is different
we opt for static, certified verification by typing
Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]

- inspect privileges in the ICC call chain,
- downgrade components as needed

```
App A
Granted: -
Requires: -

→

App B
Granted: P
Requires: -

→

App C
Granted: P
Requires: P
```
Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]

- inspect privileges in the ICC call chain,
- downgrade components as needed
Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]

- inspect privileges in the ICC call chain,
- downgrade components as needed

```
App A
Granted: -
Requires: -

App B
Granted: -
Requires: -

App C
Granted: P
Requires: P
```
Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]

- inspect privileges in the ICC call chain,
- downgrade components as needed

```
App A
Granted: -
Requires: -

App B
Granted: -
Requires: -

App C
Granted: P
Requires: P
```
Privilege escalation – Countermeasures

State of the art

- Existing solutions enforce runtime protection
  - require patches to OS
  - incur performance overhead
- Do not provide any certified guarantee

Example – ICC Inspection [Felt et. al 2011]

- inspect privileges in the ICC call chain,
- downgrade components as needed

Our approach is different

- we opt for static, certified verification by typing
LINTENT – Android source code analyzer

Implemented as an Android Lint plugin
- Full Eclipse integration

Detects Android API usage
- Component classes
- SDK method calls
- Special calling patterns

Helps developers write robust code
- Type errors, warnings and hints on the code
- A second-tier Java type-checker
LINTENT – Security checking

Analyzes
- Application permissions (manifest)
- IPC permissions used in source code
- Secrecy level of objects to track pending intents

Detects
- Attack surfaces for privilege escalation
- Over-privileged applications
- Runtime failures due to under-privilege
class MyActivity extends Activity
    ...

    // Create intent, requires BLUETOOTH
    Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

    // Create pending intent for i
    PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);

    // Create new intent
    Intent i2 = new Intent("ACT_STRING");

    // inject p into i2
    i2.putExtra("pending", p);

    // pending intent within i2 provides BLUETOOTH to any recipient
    startActivity(i2);
class MyActivity extends Activity
   
   // Create intent, requires BLUETOOTH: secrecy(i) = BLUETOOTH
   Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

   // Create pending intent for i
   PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);

   // Create new intent
   Intent i2 = new Intent("ACT_STRING");

   // inject p into i2
   i2.putExtra("pending", p);

   // pending intent within i2 provides BLUETOOTH to any recipient
   startActivity(i2);
class MyActivity extends Activity
    
    // Create intent, requires BLUETOOTH: secrecy(i) = BLUETOOTH
    Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

    // Create pending intent for i: secrecy(p) = secrecy(i)
    PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);

    // Create new intent
    Intent i2 = new Intent("ACT_STRING");

    // inject p into i2
    i2.putExtra("pending", p);

    // pending intent within i2 provides BLUETOOTH to any recipient
    startActivity(i2);
class MyActivity extends Activity

  // Create intent, requires BLUETOOTH: secrecy(i) = BLUETOOTH
  Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

  // Create pending intent for i: secrecy(p) = secrecy(i)
  PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);

  // Create new intent: secrecy(i2) = PUBLIC
  Intent i2 = new Intent("ACT_STRING");

  // inject p into i2
  i2.putExtra("pending", p);

  // pending intent within i2 provides BLUETOOTH to any recipient
  startActivity(i2);
class MyActivity extends Activity
   ...

   // Create intent, requires BLUETOOTH: secrecy(i)= BLUETOOTH
   Intent i = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

   // Create pending intent for i: secrecy(p) = secrecy(i)
   PendingIntent p = PendingIntent.getActivity(this, 0, i, 0);

   // Create new intent: secrecy(i2) = PUBLIC
   Intent i2 = new Intent("ACT_STRING");

   // inject p into i2: TYPE ERROR! secrecy(p) > secrecy(i2)
   i2.putExtra("pending", p);

   // pending intent within i2 provides BLUETOOTH to any recipient
   startActivity(i2);
LINTENT – Inter-component communication

Analyzes

- Component classes
- Inter-component dataflow in message passing (within intents)

Detects

- Mismatches in inter-component communication
- Undisciplined programming practices
LINTENT – Inter-component communication

A real problem [Maji et. al. 2011]

- Extensive testing, based on 9000+ randomly generated intents sent to 450+ components in Android 4.0 builtin applications, and 5 most popular applications on Google Play.

- Around “5% - 8% ICC crashes, depending on component type, due to uncaught exceptions”. Corresponding to a total of “641 crashes in Android 4.0, 152 in Google Play apps”

class SenderActivity extends Activity

    protected void onCreate(Bundle savedInstanceState)
        ...

        // create an explicit intent
        Intent i = new Intent(this, ReceiverActivity.class);

        // populate with primitive and user-defined
        i.putExtra("k1", 3);
        i.putExtra("k2", true);
        i.putExtra("k3", new SomeSerializable());

        // start receiver
        startActivity(i);
public class ReceiverActivity extends Activity

    protected void onCreate(Bundle savedInstanceState)
    {
        // retrieve the intent
        Intent i = getIntent();

        // retrieve first extra component
        String v1 = i.getStringExtra("k1");

        // retrieve third extra component
        WrongSerializable o = (WrongSerializable)i.getSerializableExtra("k3");

        ...
public class ReceiverActivity extends Activity

    protected void onCreate(Bundle savedInstanceState)

        // retrieve the intent
        Intent i = getIntent();

        // retrieve first extra component TYPE ERROR - k1: int
        String v1 = i.getStringExtra("k1");

        // retrieve third extra component
        WrongSerializable o = (WrongSerializable)i.getSerializableExtra("k3");

        . . .
public class ReceiverActivity extends Activity

    protected void onCreate(Bundle savedInstanceState)

        // retrieve the intent
        Intent i = getIntent();

        // retrieve first extra component TYPE ERROR - k1: int
        String v1 = i.getStringExtra("k1");

        // retrieve third extra component TYPE ERROR - k3: SomeSerializable
        WrongSerializable o = (WrongSerializable)i.getSerializableExtra("k3");

        ...
LINTENT – Architecture

Our implementation is an ADT Lint plugin
LINTENT – Features

- Full supports for Java 1.6
- Third-party libraries (external JARs) inspection
- Dataflow for intents, pending intents and bundles
  - Support for recursive nesting
  - Pending intents injected within intents and viceversa
LINTENT – First experiments

Type-checked existing apps from Google Play store

○ APN-Switch
  ▶ Any application can send an intent to turn the network on/off

○ Wifi-Fixer
  ▶ Any application can send an intent to turn the wifi on/off
  ▶ difficult to notice by manual inspection
  ▶ not ideal for a fixer, ain’t it?

Both flaws uncovered by LINTENT and then verified manually
LINTENT – Further experiments

- Preliminary results over a small number of apps
  - Tested on 11 open-source apps
  - We’re soon ready for massive experimentation
LINTENT – Further experiments

- Preliminary results over a small number of apps
  - Tested on 11 open-source apps
  - We’re soon ready for massive experimentation
- Among self-contained apps with UI components
  - 85% of intents are explicit
  - 60% of warnings and errors relate to intent mistyping
- Among Service-based apps
  - over 95% intents are implicit
  - LINTENT cannot help much with these
- Among apps that use permissions
  - 30% are over-privileged
  - 10% accidentally incur privile escalation via PendingIntent within an Intent
  - 25% are under-privileged based on Broadcast Receivers
LINTENT – Further experiments

- Preliminary results over a small number of apps
  - Tested on 11 open-source apps
  - We’re soon ready for massive experimentation

Among self-contained apps with UI components
  - 85% of intents are explicit
  - 60% of warnings and errors relate to intent mistyping

Among Service-based apps
  - over 95% intents are implicit
  - LINTENT cannot help much with these
LINTENT – Further experiments

- Preliminary results over a small number of apps
  - Tested on 11 open-source apps
  - We’re soon ready for massive experimentation

- Among self-contained apps with UI components
  - 85% of intents are explicit
  - 60% of warnings and errors relate to intent mistyping

- Among Service-based apps
  - over 95% intents are implicit
  - LINTENT cannot help much with these

- Among apps that use permissions
  - 30% are over-privileged
  - 10% accidentally incur privile escalation
    - via PendingIntent within an Intent
  - 25% are under-privileged
    - based on Broadcast Receivers
Conclusions

Enhancing the Android development process highly desired

- type-based analysis is possible and useful, though demanding
- certified security is still far away, but LINTENT is a first step in that direction
Conclusions

Enhancing the Android development process highly desired

- type-based analysis is possible and useful, though demanding
- certified security is still far away, but LINTENT is a first step in that direction

What’s next

- continue engineering of prototype, extensive testing
- integrate a front-end to a decompiler
- re-target to DALVIK code
- support for robust declassification/endorsement
Thanks


