Firefox Mobile Crashreporter Path Traversal

Roee Hay
IBM Security Systems
roeeh@il.ibm.com

***Abstract*—We present newly found Path Traversal vulnerabilities in the Firefox Android App. The vulnerabilities can be exploited by a malicious application running on the device to subvert the confidentiality and integrity of Firefox. We show how a malicious application can exploit the vulnerabilities to subvert Firefox for leaking sensitive files.**

I. Android basics

A. Threat model

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 which cannot be accessed by third-party apps. An android app may request specific privileges (permissions) during its installation; if granted by the user, the app's capabilities are extended. Permissions are defined under the application's manifest file (`AndroidManifest.xml`).

B. Activities and Services

Android apps are composed of application components of different types including activities and services. An Activity, implemented by the `android.content.Activity` class [1], defines a single UI, e.g. A browsing window or a preferences screen. Services [2] are applications components which are used for background tasks.

C. Inter-Process Communication (IPC) and Intents

Android applications make heavy use of IPC. This is achieved by Intents. These are messaging objects which contain several attributes such as an action, data, category, target and extras. The data attribute is a URI which identifies the intent (e.g. tel:0422123). Each Intent can also contain extra data fields (aka Intent extras) which reside inside a bundle (implemented by the `android.os.Bundle` class [3]). These extra fields can be set by using the `Intent.putExtra` API or by manipulating the extras bundle directly. It is important to emphasize that intents provide a channel for a malicious app to inject malicious data into a target, potentially vulnerable app. Intents can be sent anonymously (implicit intents, i.e. target is not specified) and non-anonymously (explicit intents, target is specified). Intents 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). Under the application's manifest file, an application component may claim whether it can be invoked externally using an Intent, and if so which set of permissions is required. We define a *public* application component as one which can be invoked externally by a (potentially malicious) application without a required set of permissions. All other components are defined as *private*, i.e. they can only be invoked by other application components of the same package, or externally with adequate privileges.

II. The Crash Reporter Normal Operation

The `org.mozilla.gecko.CrashReporter` class is a public activity. Its purpose is to send crash dumps to Mozilla when needed (Figure II.1). The CrashReporter activity receives the dump file path as an input (an Intent extra parameter). When the activity is launched, its `onCreate` method (Figure II.2) is executed and the following actions take place:

1) Using the `movefile` (Figure II.3) method, the given minidump file is moved to the pending minidumps path, `files/mozilla/Crash Reports/pending`.
2) A meta-data filename is deduced from the given minidump file path, by replacing all '.dmp' occurrences with '.extra'. i.e. `<filename>.dmp` becomes `<filename>.extra`.
3) The meta-data file is moved to the pending minidumps path using the `moveFile` method .

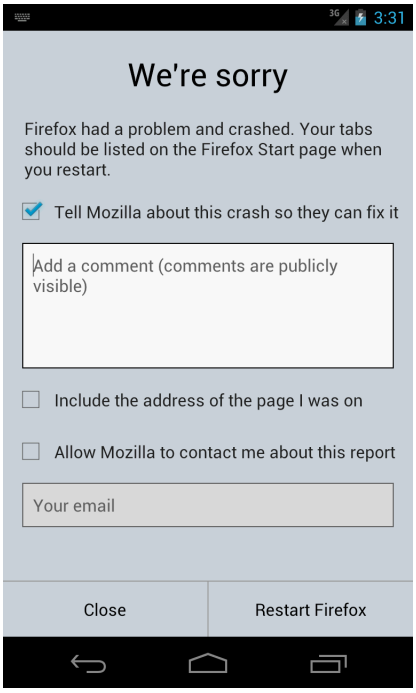

Figure II.1. Crash Reporter dialog

4) The meta-data file is parsed as a '<key>=<value>' file format (Figure II.4). The target server URL (i.e. the one which the crash information is sent to) is specified here using the `ServerURL` key.
5) The crash dialog is present to the user (Figure II.1).

If the user presses the *Close* or *Restart* buttons with the *Send report* check-box enabled, the minidump alongside with other sensitive information is sent to the specified server by calling the `sendReport` method. It should be noted that if the user has also checked the *Include URL* check-box, then Android logs are sent as well.

III. Vulnerabilities

The `CrashReporter` activity consumes the minidump path from the input intent (Figure II.2, Line 4) although it should be considered untrusted data, since the activity is public as defined in the Android Manifest file. Therefore, a malicious application can control the source path of the moved minidump file and the deduced extra file (Figure II.2, Lines 8, 11). We will see in the next section that this allows that attacker to control the destination server for the crash report, hence the response value used as a file path in Figure II.5, Line 19 should be untrusted as well.

IV. Possible Exploitation & Impact

Generally a path traversal weakness affects both the confidentiality and integrity of the vulnerable app. In the next subsections we will see concrete exploits for Firefox.

A. Information Disclosure

The attacker can exploit the Path Traversal vulnerability in order to steal sensitive files. Concretely we will demonstrate how he can leak one of the cache files to an arbitrary server.

1) Phase 1: Profile directory path leakage: Since the cache file resides under the user's profile directory which contains random bytes (`files/mozilla/<random_bytes>.default`), the first step would be to leak its path. Luckily Firefox writes the path to the Android Log:

```
D/GeckoProfile( 4766): Found profile dir:
/data/data/.../files/mozilla/24pd90uh.default
```

In *Android 4.0* and below, the Android log can easily read by all apps including the malicious one, by simply acquiring the `android.permission.READ_LOGS` permission. *Android 4.1* has introduced a change to this behavior to prevent such log leakage attacks: The `READ_LOG` permissions is now not required, however applications can only listen their own logs. Since Firefox sends its log alongside the crash dump report, it can be abused for sending the logs to the attacker by adhering to the following protocol:

- The malicious app creates *two* world readable files under its data directory: `/data/data/<malicious app>/foo.dmp` with an arbitrary content (can be left empty) and `/data/data/<malicious app>/foo.extra` which contains the attacker's server, `ServerURL=http://<attacker>/.`
- The malicious app generates an Intent object which targets the `CrashReporter` activity with a `minidumpPath` parameter set to `/data/data/<malicious app>/foo.dmp`

After the above actions take place, according to the operation of the `CrashReporter` activity (Section II), Firefox will move `foo.dmp` and `foo.extra` from the malicious app data directory to the Firefox crash reports pending path and then parse `foo.extra` for key-value pairs. Since the extra file now contains the attacker's IP as the `ServerURL`, if the user pressed one of the buttons on the `CrashReporter` dialog, Firefox would send the crash report to the attacker. If the user has also checked the *Provide IP address* check-box, then Firefox would also append the Android Log output, which contains the Firefox **private** logs which included the random profile directory

name. It should be noted that since Firefox only sends a 200 lines window of the log, the leaked path can be out of that window since it is printed to log upon Firefox's launch. In order to make sure that the path is within the window, the attacker can restart Firefox by crashing it. This can be easily done by invoking the `CrashReporter` activity with a *null* `minidumpPath` parameter.

Another approach to deduce the profile dir path stems from a different vulnerability in the profile dir generation and is explained in [4].

2) Phase 2: Data leakage: Firefox stores under the profile directory sensitive data such as the user's cookies and cache. Once the attacker has learnt the profile dir path he can leak these data with two different methods:

1) If the target file's MIME type is not HTML based, Firefox will automatically download the file to the Download directory under the sdcard (`/mnt/sdcard/Download`), which can be read by the malicious applications.
2) The second method is to indirectly inject the `ServerURL=http://<attacker>` string to the target file and trick Firefox to use this file as the minidump extra. For example, by using this method, the attacker can leak the cache. First, he opens Firefox (using an Intent) on an attacker's controlled website, which contains the above string in its HTML body (Figure IV.1). After the cache has been prepared, the attacker can leak it by exploiting the same path traversal vulnerability in the `CrashReporter` activity. He simply generates another Intent which targets the `CrashReporter` activity, with the `minidump` Path parameter set to the cache file. Since the cache file has no '.dmp' sub-string, the computed extra file will be the same, and the target server URL will be parsed out of the cache file!.

B. Denial-of-Service

A malicious app can supply an arbitrary filename or directory name as the minidump file path. Thus it can render Firefox nonoperational since any supplied file will be moved to the 'pending' path.

```
1 <HTML>
2 <BODY>
3 ServerURL=http://<ATTACKER>/post.php
4 </BODY>
5 </HTML>
```

Figure IV.1. Injected payload into Firefox's cache

V. Vulnerable versions

Firefox 25.0.1 has been found vulnerable.

References

[1] Activity class reference. http://developer.android.com/reference/android/app/Activity.html.
[2] Service class reference. http://developer.android.com/reference/android/app/Service.html.
[3] Bundle class reference. http://developer.android.com/reference/android/os/Bundle.html.
[4] Roee Hay. Derandomizing Firefox Profiles, 2013.

```
1  @Override
2  public void onCreate(Bundle savedInstanceState) {
3  ...
4    String passedMinidumpPath = getIntent().getStringExtra(PASSED_MINI_DUMP_KEY);
5    File passedMinidumpFile = new File(passedMinidumpPath);
6    File pendingDir = new File(getFilesDir(), PENDING_SUFFIX);
7    pendingDir.mkdirs();
8    mPendingMinidumpFile = new File(pendingDir, passedMinidumpFile.getName());
9    moveFile(passedMinidumpFile, mPendingMinidumpFile);
10   File extrasFile = new File(passedMinidumpPath.replaceAll(".dmp", ".extra"));
11   mPendingExtrasFile = new File(pendingDir, extrasFile.getName());
12   moveFile(extrasFile, mPendingExtrasFile);
13   mExtrasStringMap = new HashMap<String, String>();
14   readStringsFromFile(mPendingExtrasFile.getPath(), mExtrasStringMap);
15 ...
16 }
```

Figure II.2. `onCreate` method under `CrashReporter`

```
1  private boolean moveFile(File inFile, File outFile) {
2    Log.i(LOGTAG, "moving " + inFile + " to " + outFile);
3    if (inFile.renameTo(outFile))
4      return true;
5    try {
6      outFile.createNewFile();
7      Log.i(LOGTAG, "couldn't rename minidump file");
8      // so copy it instead
9      FileChannel inChannel = new FileInputStream(inFile).getChannel();
10     FileChannel outChannel = new FileOutputStream(outFile).getChannel();
11     long transferred = inChannel.transferTo(0, inChannel.size(), outChannel);
12     inChannel.close();
13     outChannel.close();
14     if (transferred > 0)
15       inFile.delete();
16   } catch (Exception e) {
17     Log.e(LOGTAG, "exception while copying minidump file: ", e);
18     return false;
19   }
20   return true;
21 }
```

Figure II.3. `moveFile` method under `CrashReporter`

```
1  private boolean readStringsFromFile(String filePath, Map<String, String> stringMap) {
2    try {
3      BufferedReader reader = new BufferedReader(new FileReader(filePath));
4      return readStringsFromReader(reader, stringMap);
5    } catch (Exception e) {
6      Log.e(LOGTAG, "exception while reading strings: ", e);
7      return false;
8    }
9  }
10
11 private boolean readStringsFromReader(BufferedReader reader, Map<String, String> stringMap)
12 throws IOException {
13   String line;
14   while ((line = reader.readLine()) != null) {
15     int equalsPos = -1;
16     if ((equalsPos = line.indexOf('=')) != -1) {
17       String key = line.substring(0, equalsPos);
18       String val = unescape(line.substring(equalsPos + 1));
19       stringMap.put(key, val);
20     }
21   }
22   reader.close();
23   return true;
24 }
```

Figure II.4. File parsing under `CrashReporter`

```
1  private void sendReport(File minidumpFile, Map<String, String> extras, File extrasFile) {
2    final CheckBox includeURLCheckbox = (CheckBox) findViewById(R.id.include_url);
3    String spec = extras.get(SERVER_URL_KEY);
4  ...
5    try {
6      URL url = new URL(spec);
7      HttpURLConnection conn = (HttpURLConnection)url.openConnection();
8  ...
9      if (Build.VERSION.SDK_INT >= 16 && includeURLCheckbox.isChecked()) {
10       sendPart(os, boundary, "Android_Logcat", readLogcat());
11     }
12 ...
13     sendFile(os, boundary, MINI_DUMP_PATH_KEY, minidumpFile);
14 ...
15     if (conn.getResponseCode() == HttpURLConnection.HTTP_OK) {
16       File submittedDir = new File(getFilesDir(), SUBMITTED_SUFFIX);
17 ...
18       String crashid = responseMap.get("CrashID");
19       File file = new File(submittedDir, crashid + ".txt");
20       FileOutputStream fos = new FileOutputStream(file);
21       fos.write("Crash ID: ".getBytes());
22       fos.write(crashid.getBytes());
23       fos.close();
24     }
25 ...
26   }
27 ...
28 }
```

Figure II.5. `sendReport` under `CrashReporter`