A Pattern for Remote Code Execution using Arbitrary File Writes and MultiDex Applications

Published: 2017-06-15 · Archived: 2026-04-05 23:21:23 UTC

Summary

The following blog explains vulnerabilities that allow attackers to execute code remotely on a Android userUs device through applications which contain both a arbitrary file write and use multiple dex files. *This remote exploitation can occur without the userUs knowledge.*

In this post, I will walk you through the issues and the process of developing a remote code execution exploit for the Talking Tom application. This is a pattern which can be applied to many other applications. If you have any questions, feel free to contact me via Twitter at @fuzion24.

A video of the vulnerability in action can be seen below:

https://youtube.com/watch?v=u9XqWuY0WG8

Vulnerabilities in External Libraries

External, third-party libraries are often used by developers and are included in hundreds of thousands of applications. These libraries are treated like a black box. Developers choosing to implement external libraries in their applications must recognize that these libraries frequently contain vulnerabilities that weaken the overall security of applications, with advertisement libraries being a particular target of interest. If and when vulnerabilities are identified in external libraries, the vulnerabilities must be patched, and the developer must then rebuild the application and redistribute it with a fixed library, as currently there is no operating system supported library sharing mechanism. This leads to applications with long-standing or, at times, permanent vulnerabilities, even if patches to the external library are released.

The Vungle advertisement library contains an arbitrary write vulnerability which affects thousands of applications. While is not uncommon for games and other types of applications to download additional resources in plaintext via a zipfile, this library provides an easy target as it is widely used and the attack vector is identical across all affected applications. We responsibly disclosed this vulnerability to Vungle prior to notifying all of the affected developers, and then directly notified developers of the affected applications.

We determined that applications using the Vungle library and containing both a remote arbitrary file write and using multiple dex files are remotely exploitable. Attackers can modify network traffic to gain code execution on a userUs phone. This code execution will be restricted to the sandbox of the application. However, since OEMs and Google are relatively inadequate at releasing device patches to known privilege escalation vulnerabilities, it becomes trivial for an attacker to escape the application sandbox once code execution has been gained on a vast majority of devices. This type of vulnerability is particularly unfortunate because developers typically have very

little visibility into the internal workings of 3rd party libraries. This is due to the fact that they are usually distributed as blackbox binaries. The developers, therefore, can only be at fault for blindly including libraries of which they have no visibility.

Vungle Arbitrary Write Vulnerability

The Vungle advertisement library is distributed as a .jar which developers can include into their application. When a developer utilizes this SDK, their application becomes vulnerable to a remote arbitrary file write vulnerability. The following is a brief synopsis of the vulnerability (assigned CVE-2014-9333):

We can set up a man in the middle attack against the phone to proxy the deviceUs network traffic and see what is being sent by the application. The first request we see from the library is requesting instructions from the server on what to display:

```
   POST http://api.vungle.com/api/v1/requestAd               _ 200 application/json 967B 947.36kB/s
```

In the JSON response, we can see the server tell the app where to download additional zip resources which contain the video advertisements that are displayed throughout the application.

```
{        ...           "postBundle": "http://cds.g8j8b9g6.hwcdn.net/bundles/526956a8584cbfa904000010-4.zip"
```

Shortly thereafter, we see this zip being downloaded:

```
   GET http://cds.g8j8b9g6.hwcdn.net/bundles/526956a8584cbfa904000010-4.zip              _ 200 application/zi
```

This ZIP file is downloaded in plaintext. There are no further mitigations to prevent tampering of the file. The Android ZIP APIs do not prevent directory traversals by default, allowing for a file with a directory traversal in the name to be injected on-the-fly into the ZIP. This allows us to gain an arbitrary write in the context of the app. With this, we can easily write a script for a proxy that will inject our payload on-the-fly into the zip. LetUs test this out by downloading the payload using our zip injecting proxy:

```
$ curl -x http://localhost:9999 http://cds.g8j8b9g6.hwcdn.net/bundles/51508704e2903eb17f000006-2.zip > tmp.zip
```

Now, letUs look at the files in the zip we downloaded through our proxy:

```
$ unzip -l tmp.zip        Archive:  tmp.zip          Length     Date   Time    Name         --------    ----
```

Notice here that the zip was injected with a directory traversal that writes inside of the app directory. During the attack, we can see that our files were written in the applicationUs data directory. This directory is only writable by the application that owns it:

```
root@hammerhead:/data/data/com.outfit7.mytalkingtomfree/code_cache/secondary-dexes # ls -l | grep i_wrote
```

Our dummy payload was successfully extracted by Vungle. Now, we need to turn this file write into something more useful.

Turning our Arbitrary File Write into Code Execution

Although the arbitrary file write is interesting, it only allows for us to be somewhat destructive by filling up the disk and overwriting files. LetUs now look for a target that the application has write access to, which we can use to gain code execution.

The executable code of an Android application is stored inside the .APK (just a zip file) in a single file named classes.dex where .dex stands for Dalvik Executable. When Android applications are installed, the classes.dex file is extracted from the APK and run through the the Dalvik optimizer. The optimized dex file, which has the file extension `.odex`, is then stored in `/data/dalvik-cache/`. Here is the optimized dex file for Google+:

```
root@flo:/data/dalvik-cache # ls -l data@app@com.google.android.apps.plus-1.apk@classes.dex       -rw-r--r-- s
```

Notice here, that even Google+ itself doesnUt have write access to its own .odex file. It canUt tamper or rewrite the original classes.dex file as it does not write to the filesystem where it can be modified by the application. Additionally, apps cannot modify their own Android package file (.apk) after installation. Taking a look at an .odex file, we notice that itUs owned by system:all_a65, but only system has write access to this file. In this way, an application cannot modify its own Dalvik bytecode. In order for an application to dynamically extend itself, it needs to utilize the DexClassLoader APIs.

The virtual machine on which Android applications execute, the Dalvik Virtual Machine, was designed for high efficiency and low power devices. The executable code for the VM is typically stored in a single file named classes.dex. In contrast, Java bytecode is spread across many files with each file containing the bytecode for one class.

The compactness of the DEX format allows for deduplication of strings and other constants. It also allows the file to be memory mapped rather than lazy loaded from disk. The Dalvik executable format (.dex) has well documented limitations which has some horrible workarounds that are exacerbated by bloated libraries. There are, however, solutions for this problem.

The Dalvik Exchange tool, which converts Java bytecode in the form of .class files into DEX format, has recently added support for automatically splitting applications that exceed the method limit into multiple .dex files. Until the latest Android support library revision, rev v21, applications had to manually manage the extra dalvik executables and dexload them when necessary.

Taking a look at the files in the APK (the file format for Android applications; a zip file), we can see there are two dalvik executable files `classes.dex` and `classes2.dex`.

```
_  ~ unzip -l com.outfit7.mytalkingtomfree-1.apk | grep classes          5286068  10-20-14 10:46   classes2.dex
```

Android itself prior to Android 5.0 has no awareness of any Dalvik executable file except for classes.dex. The multi-dex support library will need to load any secondary dex files (classesX.dex) dynamically. If we take a look at the dex loading API in Android we see:

```
public DexClassLoader (String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent)
```

This means that we can only load a DexFile from disk and, therefore, the secondary dex loader will need to write our executable code to a disk where the app has write capabilities:

```
root@flo:/data/data/com.outfit7.mytalkingtomfree/code_cache/secondary-dexes # ls -l        -rw-r--r-- u0_a285
```

Poor kitty is about to get owned and he doesn't even know it:

Let’s validate that this is the exact same dex file as we saw earlier in the APK. We check the SHA1 of the classes2.dex in the APK:

```
_  /tmp  unzip com.outfit7.mytalkingtomfree-1.apk classes2.dex       Archive:  com.outfit7.mytalkingtomfree-1
```

Let’s pull the classes2.dex from the device and check its SHA1:

```
_  /tmp  adb shell su -c "cp /data/data/com.outfit7.mytalkingtomfree/code_cache/secondary-dexes/com.outfit7.myt
```

Great. They match just as we thought.

This zip file contains the additional bytecode (the secondary dex) which is writeable by the application. Since the Dalvik bytecode is writable from application context, we have a clear path to turn a write vulnerability into code execution. Let’s get to work using our arbitrary file write via the Vungle library to overwrite the secondary dexfile.

We now have our arbitrary write primitive and we have a way to turn that into code execution. Let’s start crafting our payload. We will need to create a zip file with a classes.dex in it, which will get loaded by our target app and execute our payload. I took a very naive approach to solve this problem: I created a .dex file with a HelloWorld class and injected it into the zip.

After injecting the .dex file I created into the zip, we see that the VM tried to load my .odex and realized it was stale because I had TupdatedU the Dalvik bytecode.

```
I/dalvikvm(25418): DexOpt: source file mod time mismatch (455455dc vs 455c6d35)       D/dalvikvm(25418): ODEX
```

The DEX file the app is loading has none of the expected classes, so it crashes with a NoClassDefFoundError exception.

```
W/System.err(25418): java.lang.NoClassDefFoundError: com.outfit7.talkingfriends.clips.c       W/System.err(254
```

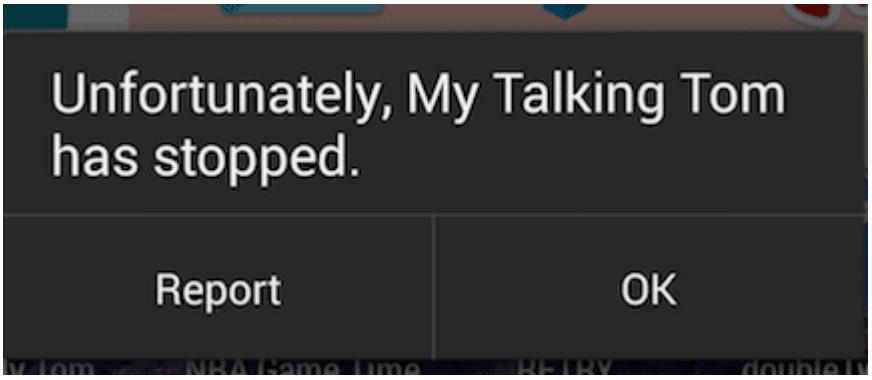
No problem! Let’s just add com.outfit7.talkingfriends.clips.c class to appease the app and toss our payload in there:

```
  package com.outfit7.talkingfriends.clips;         import android.util.Log;                public class c {
```

After closing the app and reopening, our dex gets optimized and loaded. Since our payload is simply logging a statement to logcat to prove code execution, we can check logcat:

```
_  ~  adb logcat | grep WINN       D/WINNER  ( 7490): Game Over
```

This logging statement can easily be replaced with any malicious payload.

The app crashes after this which would take down our child process. In order to make a production grade exploit, we would need to create a zombie process and replace our payload with the original so the app continues to function as normal. 

A more sophisticated exploit would be to inject the payload directly into the original dex file, by reverse engineering and patching the dalvik bytecode or by having the payload replace the original dexfile after execution.

Mitigations and Closing Thoughts

Plaintext traffic

All traffic between clients and servers should encrypted and authenticated. TLS (Transport Layer Security, previously SSL) should be used for all communications between the client and the server including for file downloads such as ZIP files. Consider the recent example of Verizon injecting unique indentifiers into traffic. Attacks like this can be mitigated best by employing TLS. Using TLS to secure all communications would require attackers to obtain control over a Certificate Authority, making it that much harder to exploit these applications.

Validation of Zip files

The default behavior of Android's Zip library is dangerous. In order for the zip to write files outside of the target extraction directory an additional flag should be provided. Let's consider how GNU unzip handles a zip file with a directory traversal in the filename:

Here is an example of a zip file that includes a relative directory traversal:

```
_  ~  unzip -l evil.zip       Archive:  evil.zip         Length      Date    Time    Name       ---------  -
```

When we try to extract this zip file with GNU unzip, it strips off any dangerous components and extracts inside of the current working directory:

```
_  ~  unzip evil.zip       Archive:  evil.zip       warning:  skipped "../" path component(s) in ../../../../
```

There exists a flag to allow GNU unzip to extract files outside of the active extraction folder tree head.

```
_  ~  unzip -: evil.zip       Archive:  evil.zip        extracting: ../../../../../../../../some/random/path/
```

From `man unzip` we can learn more about this flag:

```
        -:     [all but Acorn, VM/CMS, MVS, Tandem] allows to extract archive  members                    into
```

Android therefore, needs to take care to ensure sane defaults. If a developer needs to extract files outside of the current working directory, this should be an option, but its manual specification should not be required.

Developer mistakes are inevitable. The easiest, most effective way to prevent this and similar type of attacks is to ensure that all network traffic is properly secured via TLS. This is important enough that it bears repeating: All network traffic should be encrypted!

Source: https://www.nowsecure.com/blog/2015/06/15/a-pattern-for-remote-code-execution-using-arbitrary-file-writes-and-multidex-applications/