DOI: 10.15439/2018F11
ISSN 2300-5963 ACSIS, Vol. 15

Volatile memory-centric investigation of SMS-hijacked phones: a Pushbullet case study

Mark Vella
Department of Computer Science
University of Malta
Msida, Malta
Email: mark.vella@um.edu.mt

Vishwas Rudramurthy
Department Of Computer Science and Engineering
Channabasaveshwara Institute Of Technology
Gubbi, Tumkur, India
Email: vishwas.rudramurthy@cittumkur.org

***Abstract*—Cloak-and-Dagger attacks targeting Android devices can completely hijack the UI feedback loop, with one possible consequence being that of hijacking SMS functionality for cybercrime purposes. What is of particular concern is that attackers can decouple stealth activities from SMS hijacking. Consequently the latter could be pulled off using completely legitimate apps that normally would allow users to manage text messages from their personal computers (SMSonPC), but this time all hidden away under attacker control. This work proposes a digital investigation process aiming to uncover SMS-hijacked devices. It uses bytecode instrumentation in order to force the dumping of volatile memory areas where evidence for the hijack can be located. Eventually both the malware that conceals the SMS-hijacking and the compromised or smuggled SMSonPC app can be identified. Preliminary results are presented using a case study based on the popular SMSonPC app: Pushbullet.**

I. Introduction

THE Cloak-and-Dagger set of attacks demonstrates how through the abuse of two permissions, Android malware can take control of the entire User Interface (UI) feedback loop [1]. Essentially what this means is that through malicious crafting an attacker can snoop on or even take full control over a user's intentions when interacting with a smart-phone using touch screen taps and swipes, and conversely of all the device's reactions to them. The consequence is a Man-in-the-Middle (MiTM) posture for an attacker sitting in between users and their devices. This is a critical game changer in the sense that up till this point it was generally thought that UI attacks were more about forcing users to send clicks to marketing referral web-sites, rather than completely hijacking a device. This role up till this point was reserved to rooting/jail-braking malware that takes advantage of memory corruption errors inside firmware. The two abused permissions relate to accessibility (a11y) and overlay drawing (draw-on-top) functionality. The former permits an app to access the UI widgets of a second app, whilst the latter permits an app to draw overlays on top of on another app's UI. Their combined abuse can be disastrous due to the long-term stealth an attacker can attain.

The threat that we are concerned with in this work leverages these two permissions to silently install or compromise one of those apps that let users send/read SMS text messages from their personal computers (PCs). These apps are gaining popularity since in the larger context they let users manage all of their smart devices (phones, tablets, wearables and what not) from a single machine[1]. In the specific case of text messages, typing them on a PC keyboard is of particular convenience whenever possible, and for the rest of this paper we will refer to apps that offer this functionality as *SMSonPC*.

Once Cloak-and-Dagger malware tricks victims into giving up or stealing their login SMSonPC credentials, it moves on to activate a11y and conceal SMS-related activity by abusing draw-on-top permissions. At this point the SMSonPC app provides an attack vector to hijack the device's SMS functionality in a highly stealthy manner. What is of major concern is that the SMSonPC app in question is totally legitimate, possibly installed by the user in the first place. Furthermore, the use of draw-on-top and a11y features have been picked up by popular apps and at this point it can be very difficult to make amends from Android's end.

While Google Play's screening has been tightened accordingly, it is a well known fact that persistent attackers tend to succeed in eventually having their malware included in this trusted app store. Android Oreo also includes tightened security, yet its fragmented adoption is still expected to stand in the way[2][3]. Moreover, mitigations only address overlay drawing and which could potentially be replaced by social engineering tricks nonetheless. Further details with respect to the hijacking procedure and existing digital investigation options are provided in section II.

The idea behind the proposed digital investigation process (section III) is that in the event of a suspected SMS-hijacking, or else on a routine basis, users will be able to investigate their devices for possible infection. This approach aims directly at the core of the issue: long-term stealth. During a first stage those apps that look suspicious, either because of the aforementioned requested permissions or else due to SMS functionality, are extracted from the device in order to have their bytecode instrumented. The injected bytecode forces the dumping of those volatile memory areas where evidence uncovering the hijack could be located, without necessarily requiring device rooting. During a second stage

[1]https://www.androidauthority.com/apps-send-text-sms-pc-ways-740669/
[2]https://www.wired.com/story/cloak-and-Dagger-android-malware/
[3]https://developer.android.com/about/versions/oreo/android-8.0-changes.html#all-aw

of the investigation forensic analysis is conducted upon the collected memory dumps. They are combined with the context provided by text messages from flash/SIM memory along with any suspicious destination numbers as obtained from operator billing logs. In the interim, the device is used normally except for the additional recording of potential artifacts that can uncover both the malware that sets up and conceals the SMS-hijack, as well as the compromised/smuggled SMSonPC app. Basically any text message flows inferred to originate/end from/at SMSonPC apps without the device's owner consent, and in the presence of a draw-on-top/a11y app, indicate an ongoing SMS-hijack.

A case study using Pushbullet, a popular SMSonPC app, is used for initial exploration of this technique in terms of its effectiveness and practicality (section IV). The proposed SMS-hijack investigation process along with the preliminary results from this case study are the primary contributions of this work.

II. SMSonPC hijacking

The essential ingredients for the stealthy SMS-hijack being considered in our threat model consist of an SMSonPC app combined with a number of Cloak-and-Dagger attack techniques. In this work we focus on the abuse of Pushbullet to serve this purpose.

A. The Pushbullet SMS-hijack scenario

In order to make use of Pushbullet users need to install a controlling application on their PC in the form of a stand-alone native application or a browser extension. Otherwise they may simply log into a web interface[4]. Whichever client option, a device is instructed to send a text message by means of what is called an ephemeral message, which is possibly encrypted, and an example of which is shown in Listing 1. This is a JSON-formatted object which is sent to the Pushbullet server by the controlling application. In this case the instruction is to send a `Hello!` text message to `+1 303 555 1212` on behalf of user-id `ujpah72o0` through her Pushbullet-registered device with identification `ujpah72o0sjAoRtnM0jc`. This is an example of a Pushbullet `push` event, intended for dispatch to the identified Android device, and which accesses its SMS services as specified by the `messaging_extension_reply` type using the `com.pushbullet.android` package.

Listing 1
A Pushbullet ephemeral message instructing a phone to send an SMS text message.

```
{
  ‘‘push”: {
    ‘‘conversation_iden”: ‘‘+1 303 555 1212”,
    ‘‘message”: ‘‘Hello!”,
    ‘‘package_name”: ‘‘com.pushbullet.android”,
    ‘‘source_user_iden”: ‘‘ujpah72o0”,
    ‘‘target_device_iden”: ‘‘ujpah72o0sjAoRtnM0jc”,
    ‘‘type”: ‘‘messaging_extension_reply”
  },
  ‘‘type”: ‘‘push”
 }
```

[4]https://www.pushbullet.com

Cloak-and-Dagger is really a collection of attacks [1] that abuse Android draw(ing)-on-top of opaque or transparent overlays and a11y system services. They require the `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions respectively. Since attacks #3 and #4 (as identified in [1]) are able to hijack the device's virtual keyboard, they could be used to steal Pushbullet credentials during installation/configuration. The prior attack succeeds by placing multiple transparent "pass-through-clicks" overlays per keyboard button and then snoops on keystrokes by having all the overlays capture clicks outside their region. Subsequently it identifies the tapped button using a clever Z-order trick. The latter attack abuses accessibility services by listening to keyboard button click notifications. Attack #5 provides an alternate hijacking strategy and combines the two permissions. It exploits accessibility services to detect that the user has navigated to the Pushbullet app, and then proceeds to exploit draw-on-top by displaying a fake but authentic-looking Pushbullet log-in screen. At that instance it lures users to send their credentials directly to the attacker.

Through Cloak-and-Dagger an attacker can even move on from compromising a user-installed Pushbullet installation to the silent installation of a covert one. Specifically through attack #8, using the standard Android API an attacker can initiate an installation of Pushbullet as well as programmatically confirming the same action when prompted, all the while covering this activity through a draw-on-top overlay. Subsequently, through a11y services, the malware can proceed to cover its tracks by accessing the "recent windows" view and dismissing all of its content. The final step is to launch attack #9, i.e. navigating to app settings and outright enabling all the permissions required by Pushbullet. Consequently the user won't get prompted to grant permissions when subsequently Pushbullet is launched remotely by an attacker to disclose or send SMSes on the device's owner behalf, and thereby maintaining stealth. It is noteworthy that both draw-on-top and a11y features come along with mechanisms to protect from abuse, yet the Cloak-and-Dagger attacks don't simply bypass these protections but also go as far as abusing them. For example the aforementioned Z-order trick exploits the same security flag that informs a clicked widget about whether the click passed through an overlay drawn on top of it.

Having obtained access to draw-on-top and a11y permissions through deceit, along with a compromised or smuggled SMSonPC app through Cloak-and-Dagger, an attacker can now proceed with mischief. For example, the device can be turned into a crime text messaging proxy or even into a spying device by leaking message content. Maintaining stealth in the former case can be achieved by deleting all sent messages, once again possibly through Cloak-and-Dagger means. In the latter case it is a question of whether the SMSonPC has been smuggled or compromised. In the first case, it is simply a question of keeping the SMSonPC app installation concealed from the device owner, while the second case also requires that attackers hide their tracks within the SMSonPC controlling app. Whatever the scenario the end result is that of a stealthy SMS-hijack.

B. Android SMSonPC apps

Any SMSonPC app requires the `SEND_SMS` and `READ_SMS` permissions in order to be able to interact with the device's SMS features. The `INTERNET` permission is also required to provide a communication link with the SMSonPC server. This also applies to Pushbullet. In particular, the `SEND_SMS` permission provides access to the SMS manager service and through which app components can send text messages by calling `SmsManger.getDefault().sendTextMessage()`. An alternate method forgoes permissions by instead delegating message sending to a privileged app by means of a `startActivity(intent)` call, where the `intent` argument would have been associated with an SMS-related action. Reading of inbox/draft/outbox/sent messages on the other hand requires access to the SMS provider (`android.provider.Telephony.Sms`), which is populated from an SQLite database file that persists text messages, and which can be accessed through `getContentResolver.query()` calls.

As of Android Kitkat[5] only a designated default messaging app is actually permitted to write to this provider. This app also has exclusive privileges to handle incoming text messages. However it is then obliged to inform all interested apps of a newly delivered message, as well as to be delegated with message sending duties by unprivileged apps. It is perfectly possible that an SMSonPC app is also the designated default messaging app. That would facilitate even further the deletion of sent messages as part of the crime-proxy's functionality.

C. Limitations with existing digital investigation options

In the event of an SMS-hijack incident, existing options for digitally investigating it encompass examining the phone's SIM and flash memory for all stored text messages. This process comprises forensic imaging followed by the decoding steps concerning the manner with which text messages are encoded. SIM memory uses GSM-7 or the now obsolete GSM-8 or UCS2 encodings [2]. Android phones store text messages inside SQLite database files where UTF-8 or UTF-16 string encoding can be employed [3]. In this case there is the added difficulty that Android does not allow flash memory imaging without prior device rooting. In many cases this could be problematic due to warranty voiding, as well as it leaves the device's protection again future re-infection weakened. A more practical solution would be to simply install an SMS backup/recovery app that extracts all tables/columns individually from the SMS provider's SQLite database file. Such apps only require the `READ_SMS` permission to function, in addition to permission to copy messages to the some target destination.

[5]https://android-developers.googleblog.com/2013/10/getting-your-sms-apps-ready-for-kitkat.html

In any case the SMS crime-proxy text messages would have been cleared up using Cloak-and-Dagger steps that interact with the default messaging app. In the spying device's case the context associated with text messages inside the SMS provider only identifies the creator rather than the reading apps and is therefore useless. In fact both scenarios could only be fully reconstructed by tracing and preserving the entire sequence of events that lead to sending/deletion/reading of specific messages. Artifacts found inside volatile memory could potentially serve this purpose, however the ones concerning text messages are expected to be short-lived and all existing volatile memory dumping techniques require device rooting [4], further complicating matters.

III. Volatile memory-centric investigation

The proposed SMS-hijack investigation process is based directly on those components involved in the sequence of events when sending and reading SMS text messages, as controlled by a Cloak-and-Dagger malware. The volatile memory of these components is a candidate source for investigation-relevant artifacts, specifically the text messages themselves. The interfacing between these components is also of interest since the relevant code execution presents candidate triggers, indicating the presence of text messages within the memory areas of interest at that point in time.

A. Abused SMS components

Event sequence mapping for message sending/reading flows as abused during an SMS-hijack incident was carried out directly upon Android's source code[6], with guidance from literature sources that describe its core inter-process communication [5] and telephony stacks [6]. Figure 1 depicts the components and interfacing involved when covertly sending text messages. Firstly, the Cloak-and-Dagger malware itself needs to conceal from the user any activity related to SMS being conducted by the SMSonPC app. Draw-on-top overlays require a `TYPE_SYSTEM_OVERLAY` layout and which has now been deprecated by the more restrictive `TYPE_APPLICATION_OVERLAY`. However the new permission is only relevant for user-installed apps that do not need to be compatible with Android versions older than Oreo. Attackers interact remotely with the SMSonPC app to send instructions for sending/retrieving text messages, e.g. Listing 1, typically through HTTP(S).

The direct route for sending text messages is through the `SMSManager` service and which is hosted by the phone process `com.android.phone`. Most inter-process communication in Android happens through `Binder`, a Remote Procedure Call (RPC) mechanism, in order to trigger `SmsManager.sendTextMessage()`'s code execution. Message dispatching includes two important steps. First, the outgoing message is written to the `mmssms.db` SQLite database file by calling into the Linux Virtual File System (VFS) and eventually writing to the phone's

[6]https://source.android.com/

flash memory. This step is mandatory unless the originator of the message is the default messaging app. Secondly, the message is dispatched to the baseband processor. The `RIL.sendMessage()` triggers a chain of events that cause the outgoing message to formatted in a communication protocol-independent manner (PDU format). It is sent through a UNIX domain socket to the `rild` native daemon that in turn interfaces with a vendor-specific library. When its `ProcessCommandBuffer` event loop receives the RIL request number 25 (`RIL_REQUEST_SEND_SMS`), this library initiates message sending by calling into the baseband driver code via an `ioctl`. Eventually the baseband processor physically sends the text message onwards to the operator's core network via the closest base transceiver station. The operator logs the event for billing purposes even though this excludes message content [2].

An alternate indirect path is possible whenever an unprivileged SMSonPC app interacts with the default messaging app using intents. Intents are resolved by the `ActivityManagerService`, which is hosted by the `System Server` daemon and reachable through Binder RPC. The continuation path is similar to that of the direct path, except that at this point the persistence of outgoing messages is at the discretion of the default messaging app.

The primary components involved with the reading of text messages are shown in Figure 2. The SMS provider inside the phone process, as populated from `mmssms.db` through a call to `SQLiteQueryBuilder.query()`, is central to this operation. This provider can also get populated from the SIM memory through a `SmsManager.getDefault().getAllMessagesfromICC()` call, and which in turn sends a RIL request number 28 (`RIL_REQUEST_SIM_IO`). The task of obtaining a reference to a `ContentResolver` instance for calling `query()` is mediated by `ActivityManagerService`. Ultimately, the retrieved message is covertly leaked to the attacker via HTTP(S).

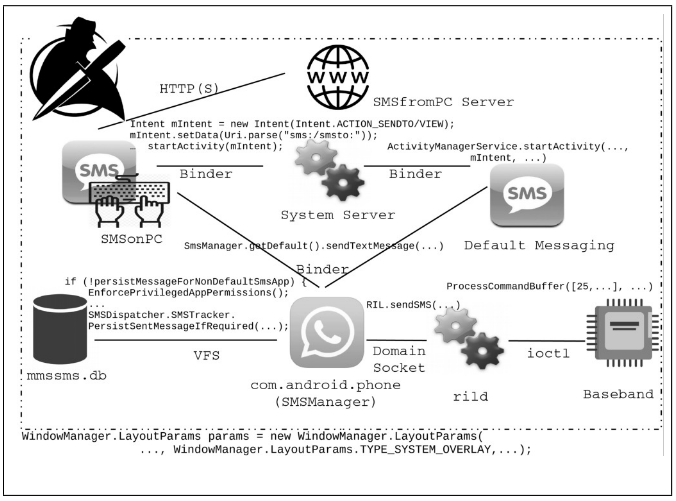
HTTP(S)
SMSfromPC Server
Intent mIntent = new Intent(Intent.ACTION_SENDTO/VIEW);
mIntent.setData(Uri.parse("sms:/smsto:"));
startActivity(mIntent);
ActivityManagerService.startActivity(...,
mIntent, ...)
Binder
Binder
System Server
SMSonPC
SmsManager.getDefault().sendTextMessage(...)
Default Messaging
Binder
if (!persistMessageForNonDefaultSmsApp) {
EnforcePrivilegedAppPermissions();
...
SMSDispatcher.SMSTracker.
PersistSentMessageIfRequired(...);
ProcessCommandBuffer([25,...], ...)
RIL.sendSMS(...)
VFS
Domain
Socket
ioctl
mmssms.db
com.android.phone
(SMSManager)
rild
Baseband
WindowManager.LayoutParams params = new WindowManager.LayoutParams(
..., WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,...);

Fig. 1: Android components abused to send SMS text messages and concealed by Cloak-and-Dagger.

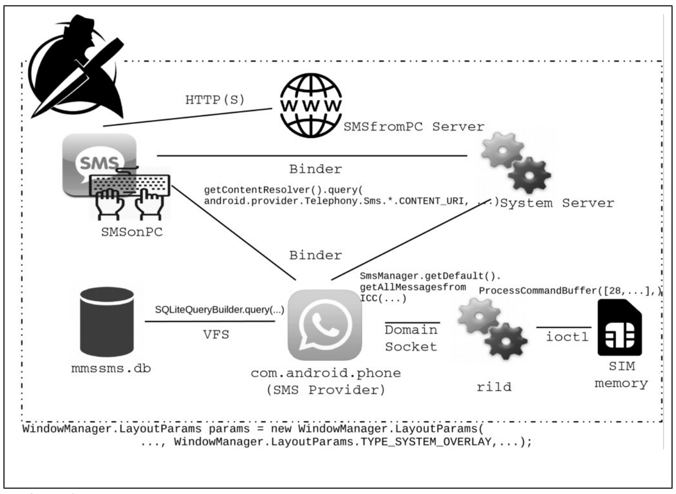
HTTP(S)
SMSfromPC Server
Binder
getContentResolver().query(
android.provider.Telephony.Sms.*.CONTENT_URI, ...)
System Server
SMSonPC
Binder
SmsManager.getDefault().
getAllMessagesfrom
ICC(...)
ProcessCommandBuffer([28,...],)
SQLiteQueryBuilder.query(...)
VFS
Domain
Socket
ioctl
mmssms.db
com.android.phone
(SMS Provider)
rild
SIM
memory
WindowManager.LayoutParams params = new WindowManager.LayoutParams(
..., WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,...);

Fig. 2: Android components abused to read SMS text messages and concealed by Cloak-and-Dagger.

B. Observations

Potentially, any text message flow originating from an SMSonPC app and which after passing through system components terminates in flash/SIM memory, coupled with suspicious draw-on-top and `a11y` activities, should raise an alert of a possible SMS-hijack. The same argument applies for the inverse route. The device owner can confirm whether the observed flows had their consent or otherwise, at which point the suspicious app is identified as the Cloak-and-Dagger malware while the SMSonPC app is confirmed to have been compromised. Given that fully tracing these flows for prompt SMS-hijack detection is expected to be particularly expensive in terms of runtime overheads, an alternate practical approach is to defer detection during memory forensics analysis [7]. The idea is to keep track just of the key in-memory artifacts related to these flows, as occurring inside the memory space of apps and intermediate system components, and from which to infer their occurrence.

The main challenge however is presented by the brief permanence in memory of the said artifacts, calling for an event-driven collection approach. Bytecode instrumentation is a key enabler, whereby injected bytecode is responsible for initiating memory dumps at the appropriate SMS-hijack triggers. The most obvious solution is to focus on SMSonPC apps, since they can be statically instrumented through app repackaging and without the need of device rooting. Furthermore, instrumenting system components is still deemed a less desirable option due to the instability that it might incur. Yet, on a non-rooted device the injected bytecode would only have the faculty to dump the Dalvik (Java) heap, and would therefore miss those artifacts that would rather reside on the native heap whenever native components are employed. In cases where device rooting is viable, rather than having to analyze all native heaps of a myriad of SMSonPC apps it could suffice to inspect just that of the Android phone process. This heap is expected to be relatively constant across devices.

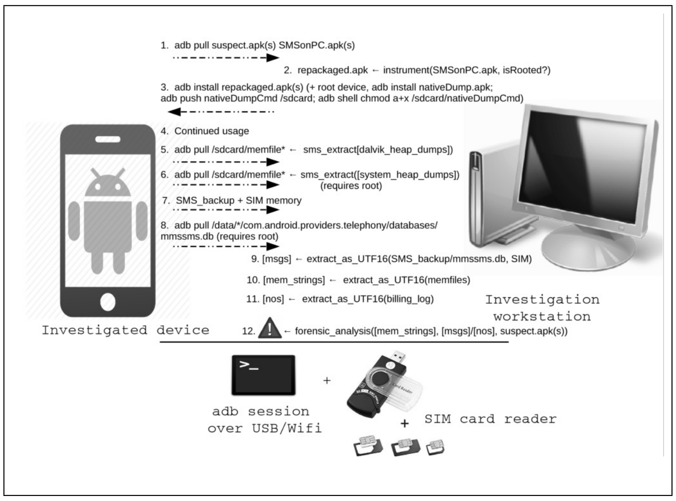
1. adb pull suspect.apk(s) SMSonPC.apk(s)
2. repackaged.apk ← instrument(SMSonPC.apk, isRooted?)
3. adb install repackaged.apk(s) (+ root device, adb install nativeDump.apk; adb push nativeDumpCmd /sdcard; adb shell chmod a+x /sdcard/nativeDumpCmd)
4. Continued usage
5. adb pull /sdcard/memfile* ← sms_extract[dalvik_heap_dumps])
6. adb pull /sdcard/memfile* ← sms_extract([system_heap_dumps]) (requires root)
7. SMS_backup + SIM memory
8. adb pull /data/*/com.android.providers.telephony/databases/mmssms.db (requires root)
9. [msgs] ← extract_as_UTF16(SMS_backup/mmssms.db, SIM)
10. [mem_strings] ← extract_as_UTF16(memfiles)
11. [nos] ← extract_as_UTF16(billing_log)
12. ← forensic_analysis([mem_strings], [msgs]/[nos], suspect.apk(s))
Investigated device
Investigation workstation
adb session over USB/Wifi
+
+
SIM card reader

Fig. 3: The volatile memory-centric process for digitally investigating SMS-hijacking.

As can be observed from Figures 1 and 2 the phone process is central to both the SMS sending and reading flows. Also, due to the native Binder RPC mechanism it must process text message flows on its native heap.

C. The SMS-hijack investigation process

The proposed digital investigation steps carried upon devices with a suspected SMS-hijack, as shown in Figure 3, are based on the observations just made. They require the inspected device to be connected to an investigation workstation over an Android Debug Bridge (adb) session. The first 3 steps are concerned with identifying suspicious apps and instrumenting the trigger points for dumping volatile memory. The focus is on those packages (in apk format) that request relevant permissions. They must either request access to draw-on-top/a11y services that render them Cloak-and-Dagger suspects, or else they request SMS permissions and therefore are potentially abused SMSonPC apps. These apps are discernible from the device's `/data/system/packages.xml`.

Once pulled from the device (step 1), the potential SMSonPC apps are repackaged (step 2) with memory-dumping instrumentation and re-installed on the device in place of the original ones (step 3). Algorithm 1 describes the `instrument()` function applied on each `SMSonPC.apk`, and which takes into account whether the device will be rooted in step 3 as indicated by the `isRoot` flag. Lines 1-4 populate the filters for SMS or native method call-related trigger points as per component interfacing described in section III-A. These are defined using smali syntax[7] which is an assembly language for Dalvik bytecode[8]. The additional wild cards $\star$ are meant to match any smali statements in a non-greedy manner.

Trigger points are instrumented with Dalvik heap-dumping bytecode (`dalvik_dump_instr` on lines 5-12) and possibly also that of the native heap of the phone process (`systemmem_dump_instr` on lines 13-20). The

Algorithm 1: Step 2: `instrument`

```
Input: Potentially abused app: SMSonPC.apk, Root mode:
       RootFlag
Output: Repackaged and cracked app: repackaged.apk
1  [sms_trigger_filters] ← "invoke-direct{*},
     Landroid/telephony/SMSManager;
     ->send(Multipart)TextMessage(*)V";
2  [sms_trigger_filters] ← "const-string "sms*
     invoke-static{*},
     Landroid/net/URI;->parse(*)Landroid/net/URI;*
     invoke-direct{*},
     Landroid/content/Context;->startActivity(*)V";
3  [sms_trigger_filters] ← "(sget-object v*,
     Landroid/provider/Telephony/Sms/*CONTENT_URI:|
     "content://mms-sms")* invoke-direct{*},
     Landroid/content/ContentResolver; ->query(*)V";
4  [native_trigger_filters] ← "invoke*{*},
     *->nativeMethod(*)";
5  dalvik_dump_instr ← "invoke-static{},
     Ljava/lang/System;->currentTimeMillis()J
6  move-result-wide vA
7  invoke-direct{vA}, Ljava/lang/long;
     ->toString()Ljava/lang/String;
8  move-result vC
9  const-string vD, "/sdcard/hdump_hprof_"
10 invoke-direct{vD,vC}, Ljava/lang/String;
     ->concat(Ljava/lang/String;)Ljava/lang/String;
11 move-result vE
12 invoke-static{vE}, Landroid/os/Debug;
     ->dumpHprofData(Ljava/lang/String;)V";
13 systemmem_dump_instr ← "const-string vA,
     "com.inspect.nativeDump"
14 const-string vB ,"com.inspect.nativeDumpSrvc"
15 new-instance vC,
     Landroid/content/ComponentName;
16 invoke-direct{vC, vA, vB},
     Landroid/content/ComponentName;
     -><init>(Ljava/lang/String;
     Ljava/lang/String;)V;
17 new-instance vD, Landroid/content/Intent;
18 invoke-direct{vD}, Landroid/content/Intent;
     -><init>()V;
19 invoke-direct{vD, vC}, Landroid/content/Intent;
     ->setComponent(Landroid/content/ComponentName;
     )Landroid/content/Intent;
20 invoke-direct{p0, vD},
     Landroid/content/Context;
     ->startService(Landroid/content/Intent;)
     Landroid/content/ComponentName;";
21 [smali_class_files] ← unpack_apk(SMSonPC.apk);
22 foreach smali_class_file in [smali_class_files] do
23     smali_class_file ←
         crack_anti_tamper(smali_class_file);
24     smali_class_file ← unpack(smali_class_file);
25     while trigger_point ←
         getNextTriggerPoint(smali_class_file,
         [sms_trigger_filters] ∪ [native_trigger_filters]) do
26         InstrMethodStart(smali_class_file, trigger_point,
             dalvik_dump_instr);
27         if RootFlag then
28             InstrMethodStart(smali_class_file,
                 trigger_point, systemmem_dump_instr);
29     end
30     repackaged.apk ← smali_class_file;
31 end
32 return apkSign(repackaged.apk);
```

[7]https://github.com/JesusFreke/smali

[8]https://source.andreoid.com/devices/tech/dalvik/dalvik-bytecode

latter relies on device rooting as well as the the installation of `nativeDump.apk` and `DumpCmd`, as per step 3 of Figure 3. `nativeDump.apk` exposes the `nativeDumpSrvc` service component reachable through `startService()` calls from the instrumented apps. In turn, `nativeDumpSrvc`'s implementation calls the `su; /sdcard/DumpCmd` shell command sequence. `DumpCmd` is a native process responsible for dumping the phone process's native heap via `/proc/<pid_suspect/phone>/maps` and `/proc/<pid_suspect/phone>/mem`. All memory dumps are placed on external (common app) storage area on the file-system (`/sdcard`). Within the same location the `sms_extract()` component is responsible to extract just the SMS-related memory areas, saving on space requirements. This file-system location facilitates later retrieval from the investigation workstation without requiring device rooting.

For both instrumentation code, the Dalvik VM register numbers `vA-vD` have to be adjusted so that no clashes occur, possibly also requiring an adjustment to the `.locals` smali directive. This directive declares the number of Dalvik VM registers needed to store the local variables of a class method (excluding method parameters). Furthermore, not shown in the instrumentation bytecode of Algorithm 1 is exception handling code, as well as an additional snippet that combined with `AndroidManifest.xml` permission entries for `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` requests access to the external storage. This operation would be required only by those apps not already including this functionality, with the instrumentation bytecode placed inside the `onCreate()` method of the app's main activity.

The trigger filters are applied for each smali representation of the compiled app classes (`smali_class_file`), as obtained through apk unpacking (`unpack_apk()`) (lines 21-31) by calling `getNextTriggerPoint()`. The identified trigger points are then instrumented by calling `InstrMethodStart()`. This is a routine that attempts to inject the instrumentation bytecode at the very start of the method containing the trigger point. This approach avoids having to renumber Dalvik registers to address clobbering, and therefore not running the risk of exceeding the highest register usable by most Dalvik opcodes (`v15`). Instrumentation for native heap dumping is only carried out in case of device rooting. Each, possibly instrumented, smali class file is eventually added to the repackaged app `repackaged.apk`. Finally, the app is signed (line 32) and is ready to be deployed back to the investigated device.

The pending explanation concerns lines 23-24. These are two pre-processing operations that would have to be applied in case the SMSonPC apk is hardened with anti-tampering (`crack_anti_tamper()`) and packed (`unpack()`) code. Their implementation is orthogonal to our work, rather the investigator must seek the assistance of third-party tools in order to successfully pre-process the said class files, with good disassembly skills coming in very handy.

Once step 3 (Figure 3) is complete, the device is returned to its owner for continued usage during step 4. Its duration is bounded by the space available for memory dumps (the `memfiles`). On investigation resumption, steps 5 and 6 take care of retrieving them from the device. Step 7 retrieves the available SMS text messages from flash memory using any SMS backup app, as well as those in SIM memory using an appropriate card reader. Additionally, on rooted devices text messages can rather be extracted directly from the `mmssms.db` SQLite database files in step 8. Steps 9-10 proceed with extracting and normalizing to UTF-16 the text message details as well all strings from the memory dumps. Step 11 on the other hand performs the same operation for those suspicious destination numbers obtained from the billing log.

The normalized content is now ready to be used for forensic analysis. In the case of a text message leakage investigation, the text messages from step 9 are central to the investigation starting point. In the case of a crime-proxy attack, where sent messages are deleted for stealth, the suspicious destination numbers from step 11 become essential. At this point, the aim of the investigator is to trace the messages/numbers inside the dumped strings, and from which to attempt to maximize the identification of SMS-hijack related artifacts. The non-comprehensive list includes: sent/leaked message times, crime-proxy message content, SMSonPC account details in case it has been smuggled, identification of the implicated SMSonPC app in case of multiple candidates, and ultimately the Cloak-and-Dagger malware itself.

IV. Case study: Pushbullet

In order to assess the potential of the proposed SMS-hijack investigation process we present a case study involving the widely used Pushbullet SMSonPC app. The chosen scenario is a simulated crime-proxy attack. Its objectives are to: *i)* Report on the instrumentation step (Algorithm 1) as applied to Pushbullet; *ii)* Measure the storage requirements needed for memory dumps, and *iii)* the overheads imposed by bytecode instrumentation; and finally *iv)* Report on the artifacts identified during the forensic analysis step.

The case study assumes a Cloak-and-Dagger malware to have stealthily installed Pushbullet and set up the device to act as an SMS crime-proxy. Eventually a sequence of suspicious outgoing text message destination numbers show up on a detailed break-down of the device owner's phone bill. Step 1 of the investigation identifies a suspicious app that requests draw-on-top and `a11y` permissions, as well as the Pushbullet app as the possibly abused SMSonPC app. At this stage the investigator is required to conduct the follow-up investigation steps. The full setup consists of an Android Virtual Device (Goldfish), Android Nougat (for Intel Atom), Pushbullet version 17.7.19-288 and Android Debug Bridge 1.0.39. Apktool 2.3.1 was used to assist bytecode instrumentation. Bash scripting was used for prototyping the instrumentation tool as well as native heap dumping.

A. Pushbullet instrumentation

The first two trigger filters from Algorithm 1, i.e. those relevant to SMS crime-proxy, identified two trigger points both inside `com.pushbullet.android.sms.h`'s `void a(String, String, String)` static method. The result of `InstrMethodStart()`'s execution is shown in Listing 2. Line 1 identifies the instrumented method and line 2 shows that the requested number of Dalvik VM registers has been increased from 8 to 12. The registers utilized by the instrumentation bytecode are in the `v2`-`v9` range, since attempts to make use of `v0` and `v1` resulted in compiler (`dex2oat`) errors. As compared to the abstracted version presented earlier in Algorithm 1, the injected instrumentation does not hard-code the location of the external common storage (line 10), makes use of the convenient `StringBuilder` class (line 15), had to resort to using `const-string/jumbo` (line 21) due to the large number of strings used by Pushbullet, and makes use of a try/catch block (line 31). On successful execution, control flow skips the exception handler and goes straight into the original entry point of the non-instrumented method (line 43), as indicated by the original `.prologue` directive (line 42).

In terms of obscured trigger points Pushbullet shows no signs of packed code or SMS-related native code. This situation simplifies matters with respect to trigger point coverage and avoids the need to dump native heaps. The use of ProGuard (a code obfuscator that is enabled by default in Android Studio) is not an obstacle either since since trigger points are defined over Android API calls. No anti-tamper protection was encountered either, although the repackaging of Pushbullet did affect Google sign-in's functionality. The case study was eventually conducted using the Facebook sign-in option since this functionality was not broken. Yet, this was an eye-opener on the perils of instrumentation. Finally since Pushbullet already requests access to external storage, no further bytecode instrumentation was necessary in this respect.

B. Storage requirements

The storage requirements were calculated on the basis of an estimated average of 33 daily sent text messages[9]. In turn this translates to 33 Dalvik heap dumps and a possible additional 33 native heap dumps per day. Table I shows the storage requirements for Pushbullet (Dalvik heap) and Android's default phone process (native heap) dumps. While this case study does not strictly require the latter they are included to present a more complete picture.

In both cases the figures for both full and SMS-related area dumps are provided. In the case of Pushbullet, the SMS areas are those containing ephemeral messages as per Listing 1. In the case of the phone process a more generic approach was followed by taking into consideration all areas containing UTF-8/16 strings. This is the main reason why native heap dump sizes are significantly larger ($> \times 100$). However, dump size reduction is staggering in both cases. The 0 standard

[9]https://www.textrequest.com/blog/many-texts-people-send-per-day

TABLE I
EST. DAILY STORAGE REQUIREMENTS FOR MEMORY DUMPS.

Dump mode	mean (kB)	std. dev. (kB)	sum (kB)
Dalvik heap - Full	11,608	244.752	380,000
Dalvik heap - SMS only	5	1.929	163
Native heap - Full	31,457	0	1,000,000
Native heap - SMS only	505	89.298	16,656

TABLE II
RUNTIME OVERHEADS.

Configuration	mean (s)	std. dev. (s)	Mann-Whitney (p-value=0.93)
`pushbullet.apk`	0.22	0.02	Sum of ranks - 1099
`repackaged.apk`	0.72	0.4	Sum of ranks - 1112
Overheads	227%	-	U - 538

deviation for full native dumps derives from the fact that their size did not change throughout the entire time-frame of sending the text messages. On the other hand the garbage-collected Dalvik heap was more dynamic.

Overall, the 163kB/day required by Dalvik heap dumps compares well to the approximate 3-5MB typically consumed by a selfie with default resolution. However this figure rises sharply to nearly 17MB had the phone to be rooted and native heap dumping enabled.

C. Runtime overheads

From an end-user's point-of-view the runtime overheads incurred by Pushbullet due to bytecode instrumentation are not noticeable. However, even minimal runtime overheads could be a factor from an attacker's point-of-view had they be exploited to detect an ongoing SMS-hijack investigation. Therefore, overheads were measured when sending SMS text messages from Pushbullet's browser interface. In doing so we gained access to the SMS event profiling logs created by `pushbullet.js` inside the javascript console. The relevant log entries are those of `sms_changed` type and examples of which are shown in Listing 3.

Table II shows statistics for the turn-around times, measured between when a text message is sent and the point at which a notification of completion is received asynchronously in a typical Ajax fashion. When computing overheads incurred by the combined Dalvik/native heap dumping instrumentation over an unmodified Pushbullet configuration, the mean overhead for a daily amount of text messages is a considerable 227%. However, when comparing ranks of the two configurations using a Mann-Whitney test the U value is roughly half that of the sum of ranks for both configurations. This indicates that the difference in mean turn-around times between the two is not statistically significant. This outcome indicates that while the turn-around times for the repackaged configuration were higher, other external factors also had an impact. Therefore their difference is not a reliable measure for attackers to detect an ongoing investigation.

```
.method public static a(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V
.locals 12

invoke-static {}, Ljava/lang/System; ->currentTimeMillis()J
move-result-wide v8
invoke-static {v8, v9}, Ljava/lang/Long;->valueOf(J)Ljava/lang/Long;
move-result-object v6
invoke-virtual {v6}, Ljava/lang/Long;->toString()Ljava/lang/String;
move-result-object v5
invoke-static {}, Landroid/os/Environment;->getExternalStorageDirectory()Ljava/io/File;
move-result-object v4

:try_start_0
new-instance v3, Ljava/lang/String;
new-instance v7, Ljava/lang/StringBuilder;
invoke-direct {v7}, Ljava/lang/StringBuilder;-><init>()V
invoke-virtual {v4}, Ljava/io/File;->toString()Ljava/lang/String;
move-result-object v8
invoke-virtual {v7, v8}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v7
const-string/jumbo v8, ”/hdump_hprof_”
invoke-virtual {v7, v8}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v7
invoke-virtual {v7, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v7
invoke-virtual {v7}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
move-result-object v7
invoke-direct {v3, v7}, Ljava/lang/String;-><init>(Ljava/lang/String;)V
invoke-static {v3}, Landroid/os/Debug;->dumpHprofData(Ljava/lang/String;)V
:try_end_0
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

goto :goto_0

:catch_0
move-exception v2
const-string/jumbo v7, ”patchgen”
invoke-static {v2}, Landroid/util/Log;->getStackTraceString(Ljava/lang/Throwable;)Ljava/lang/String;
move-result-object v8
invoke-static {v7, v8}, Landroid/util/Log;->e(Ljava/lang/String;Ljava/lang/String;)I

.prologue
:goto_0
...snip ...
```

Listing 2. A snippet of bytecode instrumentation injected into Pushbullet.

```
pushbullet.js:8615 message {”type”:”push”,”targets”:[”stream”,”android”,”ios”],”push”:{”type”:”sms_changed”,
        ”source_device_iden”:”ujBeKPNHgJMsjAzp2VNDUW”,”notifications”:[]}} 0.042 s
pushbullet.js:8615 message {”type”:”push”,”targets”:[”stream”,”android”,”ios”],”push”:{”type”:”sms_changed”,
        ”source_device_iden”:”ujBeKPNHgJMsjAzp2VNDUW”,”notifications”:[]}} 0.117 s
pushbullet.js:8615 message {”type”:”push”,”targets”:[”stream”,”android”,”ios”],”push”:{”type”:”sms_changed”,
        ”source_device_iden”:”ujBeKPNHgJMsjAzp2VNDUW”,”notifications”:[]}} 0.094 s
...snip ...
```

Listing 3. `pushbullet.js` console log entries for SMS event profiling.

```
”_id”,”thread_id”,”address”,”person”,”date”,”date_sent”,”protocol”,”read”,”status”,”type”,”reply_path_present”,
        ”subject”,”body”,”service_center”,”locked”,”sub_id”,”error_code”,”creator”,”seen”
”1”,”3”,”123456”,””,”1520866381969”,”0”,””,”1”,”-1”,”2”,””,
        ””,”CrimeProxy sms text message 1”,””,”0”,”1”,”0”,”com.pushbullet.android”,”1”
”2”,”3”,”123456”,””,”1520866402042”,”0”,””,”1”,”-1”,”2”,””,
        ””,”CrimeProxy sms text message 2”,””,”0”,”1”,”0”,”com.pushbullet.android”,”1”
”3”,”3”,”123456”,””,”1520866420029”,”0”,””,”1”,”-1”,”2”,””,
        ””,”CrimeProxy sms text message 3”,””,”0”,”1”,”0”,”com.pushbullet.android”,”1”
...snip ...
```

Listing 4. Extract from `mmssms.db`.

D. Forensic analysis

Listing 4 shows an extract from an `mmssms.db` export produced after all text messages were sent. Each entry clearly identifies the destination number (`123456`), message content (`CrimeProxy sms text message` *n*) and the creator app (`com.pushbullet.android`). If this content was present on the device, any SMS backup app with `READ_SMS` permission would have been able to extract this information to solve the SMS-hijack case. However, with a Cloak-and-Dagger malware that deletes the SMS crime-proxy messages, an investigator would have to resort to the volatile memory dumps for evidence. Starting off from the suspicious `123456` destination number extracted from a detailed bill breakdown (step 11 of the investigation process), the subsequent forensic analysis (step 12) retrieved the entries shown in Listing 5. Each entry provides the missing context from the deleted `mmssms.db` entries, namely the message content. Furthermore given that we are dealing with a Pushbullet dump automatically implicates the app in this SMS activity. Moreover the prefix `ujBeKPNHgJMs` is observed to remain constant throughout all the `iden` entries, identifying the utilized Pushbullet account, and therefore also provides the necessary evidence for reporting abuse. Finally, the `created` and `modified` fields store the timestamps related to the text message sending events.

Listing 6 shows the corresponding native heap dump artifacts as retrieved from the phone process. In this case the user account identification is missing, however the message content and creator app are clearly identifiable. Concluding, in both the Dalvik and native heap dump cases, all information that could have gone missing from `mmssms.db` could be reconstructed. At this point with the device owner's assistance the investigator would be able to confirm whether those outgoing messages were related to an SMS-hijack by confirming the user's consent or otherwise. In the latter case, the suspicious app's bytecode from step 1 should be analyzed in order to identify the Cloak-and-Dagger code.

E. Limitations

An alternative to using message turn-around times in order to detect an ongoing investigation, the Cloak-and-Dagger malware could be equipped with checks for the presence of memory dumps inside external storage, suspending its activities if found. While in a way this can be seen as beneficial, this could be problem if the SMS-hijack operation is resumed as soon as the device returns to normal operation. Furthermore, while in the case of Pushbullet no anti-tamper or obfuscation came in the way, the Google sign-in failure is an eye-opener with respect to the difficulties expected during SMSonPC app instrumentation.

V. Related work

Ideally SMS-hijack attacks are thwarted during the app store upload stage using automated malware analysis. Given that a significant part of the attack is actually carried out by a legitimate SMSonPC app, it is rather the identification of the Cloak-and-Dagger malware that should be targeted. Yet, a number of challenges abound. Firstly app obfuscation, e.g. using encryption and runtime class loading, could hide the malware's real intention from static analysis. This issue could be addressed with dynamic analysis [8] where suspicious apps are executed inside a malware sandbox. However this alternative is not without its own limitations, with trigger-based behavior [9], [10] and device emulation detection-based evasion [11] posing major hurdles. The same limitations are encountered whenever malware analysis is carried out for forensics purposes [12], where malware samples are hunted and extracted from within a mobile device for event reconstruction purposes.

In contrast, our proposed digital investigation process differs in scope. It targets those situations where Cloak-and-Dagger malware succeeds in evading app store scanning. Furthermore, the malware's behavior is tracked within its intended runtime environment, with the exception for SMSonPC app repackaging and the resulting dumps. Our work is more akin to related work concerning the digital investigation of mobile devices, for example for SMS text message forensic purposes [13], [14]. Yet, our proposed technique involves a prolonged investigation period, where the device is returned to its owner for continued usage as enhanced with memory dumping instrumentation. Finally, our proposition can also pave the way to thwart the 'Trojan Horse defense' [15], where text messages considered as evidence for a crime investigation are refuted by claims that the device could have actually been compromised to serve as a communication proxy by the actual criminals.

VI. Conclusions

In this paper we considered the problem of Cloak-and-Dagger malware pulling off stealthy SMS-hijacks by abusing legitimate SMSonPC apps. We proposed a solution whereby injected bytecode instrumentation dumps the SMS-relevant areas of volatile memory from the device under investigation at the right triggers. A case study was carried out using Pushbullet as the SMSonPC app abused for setting up an SMS crime-proxy. Results show that the technique can be both effective in collecting the evidence required to solve the SMS-hijack, as well as practical in terms of SMSonPC app instrumentation and storage costs. The runtime overheads incurred were shown to be difficult to exploit by attackers to uncover an ongoing investigation, while at the same time not impacting the device owner.

This case study provided the right setting for initial exploration of the proposed SMS-hijack investigation process, with results showing promise. A similar case study for information leakage is planned. Further experimentation also aims to evaluate the technique at a larger scale using an array of physical smart-phone devices and possibly even involving malware samples captured in the wild. A primary pre-requisite for such an undertaking is the engineering of the investigation tool that also incorporates existing techniques that deal with obfuscated

```
{"active":true,"iden":"ujBeKPNHgJMsjz7aNoLJeK","created":1.5208663683633862E9,"modified":1.520866368366242E9,
"data":{"target_device_iden":"ujBeKPNHgJMsjAzp2VNDUW","addresses":["123456"],"guid":"vfhj9v3t24o2q9544u0frg",
"message":"CrimeProxy sms text message 1"}}!
...snip ...
{"active":true,"iden":"ujBeKPNHgJMsjAsOdablfg","created":1.5208664012296782E9,"modified":1.520866401232248E9,
"data":{"target_device_iden":"ujBeKPNHgJMsjAzp2VNDUW","addresses":["123456"],"guid":"ctqvmagsveodhll4hldpv",
"message":"CrimeProxy sms text message 2"}}!
...snip ...
{"active":true,"iden":"ujBeKPNHgJMsjz2mR5H8yy","created":1.520866418990317E9,"modified":1.520866418994791E9,
"data":{"target_device_iden":"ujBeKPNHgJMsjAzp2VNDUW","addresses":["123456"],"guid":"9uv0upvsb1gdjch1qe3f1g",
"message":"CrimeProxy sms text message 3"}}!
...snip ...
```

Listing 5. Dalvik heap dump extracts.

```
123456'\n'CrimeProxy sms text message 1com.pushbullet.android
...snip ...
123456'\n'CrimeProxy sms text message 2com.pushbullet.androidV
...snip ...
123456'\n'CrimeProxy sms text message 3com.pushbullet.androidV
...snip ...
```

Listing 6. Native heap dump extracts.

code and anti-tamper checks. Even more importantly, collaboration with SMSonPC app developers is required to deal with app instrumentation in a cleaner way whenever this breaks functionality in some way. Collaboration is specifically sought on the anti-tampering front.

REFERENCES

[1] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and Dagger: from two permissions to complete control of the UI feedback loop," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017. doi: 10.1109/SP.2017.39 pp. 1041–1057.

[2] Y. Leguesse, C. Sidiropoulos, and L. Palkmets, *Mobile Threats Incident Handling (Part II)*. enisa, 2015.

[3] C. Anglano, "Forensic analysis of WhatsApp Messenger on Android smartphones," *Digital Investigation*, vol. 11, no. 3, pp. 201–213, 2014. doi: 10.1016/j.diin.2014.04.003

[4] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and analysis of volatile memory from Android devices," *Digital Investigation*, vol. 8, no. 3, pp. 175–184, 2012. doi: 10.1016/j.diin.2011.10.003

[5] A. Gargenta, "Deep dive into Android IPC/Binder framework," in *AnDevCon: The Android Developer Conference*, 2012.

[6] A. Singh and A. Bhardwaj, "Android internals and telephony," *Int. J. Emerg. Technol. Adv. Eng*, vol. 4, pp. 51–59, 2014.

[7] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons, 2014.

[8] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis: Android malware under the magnifying glass," *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001*, 2014.

[9] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 2013. doi: 10.1145/2536853.2536881 p. 68.

[10] S. Pooryousef and M. Amini, "Enhancing accuracy of Android malware detection using intent instrumentation." in *ICISSP*, 2017. doi: https://doi.org/10.5220/0006195803800388 pp. 380–388.

[11] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "BareDroid: Large-scale analysis of Android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015. doi: 10.1145/2818000.2818036 pp. 71–80.

[12] J. Li, D. Gu, and Y. Luo, "Android malware forensics: Reconstruction of malicious events," in *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*. IEEE, 2012. doi: 10.1109/ICDCSW.2012.33 pp. 552–558.

[13] M. I. Husain and R. Sridhar, "iForensics: forensic analysis of instant messaging on smart phones," in *International Conference on Digital Forensics and Cyber Crime*. Springer, 2009. doi: 10.1007/978-3-642-11534-9-2 pp. 9–18.

[14] I. Murynets and R. Piqueras Jover, "Crime scene investigation: SMS spam data analysis," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012. doi: 10.1145/2398776.2398822 pp. 441–452.

[15] C. M. Steel, "Technical soddi defenses: The Trojan Horse defense revisited," *The Journal of Digital Forensics, Security and Law: JDFSL*, vol. 9, no. 4, p. 49, 2014.