# DERANDOMIZING FIREFOX PROFILES

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

***Abstract*—We present a newly found vulnerability in the Firefox Android App. This vulnerability allows a malicious app to to guess correctly the Firefox profile directory name in a practical amount of time. Resolving the profile directory name enables a malicious app to leak any file which resides in that directory (such as cookies and cached information) thus it breaks Android's sandbox.**

## 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 MATH.RANDOM() PRNG

The `Math.random()` static method returns a uniformly distributed pseudo-random number between 0 and 1. In Android, the implementation is under `libcore`[1]:

```
1  public static synchronized double random()
2  {
3    if (random == null)
4    {
5      random = new Random();
6    }
7    return random.nextDouble();
8  }
```

Figure II.1. `Math.random()`

Let's deep-dive into the origins of the seed of the PRNG. We can see that this method relies on the `java.util.Random` class which is seeded in its default constructor. In Android 4.4 and below, it is implemented as follows[2]:

[1] https://android.googlesource.com/platform/libcore/
[2] https://android.googlesource.com/platform/libcore/+/android-4.4_r1.2/luni/src/main/java/java/util/Random.java

```
1  private static String
2          saltProfileName(String name)
3  {
4    String allowedChars =
5    "abcdefghijklmnopqrstuvwxyz0123456789";
6
7    StringBuilder salt =
8            new StringBuilder(16);
9
10   for (int i = 0; i < 8; i++) {
11     salt.append(allowedChars.charAt((int)
12    (Math.random() * allowedChars.length())));
13   }
14   salt.append('.');
15   salt.append(name);
16   return salt.toString();
17 }
```

Figure III.1. `GeckoProfile.saltProfileName(String name)`

```
1  public Random()
2  {
3     setSeed(System.currentTimeMillis()
4        + System.identityHashCode(this));
5  }
```

Figure II.2. `java.util.Random`'s constructor

Therefore the seed depends on a couple of values:

1) The current time in milliseconds precision (`System.currentTimeMillis()`)
2) The identity hash code of the Random object (`System.identityHashCode(this)`)

`System.identityHashCode` is implemented by native code in the Dalvik VM implementation[3]. The identity hash code value is simply the object's virtual address which resides in the heap of the Dalvik VM process.

## III. Firefox Profile Directories

Firefox stores the personal data under the profile directory, located at `files/mozilla/X.default/` where `X` is a randomly chosen word of the language `[a-z0-9]{8}`. The generation of `X.default` is implemented by the `saltProfileName` function the `GeckoProfile` class (Figure III.1).

Since the profile directory name is random, Firefox stores a mapping under the `files/mozilla/profiles.ini` file (Figure III.2)

If Firefox does not find a valid profile, for example due to a missing profiles.ini, it creates a new one by calling the `createProfileDir` function under the `GeckoProfile` class.

[3]https://android.googlesource.com/platform/dalvik.git/+/android-4.4_r1.2/vm/Sync.cpp

```
[Profile0]
Default=1
Name=default
IsRelative=1
Path=475jbgu6.default
[General]
StartWithLastProfile=1
```

Figure III.2. `profiles.ini`

## IV. Vulnerability

`saltProfileName` uses `Math.random()` which is cryptographically insecure. We have seen that its seed relies on the inner-Random object creation time (in *ms* precision) and its Virtual Address (VA). Both factors are not random. The creation time can be leaked by an adversary and the VA lacks randomness due to missing ASLR in the Dalvik VM process. Since the Dalvik VM is forked from Zygote process, the VA of the Dalvik Heap is the same for all Android Apps. To conclude, the seed is not random, thus the profile directory name entropy is *far* from the ideal 41.36 random bits ($\log_2 36^8$) and can be predicted by the adversary as we will see next.

## V. Exploitation

Our attacker is a malicious app. We will show how it can exploit this vulnerability in order to leak any file under the Firefox profile directory.

### A. Derandomizing the seed

Let $Y$ be the chosen seed by Firefox. The goal of the malicious app is to derandomize it by recording some value which shares a strong relation with it. Its VA part cannot be deterministically, however most of its bits can be leaked easily by the malicious app by simply querying its own process using the `System.identitiyHashCode` routine on some object. The *ms* time factor cannot be deterministically determined, however most of its bits can be leaked in two different ways. First, if Firefox had been installed after the malicious app, the latter can record the first Firefox run by simply monitoring the device's process list. Otherwise the malicious app can exploit another vulnerability in Firefox [4] to move the *profiles.ini* file to the different directory. Firefox will create a new profile on its next run, which can be forced by crashing it (using the same vulnerability in the Crash Reporter). Again the attacker can record the Firefox launch time. Let $X$ be the attacker's inaccurate sample where $X = VA + MeasuredTime$. Let $\varepsilon$ be the error, i.e. $\varepsilon = Y - X$. In our brute-force attack we try values in the order of their inferred probabilities

which can be computed *offline* by the attacker. Let $N$ be the number of tries till success, The expected number of tries is $\mathbb{E}(N) = 1 + n - \sum_{k=1}^{n} k \cdot p_{(k)}$ where $\{p_k\}_{k=1}^{n}$ are the inferred probabilities. A simple model is to assert that $\varepsilon$ is a (discrete) bell-curve with the center estimated by $\bar{\varepsilon} = \bar{x}\text{-}\bar{y}$. It can be shown that $\mathbb{E}(N) = O(\mathbb{E}|X - \mathbb{E}X|) = O(\sigma_\varepsilon)$ so a more narrow bell-curve yields a shorter attack time (in average).

### B. Generating the candidate profile names

By using the inferred probabilities of $\varepsilon$ and the sampled seed, $X$, the malicious app creates a list of profile names by mimicking the `saltProfileName` implementation (this computation can be also done off the device and downloaded from the attacker).

### C. A Smart Brute-force

This phase is *online*. The malicious app creates a specially crafted world-readable HTML file and commands Firefox to load it (by using an Intent). The input to this file are the list of profile names ordered by their probabilities generated earlier. The JavaScript code in the HTML file goes over the list, searching for the correct profile. When there is a match, it can download any file under the profiles directory by creating an `iframe` with the filename as its source. If Firefox cannot render the file, it will automatically download the file to the sdcard (`/mnt/sdcard/Download`), a folder which can be read by the malicious app!

## VI. Evaluation

We tested our exploit on Firefox 25.0.1 running on Samsung GT-I9500 Galaxy S4 device equipped with Android 4.2.2. In order to infer the probabilities of $\varepsilon$ we ran 404 independent runs of the following test:

1) As the malicious app, call Math.Random() and retrieve its identityHashCode (Figure VI.1)
2) Start monitoring the process list in a frequency of 20 Hz for recording the Firefox launch time.
3) Remove the profile.ini file from the Firefox directory by exploiting using the Crash Reporter vulnerability
4) Record the created profile directory
5) Kill Firefox
6) Restart Firefox
7) Print the needed values.

For example, for each run we received and recorded the following data:

```
Sampled addr:     42dc6cd8
Sampled Time:     1385337348079
```

```
1 Math.random();
2 Field f = Math.class.
3             getDeclaredField("random");
4 f.setAccessible(true);
5 Random r = (Random)f.get(null);
6 Long addr = System.identityHashCode(r);
```

Figure VI.1. Retrieving the identityHashCode of the Math inner Random object

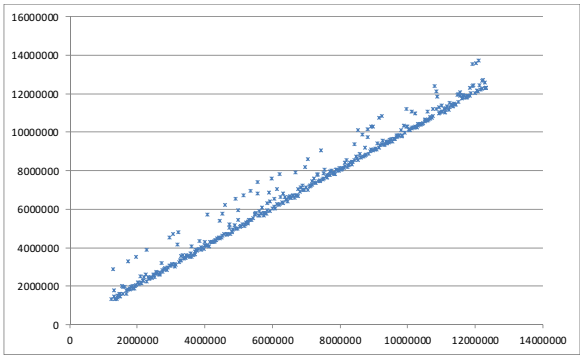


Figure VI.2. Linear connections between the real seed and the sampled one

```
Sampled Seed:     142cf66ccc7
Created profile: xa1x453r.default
```

Later we calculated the real seed and $\varepsilon$ by an offline brute-force. For example, for the data above, the real seed is 1386459232784 and $\varepsilon = 142665$.

We witnessed three strong linear connections between $Y$ and $X$ (Figure VI.2). We believe that the reason for the less dense lines rely on Dalvik Heap internals which caused the offset to 'jump' with a fixed number for a few tests (i.e. the probability for this to happen is low).

We created a histogram for inferring of $\varepsilon$ for the probabilities by their frequencies (Figure VI.3) and calculated the expected number of tries, $\mathbb{E}(N)$, which is 152779.65 and is equivalent to 17.22 bits. Note that the Shannon entropy is very close, 18.63 bits. We've got an entropy which is much lower than the ideal 64 bits (the java `long` primitive size) seed entropy, thus the attack is feasible.

We then adhered to the steps of section V, taken the simpler approach where we assert a symmetric bell curve. Since we received three linear connections, we have taken a slightly different approach. We denote $p_1, p_2, p_3$ as the probabilities for the error to be taken from each curve and assert that each error is normally distributed with its respective variance (the actual discrete error is the rounded value, see Figure VI.4). Therefore $\varepsilon$ is $\varepsilon_1 \sim \mathcal{N}\left(0, \sigma_1^2\right)$ in probability $p_1$ , $\varepsilon_2 \sim \mathcal{N}\left(0, \sigma_2^2\right)$ in probability $p_2$ and $\varepsilon_3 \sim \mathcal{N}\left(0, \sigma_3^2\right)$

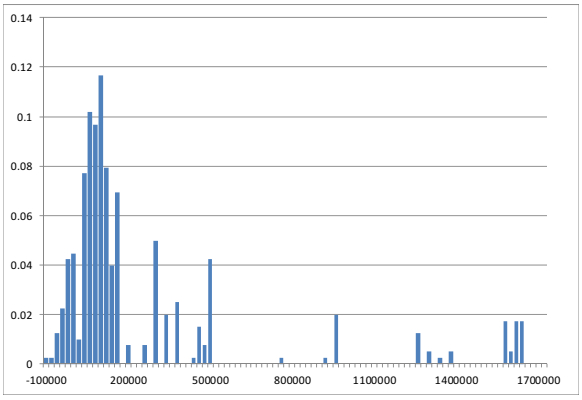


Figure VI.3. $\varepsilon$ Histogram

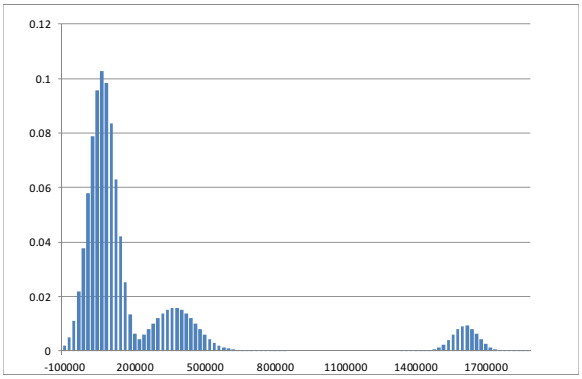


Figure VI.4. Normalized$-\varepsilon$ histogram

in probability $p_3$. We brute-force in the order of the probabilities. We ran our exploit and successfully caused Firefox to download sensitive files into the sdcard which can be read by the malicious app (Figure VI.5). It should be noted that according to the 'normalized' model, the expected number of tries is 214373.3 which is equivalent to 17.709 bits.

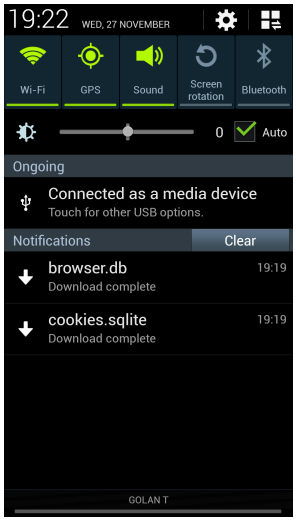


Figure VI.5. Malicious file download

## VII. 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. Firefox Mobile Crash Reporter Path Traversal, 2013.