IOActive Security Advisory

Title	Android (AOSP) User Dictionary Content Provider Authorization Bypass (CVE-2018-9375)
Severity	Moderate
Discovered by	Daniel Kachakil
Advisory Date	August 2, 2018

Affected Products

Android Open Source Project (AOSP)

Android versions: 6.0, 6.0.1, 7.0, 7.1.1, 7.1.2, 8.0, and 8.1.

Impact

A malicious application without any permission could perform the following actions on the user personal dictionary:

- Retrieve all entries
- Update all entries
- Delete all entries

Background

Android provides a user personal dictionary, which stores the spellings of non-standard words that the user wants to keep. This dictionary is usually accessed from “Settings → Language & keyboard → Personal dictionary” and may contain sensitive information, such as names, addresses, phone numbers, emails, passwords, or even credit card numbers.

While in theory access to the user’s personal dictionary should only be granted to privileged accounts, authorized Input Method Editors (IMEs), and spell checkers, IOActive identified a way to bypass some of these restrictions. By exploiting this vulnerability, a malicious application can update, delete, or even retrieve all its contents without requiring permission or user interaction.

Technical Details

In older versions of Android, access to the personal dictionary was protected by the following permissions:

- `android.permission.READ_USER_DICTIONARY`
- `android.permission.WRITE_USER_DICTIONARY`

After a commit in the source code pushed to the repository on June 23rd, 2015, this behavior changed.[1] The permissions were replaced by an internal check, so, theoretically, only privileged accounts (such as `root` and `system`), the enabled IMEs, and spell checkers could access the user personal dictionary content provider (`content://user_dictionary/words`).

In that change, a new private function named `canCallerAccessUserDictionary()` was introduced and was invoked from all the standard `query()`, `insert()`, `update()`, and `delete()` functions in this content provider, to prevent unauthorized calls to these functions.

While the change seems to be effective in both the `query()` and `insert()` functions, the authorization check occurs **too late** in `update()` and `delete()`, introducing a security vulnerability that allows any application to successfully invoke the affected functions via the exposed content provider, therefore bypassing the misplaced authorization check.

In the following code for the `UserDictionaryProvider` class,[2] pay attention to the highlighted fragments and see how the authorization checks are performed **after** the database is already altered:

```
@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case WORDS:
            count = db.delete(USERDICT_TABLE_NAME, where, whereArgs);
            break;

        case WORD_ID:
            String wordId = uri.getPathSegments().get(1);
            count = db.delete(USERDICT_TABLE_NAME, Words._ID + "=" +
wordId
                    + (!TextUtils.isEmpty(where) ? " AND (" + where + ')'
: ""), whereArgs);
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
```

[1] Gerrit's Change-Id: I6c5716d4d6ea9d5f55a71b6268d34f4faa3ac043
https://android.googlesource.com/platform/packages/providers/UserDictionaryProvider/+/e0e0e0c9b27e10f7e33c371c490fdae8b634f117

[2] At the time of discovery, it was found in the AOSP master branch:
https://android.googlesource.com/platform/packages/providers/UserDictionaryProvider/+/master/src/com/android/providers/userdictionary/UserDictionaryProvider.java
After the fix, the equivalent contents can be found in the following commit:
https://android.googlesource.com/platform/packages/providers/UserDictionaryProvider/+/0272515f306cf7f12d2babe03d625850990a00c1/src/com/android/providers/userdictionary/UserDictionaryProvider.java

```
    }

    // Only the enabled IMEs and spell checkers can access this provider.
    if (!canCallerAccessUserDictionary()) {
        return 0;
    }

    getContext().getContentResolver().notifyChange(uri, null);
    mBackupManager.dataChanged();
    return count;
}
```

```
@Override
public int update(Uri uri, ContentValues values, String where, String[]
whereArgs) {
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case WORDS:
            count = db.update(USERDICT_TABLE_NAME, values, where,
whereArgs);
            break;

        case WORD_ID:
            String wordId = uri.getPathSegments().get(1);
            count = db.update(USERDICT_TABLE_NAME, values, Words._ID + "="
+ wordId
                    + (!TextUtils.isEmpty(where) ? " AND (" + where + ')'
: ""), whereArgs);
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }

    // Only the enabled IMEs and spell checkers can access this provider.
    if (!canCallerAccessUserDictionary()) {
        return 0;
    }

    getContext().getContentResolver().notifyChange(uri, null);
    mBackupManager.dataChanged();
    return count;
}
```

Finally, notice how the `AndroidManifest.xml` file does not provide any additional protection (e.g. intent filters or permissions) to the explicitly exported content provider:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.providers.userdictionary"
        android:sharedUserId="android.uid.shared">

    <application android:process="android.process.acore"
        android:label="@string/app_label"
        android:allowClearUserData="false"
```

```
        android:backupAgent="DictionaryBackupAgent"
        android:killAfterRestore="false"
        android:usesCleartextTraffic="false"
        >
        <provider android:name="UserDictionaryProvider"
                android:authorities="user_dictionary"
                android:syncable="false"
                android:multiprocess="false"
                android:exported="true" />
    </application>
</manifest>
```

It is trivial for an attacker to update the contents of the user dictionary by invoking code like the following from any malicious application, without the need to ask for permission:

```
ContentValues values = new ContentValues();
values.put(UserDictionary.Words.WORD, "IOActive");

getContentResolver().update(UserDictionary.Words.CONTENT_URI, values,
        null, null);
```

It would be also trivial to delete any content, including the entire personal dictionary:

```
getContentResolver().delete(UserDictionary.Words.CONTENT_URI, null, null);
```

Both methods (`update` and `delete`) are supposed to return the number of affected rows, but, in this case (for non-legitimate invocations), they will always return zero, making it slightly more difficult for an attacker to extract information from the content provider.

Even if the `query` function is not directly affected by this vulnerability, it is still possible to dump the entire contents by exploiting a time-based, side-channel attack. The `where` argument is fully controllable by the attacker, and **a successful update of any row takes more time to execute** than the same statement when it does not affect any row.

Proof of Concept

Consider the following code fragment running locally from a malicious application:

```
ContentValues values = new ContentValues();
values.put(UserDictionary.Words._ID, 1);

long t0 = System.nanoTime();
for (int i=0; i<200; i++) {
    getContentResolver().update(UserDictionary.Words.CONTENT_URI, values,
                    "_id = 1 AND word LIKE 'a%'", null);
}
long t1 = System.nanoTime();
```

Invoking the very same statement enough times (e.g. 200 times, depending on the device), the **time difference** (`t1-t0`) between an SQL condition that evaluates to "`true`" and the ones that evaluate to "`false`" will be noticeable, allowing the attacker to extract all the

information in the affected database by exploiting a classic time-based, Boolean blind SQL injection attack.

Therefore, if the first user-defined word in the dictionary starts with the letter “a”, the code fragment will take more time to execute (e.g. 5 seconds), compared to the lesser time required when the guess is false (e.g. 2 seconds), since no row will be actually updated in that case. If the guess is wrong, we can try with “b”, “c”, and so on. If the guess is correct, that means that we know the first character of the word, so we can proceed with the second character using the same technique, and so on.

Notice how the database contents will not actually be altered, even when the condition is true (it will update the “`_id`” value to its same original value), but other columns (such as “`frequency`” or “`locale`”) can also be arbitrarily smashed with any constant value if the attacker does not care about their contents, leading to the same results.

This process could be fully automated, and optimized with more advanced techniques, all trivial to implement, such as calibrating the number of repetitions, applying the binary search algorithm, injecting sub-selects to determine the first identifier (e.g. “`SELECT MIN(_id) FROM words`”), etc., reducing the overall effort required in runtime to dump all the personal dictionary contents.

Fixes

One of the simplest fixes would be to call the “`canCallerAccessUserDictionary`” function and check if the caller is allowed to invoke the method before doing anything else.

A standard patch file is provided below as a reference:

```
--- a/src/com/android/providers/userdictionary/UserDictionaryProvider.java
+++ b/src/com/android/providers/userdictionary/UserDictionaryProvider.java
@@ -252,6 +252,11 @@

     @Override
     public int delete(Uri uri, String where, String[] whereArgs) {
+        // Only the enabled IMEs and spell checkers can access this
provider.
+        if (!canCallerAccessUserDictionary()) {
+            return 0;
+        }
+
         SQLiteDatabase db = mOpenHelper.getWritableDatabase();
         int count;
         switch (sUriMatcher.match(uri)) {
@@ -269,11 +274,6 @@
                 throw new IllegalArgumentException("Unknown URI " + uri);
         }

-        // Only the enabled IMEs and spell checkers can access this
provider.
-        if (!canCallerAccessUserDictionary()) {
-            return 0;
-        }
-
```



```
         getContext().getContentResolver().notifyChange(uri, null);
         mBackupManager.dataChanged();
         return count;
@@ -281,6 +281,11 @@

     @Override
     public int update(Uri uri, ContentValues values, String where,
String[] whereArgs) {
+        // Only the enabled IMEs and spell checkers can access this
provider.
+        if (!canCallerAccessUserDictionary()) {
+            return 0;
+        }
+
         SQLiteDatabase db = mOpenHelper.getWritableDatabase();
         int count;
         switch (sUriMatcher.match(uri)) {
@@ -298,11 +303,6 @@
                 throw new IllegalArgumentException("Unknown URI " + uri);
         }

-        // Only the enabled IMEs and spell checkers can access this
provider.
-        if (!canCallerAccessUserDictionary()) {
-            return 0;
-        }
-
         getContext().getContentResolver().notifyChange(uri, null);
         mBackupManager.dataChanged();
         return count;
```

Mitigation

The vulnerability has been fixed in the official repository. Specifically, in the following commit:

https://android.googlesource.com/platform/packages/providers/UserDictionaryProvider/+/ccf7d5c98fc81ff4483f921fb4ebfa974add9c6

Several vendors integrating Android had released security patches for this vulnerability in June 2018. IOActive recommends applying the latest security patches from your vendor. If for any reason it is not possible to apply such updates, consider reviewing the contents of your personal dictionary and make sure it does not contain any sensitive information in the unlikely event the issue becomes actively exploited.

Timeline

2018-03-13	IOActive discovers vulnerability
2018-03-14	IOActive reports vulnerability to Google
2018-06-07	Google fixes the vulnerability
2018-08-02	IOActive advisory published

