Memory Forensics of Insecure Android Inter-app Communications

Mark Vella and Rachel Cilia
Department of Computer Science, University of Malta, Msida, Malta
{*mark.vella, rachel.cilia.13*}*@um.edu.mt*

Keywords: Mobile Systems Security, Android Memory Forensics, Capability Leaks.

Abstract: Android is designed in a way to promote the implementation of user task flows among multiple applications inside mobile devices. Consequently, app permissions may be leaked to malicious apps without users noticing any compromise to their devices' security. In this work we explore the possibility of detecting insecure inter-app communications inside memory dumps, with forensic analysis results indicating the possibility of doing so across the various layers of Android's architecture. Yet, for the detailed evidence reconstruction that could be required during digital investigation, current capabilities have to be complemented with evidence collected through live forensics. We propose that this process should still be based on carving forensic artifacts directly from memory.

1 INTRODUCTION

Android, the most popular mobile device operating system, is designed in a way to both isolate potentially security-critical apps from each other as well as to have apps seamlessly delegate their functionality to third-parties (Elenkov, 2014). Clearly these two design objectives are at odds, leading to various insecure inter-app communication links such as when vulnerable apps leak their legitimately granted privileges to malicious ones, known as capability leaks. In this work we begin to explore how to expose inter-app links directly from physical memory; which is a technique that does not have to rely on possibly tampered code, and which successfully executing malware cannot evade. The value proposition is that of increased systems security of Android-powered mobile devices.

Android's Binder is the inter-process communication mechanism through which all inter-app links are established (section 2). Code comprehension is utilized to predict the relevant forensic artifacts across Android's architectural layers (section 3). Our primary contribution lies in proposing a memory dump parsing algorithm for each layer (section 4) and subsequently used to verify artifact suitability through forensic analysis (section 5). Results show that by combining artifacts identified within different sections of memory, all types of inter-app links can be disclosed from memory dumps despite varying degrees of reliability. However for the level of detail that may be required during digital investigation, a live forensics approach would be necessary. Therefore we conclude this work (section 6) with a proposition that relies on collecting evidence directly from live memory on-demand, using runtime patching.

2 ANDROID INTER-APP COMMUNICATION

Android apps are composed of classes that extend framework components, namely: activities (app windows), services (long running, mostly background, code), broadcast receivers (handlers of device-wide events) and content providers (shared data repositories); eventually fleshed out with application logic that makes use of Android system services, e.g. messaging, telephony, Internet-access, location services and so on (Gargenta, 2012). The first three of these components are activated through *intents* whilst the latter through *content resolvers*.

Intents are application framework constructs representing the 'intent'ion of an application component to provide some functionality. Intent-servicing components can even be exported for use by external applications, thereby constituting a primary IPC construct. Explicit intents identify specifically the target apps hosting the activities/services to be activated in correspondence to the specified intent action (listing 1). On the other hand, implicit intents only identify the action and possibly some other attributes (listing 2). In this case Android takes care of routing them to

Vella, M. and Cilia, R.
Memory Forensics of Insecure Android Inter-app Communications.
DOI: 10.5220/0006215504810486
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 481-486
ISBN: 978-989-758-209-7

those apps that would have declared interest in servicing them during install time, also possibly asking the user to choose between multiple such candidate apps. On the other hand, content resolvers provide access to external content providers.

Listing 1: Explicit intents are used to launch activities or services inside external applications.

```
//Starting an external activity:
PackageManager pm = this.getPackageManager();
try
{
    Intent it = pm.getLaunchIntentForPackage("
        com.example.mvella.eiserver");
    if (null != it) this.startActivity(it); ..
        SNIP...

//Starting an external service:
Intent intent = new Intent("com.example.mvella.
    eiserver.ACTION_START");
intent.setComponent(new ComponentName("com.
    example.mvella.eiserver", "com.example.
    mvella.eiserver.DiffAppStartService"));
startService(intent);
```

Listing 2: Implicit intents only specify actions along with various possible qualifiers.

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.
  parse("http://www.google.com")));
```

2.1 Capability Leaks

Intents and content resolvers are core Android application development constructs encouraging inter-app functionality and data sharing, providing the required support for user task flow implementation across multiple apps. A number of system intents are in fact made available by the Android framework itself and are expected to be serviced by pre-installed/default apps e.g. picture viewers, instant messaging, and telephony. The transition from one app to another can be so seamless that it may not be even noticed by the device user. Since Android makes sure to isolate Android apps from each other as well as from sensitive device services (e.g. telephony and networking), a permissions system is used to relax the level of isolation in a controlled manner (Elenkov, 2014). System permission requests are declared inside manifest files (app package meta-data) through the `uses-permission` tag.

Listing 3 (lines 3-4), for example, requests read/write access to the device's contacts. Yet, it is the responsibility of the app developer to check that external apps activating the `updateContacts` service (line 6) through the `ACTION_START` intent (line 8) and that makes use of these permissions, also have the sufficient privilege level to access the device's contacts (e.g. by calling `Context.checkCallingPermission()`). Otherwise a *capability leak* is said to occur whenever an unprivileged app obtains contacts through it (Zhang et al., 2016). In this work we focus specifically on Intent-derived leaks, given that content resolvers should normally be used to directly access system/external data repositories only, rather than potentially permission-leaking code.

Listing 3: Vulnerable app's `AndroidMainfest.xml`.

```
1
2  <package name="com.example.mvella.vulnapp"
       codePath="...SNIP...
3      <uses-permission android:name="android.
           permission.READ_CONTACTS"/>
4      <uses-permission android:name="android.
           permission.WRITE_CONTACTS"/>
5  ...SNIP...
6          <service android:name="com.example.
               mvella.vulnapp.updateContacts">
7              <intent-filter>
8                  <action android:name=".
                       ACTION_START"/>
9              </intent-filter>
10
11         </service>
12 ...SNIP...
```

2.2 Related Work

While preventable both from secure coding and permission enforcement hardening stances (Zhang et al., 2016), capability leaks are still bound to happen in existing devices due to the difficulty concerned with complete eradication of vulnerabilities. We focus on the provision of graceful recovery using memory forensics. Robust memory acquisition from Android devices has been explored extensively (Sylve et al., 2012). This body of knowledge underpins the `LiME` acquisition tool utilized in this work. Memory acquisition differs from disk image acquisition since the address bus space in typical microprocessor architectures is not exclusive to physical RAM, but is rather shared with all memory-mapped hardware devices. Erroneously reading from non-RAM addresses compromises system integrity. This same issue could however be leveraged by malware to evade memory acquisition (Stüttgen and Cohen, 2013). Mobile device forensics is challenging even at secondary storage level due to the variety of available custom hardware (Kong, 2015). However, the wide-spread adoption of Android creates an opportunity for a common source of in-memory forensic evidence. In this work we focus specifically on Android Binder and which

security-criticality has already been targeted by offensive security studies (Artenstein and Revivo, 2014).

3 ANDROID BINDER

All IPC constructs available to Android app developers (Gargenta, 2012) are implemented as Remote Procedure Calls (RPC), with support from Android Binder and shared memory (ashmem). At the highest level of abstraction developers are oblivious of this detail, however this mechanism presents the focus for forensics artifact exploration. Figure 1 illustrates the main components involved in supporting inter-app links across the framework (Java), middleware (C/C++) and the (modified) Linux kernel (C) layers of the Android architecture. A client app *(App_1)* accesses various system service (e.g. Telephony service) using framework API calls. System service code actually resides in separate OS processes, most of which is lumped into *system_server*. Therefore in actual fact all such calls already constitute IPC. Moreover, system services may just constitute a stepping stone so that *App_1* can activate components inside another, *App_2*, and eventually communicating directly using a Message Queue or an RPC type of inter-app communication.

We resort to code comprehension in order to formulate predictions about locating and parsing artifacts related to inter-app links. This is similar to academic work performing forensic analysis for on-disk or in-network traffic artifacts (Anglano et al., 2016). We made use of Android Open Source Project's (AOSP) master branch[1] for this purpose. The comprehension exercise mainly consisted of (statically) tracing execution flow involving components communicating across app boundaries, and identifying those data structures useful for forensic event reconstruction. In this section we summarize the primary binaries involved and a representative cross-application boundary flow, with specific detail concerning locating and parsing artifacts presented in the following one. The primary source code files utilized during comprehension are shown in parenthesis.

Let's take the example when *Activity1* inside *App_1* sends a message to a service component exposed by *App_2* using the API `Messenger.send(Message aMsg)` method. What happens here is that `aMsg` is passed onto the middleware layer through (android.os.Binder)'s `transact()` method by `android.os.Messenger` stub code (`core/java/android/os/IMessenger.aidl`). Specifically, using the Android Runtime (ART)'s (*libart.so*) Java-Native Interface (JNI) mediation, the proxy creates a serialized representation of `aMsg` inside the middleware layer (`libs/binder/Parcel.cpp`), across a Java-native link registered by *libandroid_runtime.so* (`core/jni/android_util_Binder.cpp`). Subsequently it is passed onto the Binder framework: *libbinder.so* (libs/binder/BpBinder.cpp), that first packages it into a Binder transaction that conforms to the Binder driver's protocol[2] and then dispatches it to the Binder driver (`drivers/staging/android/binder.c`) inside the kernel using an `ioctl()` call on the */dev/binder* device file (`libs/binder/IPCThreadState.cpp`). The role of the driver is to look up the location of the requested remote service, the service inside *App_2* in this example, and delivers `aMsg` to it where an associated thread pool would be blocked awaiting Binder transactions (in turn from a previous `ioctl()` call from `libs/binder/IPCThreadState.cpp`).

Transaction delivery is completed using a kernel memory region that is mapped to *App_2*'s user-space (`libs/binder/ProcessState.cpp`). In case of large messages, a faster alternative would be to only serialize a file descriptor associated with a shared memory region between *App_1* and *App_2* (accessible through the */dev/ashmem* device file through *libcutils.so*). On message delivery completion it is the Binder framework's `onTransact()` method that gets called first (`libs/binder/JavaBBinder.cpp`), eventually calling into `android.os.Messenger`'s proxy `onTransact()` method. The last phase of the delivery involves in unserializing `aMsg` and passing it to the messenger's handling routine as argument. In this example Binder's RPC mechanism is used to let *App_1* and *App_2* communicate using a message queue abstraction, however it also possible to use RPC directly, with service proxy/stub code generated from Android Interface Definition Language (AIDL) files providing the necessary glue code. This is the case for example when *App_1* calls into *system_server* to start a service inside *App_2* through a `bindService()` API call for example.

The above flow relies on two very important registries. The first one is the *Activity Manager* service that operates at the framework layer and resolves intents based on a number of activity/service/broadcast records that are registered with it, in turn based on information derived from the various manifest files of installed apps. The second registry is part of

[1] https://source.android.com/source/index.html

[2] http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html

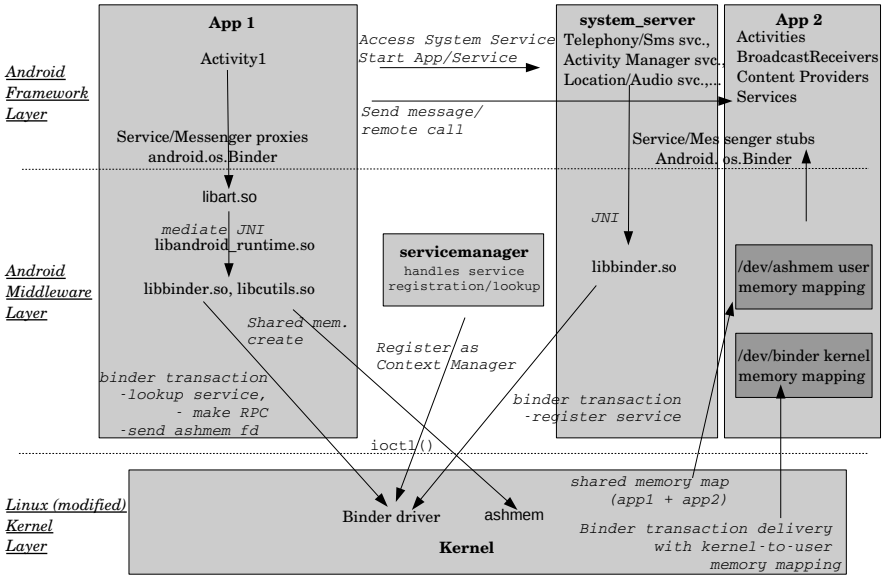

Figure 1: Android inter-app communication is implemented by Android's Binder with support from Android shared memory.

the Binder framework. The Binder driver needs to perform sufficient book-keeping so that remote objects can be registered with it, and then have client apps look them up to obtain a handle for them and through which perform remote calls. Given that both these operations are sensitive a central process, *servicemanager*, mediates between all processes and the Binder driver. This process is called the *Context Manager*, and only one process can be registered as such. Communication between processes and servicemanager still occurs through Binder RPCs, and to which a specially reserved handle (0) is utilized. Whenever enabled, Binder driver's book-keeping structures are dumped to *debugfs* (`/sys/kernel/debug/binder` file-system path).

4 MEMORY DUMP PARSERS

All three parsers take a dump of physical memory as input along with an `androidBinderProfile` object. This is an abstract data object that contains: the virtual addresses of global variables; data structure offsets; and an initial Directory Table Base (DTB) (CR3 register value). This last value is the all-important offset inside the physical memory dump for the page table directory which is used for virtual-to-physical address translation of kernel memory. Specifically, `androidBinderProfile` is used by the `getKernelAddressSpace()` function which recreates the kernel's virtual address space.

At the kernel layer (Algorithm 1) the Binder book-keeping structures (`drivers/staging/android/binder.c`) have been identified as a linked list of `binder_proc` and associated structures, and which can be located through the `binder_procs` global variable. These structures keep track of the processes hosting remotely accessible objects, binder objects, and the threads servicing the inter-process request/replies among others. Additionally, the well-known Linux linked list of `task_struct` (Mauerer, 2010) and associated structures is required to recover the virtual memory sections of individual Linux processes hosting app components. This linked list can be located using the `init_task` global variable.

One layer upwards, the middleware, it is expected that *Parcel*s containing binder transaction (Android RPC calls) data intended for the Activity Manager service can be identified through the `android.app.IActivityManager` length-encoded wide character string. Specifically, serialized intents within such parcels include the package and component names of the targets in addition to target-intended data arguments. They are created on the client-side's process heap and eventually get copied to a location inside kernel memory that in turn would already have been mapped inside the target process as `/dev/binder` during the binder object's creation.

Target app component identifiers and arguments in turn would be stored inside ART's own heap at the framework layer during intent creation (for e.g. listing 1).

Algorithm 2 shows the framework-layer parser. It takes the additional `/data/system/packages.xml`

Algorithm 1: Kernel-layer parser.

```
input : androidMemDump, androidBinderProfile
output: Per-app binder object references

task_struct aTaskStruct;
hashmap procNameMap;
binder_proc aBinderProc;
binder_node aBinderNode;
binder_ref aNodeRef;

kernel_addsprace LinuxAS ←
  getKernelAddressSpace(androidMemDump,
  androidBinderProfile);

foreach aTaskStruct ← traverseLinkedList(LinuxAS,
  androidBinderProfile, global_addr="init_task",
  datatype="task_struct") do
    procNameMap
      [aTaskStruct.pid]←aTaskStruct.comm;
end

foreach aBinderProc ←
  traverseBinaryTree(LinuxAS, androidBinderProfile,
  global_addr="binder_procs",
  datatype="binder_proc") do
    output("process pid & name:", aBinderProc.pid,
    procNameMap [aBinderProc.pid]);
    foreach aBinderNode ←
      traverseBinaryTree(LinuxAS,
      androidBinderProfile, addr=aBinderProc.nodes,
      datatype="binder_node") do
        output("node id:", aBinderNode.debug_id);
        output("referenced by:");
        foreach aNodeRef ←
          traverseLinkedList(LinuxAS,
          androidBinderProfile,
          addr="aBinderNode.refs",
          datatype="binder_ref") do
            output(aNodeRef.proc.pid, procNameMap
              [aNodeRef.proc.pid]);
        end
    end
end
```

to assist searching for all wide character strings that include an app package name. Effectively this is a signature that takes advantage of the fact that package names are used as name-spaces for intent arguments. The framework's `android` package and the package for the current app being analyzed are excluded given that no inter-app links are concerned. Directory table addresses for individual processes are obtained from `task_struct.mm->pgd`. This is the offset inside the physical dump used by `getAndroidRuntimeHeap()` to reconstruct ART's heap. The memory sections for each process are locatable through the `task_struct.mm->mmap` linked list of `vm_area_struct` structures inside the kernel space. `vm_area_struct.vm_file->f_path.dentry->d_iname` is the final memory lookup required to filter just the memory sections corresponding to ART's heap, i.e. the ones corresponding to device file maps prefixed with `/dev/ashmem/dalvik`. The algorithm for the middleware layer is similar, with the main difference being that this time it is the process heap (delimited by `task_struct.mm->start_brk` and `task_struct.mm->brk` start/end addresses) and the `/dev/binder` memory sections that are scanned for content tagged with `android.app.IActivityManager` wide character strings.

Algorithm 2: Framework-layer parser.

```
input : androidMemDump, androidBinderProfile,
        packages.xml
output: Per-app intent/content provider-related strings

task_struct aTaskStruct;
buffer heapBuffer;
Array packageArray ←
  getPackageList(packages.xml);
String packageName;
String match;

kernel_addsprace LinuxAS ←
  getKernelAddressSpace(androidMemDump,
  androidBinderProfile);

foreach aTaskStruct ← traverseLinkedList(LinuxAS,
  androidBinderProfile, global_addr="init_task",
  datatype="task_struct") do
    output("App: ", aTaskStruct.comm, "pid: ",
      aTaskStruct.pid);
    heapBuffer ← getAndroidRuntimeHeap(LinuxAS,
      aTaskStruct.mm→mmap, aTaskStruct.mm→pgd,
      mappingid="/dev/ashmem/dalvik*");
    foreach packageName ← (packageArray
      \{android, aTaskStruct.comm}) do
        foreach match ←
          scanWideCharStr(heapBuffer,packageName)
          do
            output("Match: ", match);
        end
    end
end
```

5 PRELIMINARY RESULTS

Experimentation was carried out on a system image created using Android 6.0.0 (Marshmallow - API level 23) and a kernel v3.4 image modified to support kernel modules. Prototype implementations of the memory parsers were developed for the `Volatility 2.5` memory forensics framework and utilize the `LiME` memory extractor kernel module.

Table 1 shows the forensic analysis results obtained for the six IPC configurations, with `debugfs` providing a live forensics baseline. As expected,

Table 1: Memory forensic analysis results.

IPC	debugfs	Kernel	Middleware	Framework
Explicit intent - Start activity	×	×	×	✓
Explicit intent - Start service	×	×	×	✓
Explicit intent - Bind service (messaging)	✓	✓	×	✓
Explicit intent - Bind service (RPC)	✓	✓	×	✓
Implicit intent - Start activity	×	×	×	✓
Implicit intent - Send broadcast	×	×	×	✓

its effectiveness in uncovering IPC matches the kernel layer dump parser and which corresponds to applications communicating directly with each other. On the other hand, apps communicating indirectly through Activity Manager are fully detectable only at the framework layer. The middleware layer, Android Binder's user-level code, misses out on all of them through a design that favors immediate transaction recycling. The memory parsing prototype was then extended into a full capability leak detector, with on-device app package dumping and reversal (using apktool). This procedure enables disclosure of the apps' requested permissions and consequently the identification of inter-app links that cross privilege boundaries. Its detection capability was demonstrated using a case study app that leaks its contacts permissions (manifest shown in listing 3) by not performing any privilege checks on the caller apps.

Listing 4: Memory artifacts disclosing the suspicious inter-app link.

```
App: com.example.mvella.malapp
pid: 822
==> com.example.mvella.vulnapp.action.INSERT
==> com.example.mvella.vulnapp#
==> com.example.mvella.vulnapp.updateContacts
==> com.example.mvella.vulnapp.extra.NAMEp#
==> com.example.mvella.vulnapp.extra.TEL
...SNIP
```

6 CONCLUSION

While valuable, the current detector cannot identify specific targets of implicit intents whenever multiple target apps are possible. It also lacks details of the exact IPC sequences to support in-depth investigations since it is restricted to operate just upon instantaneous memory snapshots. This can be achieved through runtime patching of ARM/Intel atom machine instructions by adding an Android middleware layer process that leverages `ptrace` to load runtime memory forensics patches supplied over network connections or removable flash memory. This security-critical process should be protected through an appropriate `SELinux` policy, with only vendor/approved forensic investigator-signed code patches being accepted while at the same time verifying Binder's integrity. This idea can extend to include adaptive permission enforcement, whereas the user could be prompted to consciously either patch the vulnerable app, or even elevate the privilege level of the caller app in case of false positives.

REFERENCES

Anglano, C., Canonico, M., and Guazzone, M. (2016). Forensic analysis of the chatsecure instant messaging application on android smartphones. *Digital Investigation*, 19:44–59.

Artenstein, N. and Revivo, I. (2014). Man in the Binder: He who controls IPC, controls the droid. In *Europe BlackHat Conf.*

Elenkov, N. (2014). *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press.

Gargenta, A. (2012). Deep dive into Android IPC/Binder framework. In *AnDevCon: The Android Developer Conference*.

Kong, J. (2015). Data extraction on MTK-based Android mobile phone forensics. *Journal of Digital Forensics, Security and Law*, 10(4):31–42.

Mauerer, W. (2010). *Professional Linux kernel architecture*. John Wiley & Sons.

Stüttgen, J. and Cohen, M. (2013). Anti-forensic resilient memory acquisition. *Digital investigation*, 10:S105–S115.

Sylve, J., Case, A., Marziale, L., and Richard, G. G. (2012). Acquisition and analysis of volatile memory from Android devices. *Digital Investigation*, 8(3):175–184.

Zhang, D., Wang, R., Lin, Z., Guo, D., and Cao, X. (2016). Iacdroid: Preventing inter-app communication capability leaks in android. In *ISCC, 2016 IEEE Symposium on*, pages 443–449. IEEE.