

Project 4: Android Application Privacy

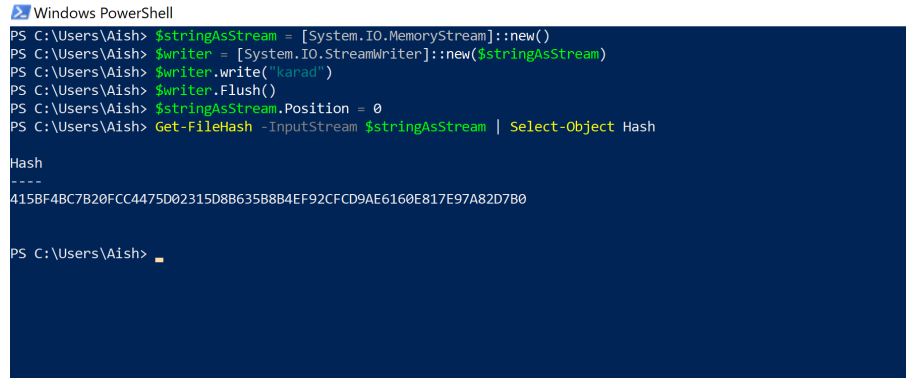
Aishwarya Karad

April 2020

1 Question 1

1.1 Download and set-up

I downloaded the "mp4-appset-4" based on the first character of my SHA256 hex code of my last name "karad".



```
Windows PowerShell
PS C:\Users\Aish> $stringAsStream = [System.IO.MemoryStream]::new()
PS C:\Users\Aish> $writer = [System.IO.StreamWriter]::new($stringAsStream)
PS C:\Users\Aish> $writer.write('karad')
PS C:\Users\Aish> $writer.Flush()
PS C:\Users\Aish> $stringAsStream.Position = 0
PS C:\Users\Aish> Get-FileHash -InputStream $stringAsStream | Select-Object Hash

Hash
----
415BF4BC7B20FCC4475D02315D8B635B8B4EF92CFCD9AE6160E817E97A82D7B0

PS C:\Users\Aish>
```

Figure 1: commands to generate the hash of "karad"

Setting up VM:

- I used GCP to launch an Ubuntu 16 VM and I launched one with 4 vC-PUs and 32 GB RAM.
- I uploaded the argus-saf-version-assembly.jar file and the mp4-appset-4 dataset.

1.2 Running Argus-SAF:

- I ran the following script in the VM to run Amandroid on all the apks overnight(this whole process took well over 24 hours).

Listing 1: bash version

```
#!/bin/bash
num=1
for i in mp4-appset-4/*
do
echo "APK number $num";
java -Xmx24g -jar argus-saf_2.12-3.2.0-assembly.jar -mo
DATA_LEAKAGE -a COMPONENT_BASED t -o ./output/$num ./ $i ;
((num=num +1))
done
```

- After processing, all the output of Amandroid was put it in the **output** folder.
- I used the following command to find all the apks with at least one taint path. There was a total of **16 apks** that had a tainted path.

```
find ./ -name AppData.txt | xargs grep -i taintpath: | sort --unique
```

- I have from this point onwards referred to the APKs as their sequence in the mp4-appset-4 folder.

1	br.com.webadger.intoyou-92.apk
2	breastenlarger.bodyeditor.photoeditor-23.apk
3	com.flightaware.android.liveFlightTracker-1053.apk
.	.
.	.
.	.
.	.
.	.
24	spapps.com.earthquake-20.apk
25	tunein.player-260715.apk

1.3 Statistics

Counts of Sources and Sinks:

- I collected the total number of sources and sinks per APK and found out how many were ICC(Inter Component Communication) sources and sinks.
- I thought the collection of this data might be useful to me because I noticed that in all the TaintPaths all the sources and sinks were API sources and sinks which made me think that either ICC sources/sinks were generally not used to leak information or that these breaches were undetected by Amandroid.

- I looked online and found a [paper](#) that talked about how icc is used to override "android.permission.INSTALL_PACKAGES" and install other packages without user permission. The author of the paper also mentioned that these attacks were hard to detect.
- I collected the count by using the following commands:

```
grep -c "TaintPath:" AppData.txt
grep -c "<Descriptors: api_sink" AppData.txt
grep -c "<Descriptors: api_source" AppData.txt
grep -c "<Descriptors: icc_source" AppData.txt
```

APK	Number of Sources	Number of Sinks	Taint Paths found
1	32	95	4
2	43	221	21
3	51	723	92
4	-	-	-
5	69	777	7
6	70	256	16
7	27	109	114
8	18	110	12
9	6	45	-
10	70	284	136
11	29	342	-
12	17	100	38
13	22	323	-
14	-	-	-
15	42	94	-
16	-	-	-
17	3	33	-
18	72	274	94
19	72	104	10
20	64	21	-
21	5	107	12
22	86	388	44
23	24	2	3
24	65	830	96
25	42	240	16

-	ICC	API
sources	648	260
sinks	165	5314

- I also plotted the results of the taint analysis for each APK.

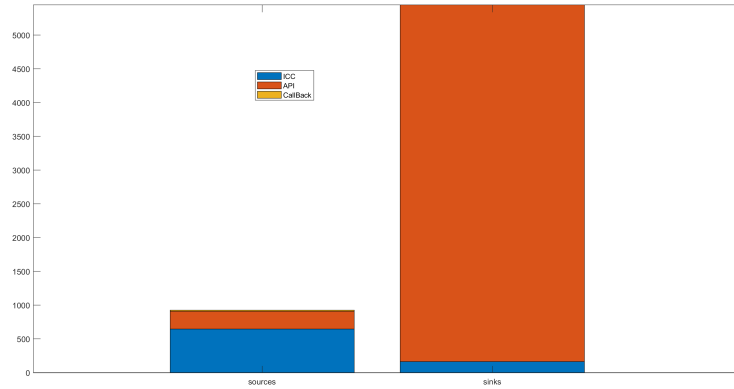


Figure 2: ICC and API sources/sinks.

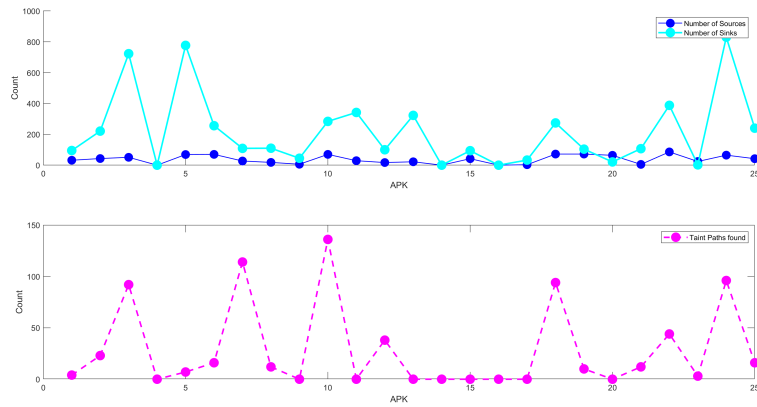


Figure 3: Taint Analysis Statistics

Flow Analysis:

- I then began to find some common Source/sink pairs and tried to understand how the information flowed.

Landroid/content/pm/PackageManager: PackageManager

Landroid/content/SharedPreferences: SP

Landroid/app/Activity: Activity

Landroid/util/Log:log

java/io:IO

java/net/URLConnection: URL (network source/sink)

org/apache/http/HttpResponse: Apache (network sink/source)

In the table below network sinks means Apache and URL.

Source → Sink	Count	APKs
PackageManager → log	173	3,18,21,24
PackageManager → SP	153	3,7,1821,23,10
PackageManager → Network Sinks	11	10,24
Activity → Network Sinks	12	6
Network Sources → IO	9	2,5,7,12,22
URL → log	33	2,8,12,24,25

I didn't find any APKs with Location parameters in the Taint path but I did find several apps that had Locations in the their list of sources (APKS 24,12,25) whereas many apps had set the user permission location access(android.permission.ACCESS_FINE_LOCATION and android.permission.ACCESS_COARSE_LOCATION) (APKS 6,10,12,22,19,10,24,25)

2 Question 2

Overview: I found four APKs with network sinks in the taintPath. A network sink doesn't necessarily mean that the APK is leaking information because it could some data require for services and sent with the permission of the user. Here are some of the things that i tried to notice for each apk. I used the JadX decompiler to decompile the apk into readable java code.

2.1 APK #25:

Taint Paths with Network Sinks:4

These were the source/sinks for the 4 taint paths In the decompiler you can see the various packages and classes and track the flow by first tracking the source and then finding where it goes. I found the "peekService()" function in the path as mentioned in the sink information.

path was: /tunein/services/CastServiceBroadcastReceiver. The sink was quite hard to find exactly because there was so many(119 search results to be precise on JadX) java.net.URLConnections and it was getting quite tedious to parse through all that code. Based on my limited knowledge in this domain and not being able to exactly trace the path from source to sink I'm not a 100% positive but I think that no this app is not violating some privacy rules but I can't say exactly how. The peekservice function has Context and Intent parameters which are being returned from this function and as long as context and Intent doesn't have any private information this should be safe. Also, this function is listed in the CastServiceBroadcastReceiver service which probably means that this is the app performing its assigned tasks.

The other source sink pair was collecting some input via `net/URLConnection` `getInputStream()` and it ended up in a network sink. I don't think any sensitive information was being leaked here but maybe someone who has more experience will understand it better.

```
Source: <Descriptors: api_source: Ljava/net/URLConnection;.getInputStream:()
Ljava/io/InputStream;>
```

```
Sink: <Descriptors: api_sink: Ljava/net/URL;.openConnection:
()Ljava/net/URLConnection; 0>
```

```
Source: <Descriptors: callback_source: Ltunein/services/
CastServiceBroadcastReceiver;
.peekService:(Landroid/content/Context;
Landroid/content/Intent;)Landroid/os/IBinder;>
```

```
Sink: <Descriptors: api_sink: Ljava/net/URL;.openConnection:
()Ljava/net/URLConnection; 0>
```

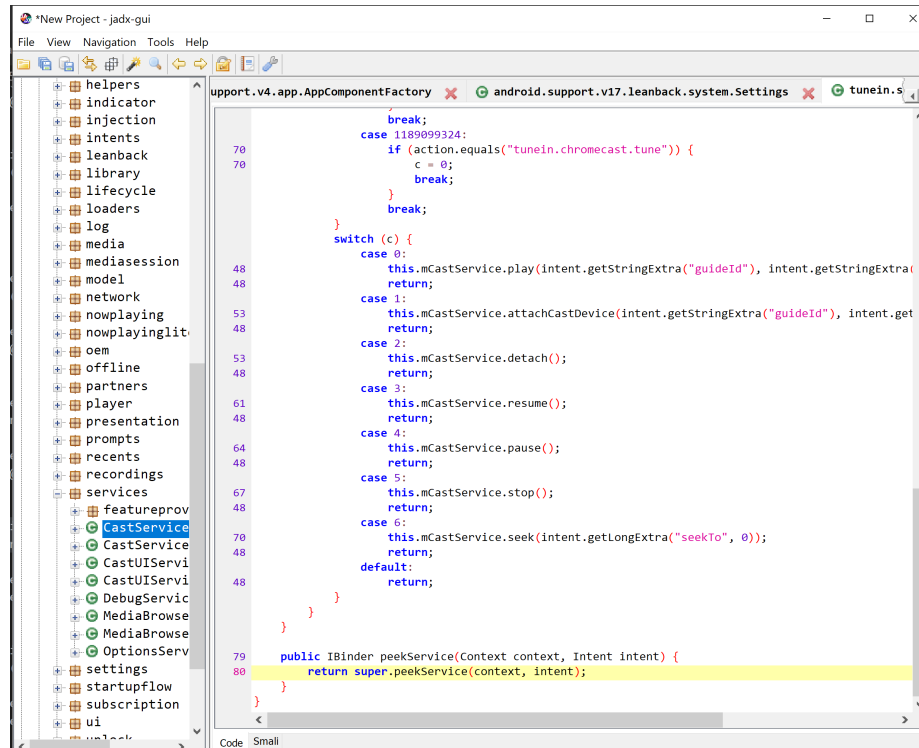


Figure 4: Tracking down a taint source in the decompiled code

2.2 APK # 6:

of Taint Paths with Network Sinks: 12

I followed a similar method to track down the taint path as much as I could. This is a flight tracking app, here the flow from source to sink could mean a user searching for a particular flight and this search field is passed to the network sink. I looked at the path given and from what I understood the `ConfigureAircraftWidgetActivity` looked like it was just try to retrieve all the flight related information and displaying it to the user. "`Lcom/flightaware/android/live-FlightTracker/activities/ConfigureAircraftWidgetActivity`" there were 12 such paths and I doubt if this was some information leakage activity being performed then there wouldn't be 12 seemingly identical paths

```
Source: <Descriptors: api_source: Landroid/app/Activity;.findViewById:
(I)Landroid/view/View;>
      Sink: <Descriptors: api_sink: Ljava/net/URL;.openConnection:()
      Ljava/net/URLConnection; 0>
```

2.3 APK #10:

Taint Paths with Network Sinks:9 All the taint paths given were from `PackageManager` to Network sink. I couldn't really completely trace the path. Whatever I did trace didn't seem like it was leaking any information

```
Source: <Descriptors: api_source: Landroid/content/pm/PackageManager;
.getInstalledApplications:(I)Ljava/util/List;>
      Sink: <Descriptors: api_sink: Ljava/net/URL;.openConnection:
      ()Ljava/net/URLConnection; 0>
```

2.4 APK #24:

Taint Paths with Network Sinks:2 Here sending the location is not a breach of trust because they're asking for permissions. I also read through the `MainActivity` class to look for variable with any sensitive information like phone number, address, etc. but didn't find anything. All these apps had `CookieBuilders` and `Cookies` and when I looked them up on the app store I found multiple reviews saying this app was not very good and too many ads. I also tried to follow the taint path again and searched for the source "`Source: ;Descriptors: api_source: Landroid/content/pm/PackageManager;.queryIntentServices: (Landroid/content/Intent;I)Ljava/util/List;`" There were multiple results for this and while it mostly looked like it was just trying to perform it's functions I am not sure if there was any information leakage because none of the information collected here looked sensitive.(figure 5)

```

static Intent getIntent(Context context) {
    PackageManager packageManager = context.getPackageManager();
    Intent intent = new Intent().setAction("com.onesignal.NotificationExtender").setPackage(context.getPackageName());
    List<ResolveInfo> queryIntentServices = packageManager.queryIntentServices(intent, 128);
    if (queryIntentServices.size() < 1) {
        return null;
    }
    intent.setComponent(new ComponentName(context, queryIntentServices.get(0).serviceInfo.name));
    return intent;
}

```

Figure 5: Tracking down source in the code

2.5 APK #1:

No network sinks. There were 4 taint paths discovered by Argus-SAF in this and all of these paths ended with log files as sinks, so I think that as long as the log files are safe the app isn't leaking any information through the network but if the log files are sent across the network and if they don't contain any sensitive information then the app is still secure. I tried to follow the variables through the decompiled code but couldn't find any such breach of the logs generated. I read through the Logger class and didn't find any breaches.