Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications

Daniele Gallingani, Rigel Gjomemo, V.N. Venkatakrishnan, Stefano Zanero
Android apps are composed of different components.

*Intents* carry messages among components and applications.

Components declare the types of intents they are willing to receive.

Intents can be sent explicitly or implicitly.
Motivation

**Problem**: Android Components have no message origin verification capabilities

An attacker can spoof legitimate intents and send malicious input

**Questions**

- Could we check if applications validate input?
- If so, can we automatically generate exploit opportunities?
Contributions

• Static analysis method to automatically detect data flows leading to sensitive operations
  – Formulation of the problem as an IFDS problem
• Method for automatically generating exploits that trigger malicious behavior
• Results
  – Automatically generated exploits for 26 applications and showed they are vulnerable to user interface spoofing attacks
Outline

• Problem Statement
• Approach
• Implementation
• Results
Problem Statement

String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url = "http://www.example.com";
if (host.contains("example.com"))
    url = "http://" + host + "/";
if (file.contains(".."))
    file = file.replace("..", "");
String httpPar = toBase64(file);
...
DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);
...
httpC.execute(get);
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url = "http://www.example.com";
if (host.contains("example.com"))
    url = "http://" + host + "/";
if (file.contains(".."))
    file = file.replace("..", "");
String httpPar = toBase64(file);

DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);

httpC.execute(get);
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url = "http://www.example.com";
if (host.contains("example.com"))
    url = "http://" + host + "/";
if (file.contains(".."))
    file = file.replace("..", "");
String httpPar = toBase64(file);

DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);

httpC.execute(get);
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url = "http://www.example.com";
if (host.contains("example.com"))
    url = "http://" + host + ";
if (file.contains(".."))
    file = file.replace("..", "");
String httpPar = toBase64(file);

DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);
...
httpC.execute(get);

• Finding paths from sources to sinks is not sufficient

• Question: Are those paths feasible for an attack?
Approach

• Input state: $V_i$

• Exploit state(s): $V_e$
  Value patterns related to sinks

• Find relationship $F$ between $V_i$ and $V_e$, such that $V_i = F(V_e)$

![Diagram showing source and sink nodes with value patterns]

Source $V_i = \{(v_1, c_1), ..., (v_n, c_n)\} = F(V_e)$

Sink $V_e = \{(v_{e1}, c_{e1}), ..., (v_{em}, c_{em})\}$
Approach Overview
Approach Overview

- Path Computation
  - Find all paths from sources to sinks
Approach Overview

• Path Computation
  – Find all paths from sources to sinks

• Symbolic Execution
  – Generate a symbolic formula $F_p$

\[ C_1 \land C_2 \land C_3 = F_p \]
**Approach Overview**

- **Path Computation**
  - Find all paths from sources to sinks

- **Symbolic Execution**
  - Generate a symbolic formula \( F_p \)

- **Exploit generation**
  - Solve \( F_p \land V_e \rightarrow V_l \)
Path Computation

- Supergraph contains CFGs of all the functions
- Taint Propagation
  - Identifies statements that can be influenced by attacker
  - Reduces size of the problem
Implementation (Background)

• Path Computation: IFDS framework (Soot&Heros)
  – Transforms dataflow problems into graph reachability problems
  – Framework user defines a fact
  – Framework user defines update rules for a fact

• Exploit Generation: Kaluza
  – Efficient string solver
  – Native support for many string operations
Implementation

• **Path Computation**
  – A fact contains path and taint information for every node
  – Different rules update the fact information during graph traversal

• **Exploit Generation**
  – Translate $F_p \land V_e$ into a Kaluza formula
  – Additional string operations modeled using the Kaluza language
    E.g.,: $a\text{.contains(“test”)} \rightarrow a \text{ in CapturedBrack}(/.*test.*/);
Results Overview

• 64 applications of different sizes
  – 26 exploits generated and manually verified
• Sink statements: GUI operations
• $V_e$ chosen to change apps GUls (phishing)
• Different GUI targets
  – Entire screen change
  – Alerts screen change
  – User input fields
  – Other Components
## Results

<table>
<thead>
<tr>
<th>App</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mint</td>
<td>Display an arbitrary web page inside an Activity</td>
</tr>
<tr>
<td>GoSMS</td>
<td>Prompt to the user notification about a new message with arbitrary sender and SMS content</td>
</tr>
<tr>
<td>GoSMS</td>
<td>Prompt notification about a new message received with arbitrary sender and receiver</td>
</tr>
<tr>
<td>Yelp</td>
<td>Modify venue review draft screen and enter review on behalf of the user</td>
</tr>
<tr>
<td>Poste Pay</td>
<td>Modify and show the application prompt alerts with arbitrary messages</td>
</tr>
<tr>
<td>Craigslist</td>
<td>Change the Action Bar title, compromising the interface integrity</td>
</tr>
</tbody>
</table>

**Entire Screen**

**User Input**

**Alert Screen**

**Other Components**
## Results

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-application execution time</td>
<td>2.4 min</td>
<td>33.2 min</td>
<td>12.3 min</td>
</tr>
<tr>
<td>Per-application components</td>
<td>3</td>
<td>31</td>
<td>24.5</td>
</tr>
<tr>
<td>Per-application vulnerable paths</td>
<td>2</td>
<td>19</td>
<td>4.2</td>
</tr>
<tr>
<td>Per-path statements</td>
<td>5</td>
<td>81</td>
<td>17.2</td>
</tr>
<tr>
<td>Per-path if-statements</td>
<td>0</td>
<td>3</td>
<td>0.98</td>
</tr>
</tbody>
</table>

- Very few validation checks present
  - Mostly null pointers
- 31% of the String library functions approximated with Kaluza
Limitations

• Untainted variables contribute to application state. May introduce false positives

• Solver approximations. May introduce false positives
Conclusions

• Conclusions
  – We present an automatic method to discover vulnerable paths inside Android application components
  – Our method is modelled as an IFDS problem
  – We provide proofs for the vulnerabilities under the form of actual exploits, generated automatically.
Questions?