Data-in-Use leakages from Android Memory - Test and Analysis

Pasquale Stirparo*†, Igor Nai Fovino*, Ioannis Kounelis*†
*Institute for the Protection and Security of the Citizen
Joint Research Centre (JRC), European Commission, Ispra (VA), Italy
†Royal Institute of Technology (KTH), Stockholm, Sweden
{pasquale.stirparo, igor.nai-fovino, ioannis.kounelis}@jrc.ec.europa.eu

Abstract—**Due to their increasing pervasiveness, smartphones and more in general mobile devices are becoming the citizen's companions in the daily life activities. Smartphones are today the repositories of our secrets (photos, email), of our money (online e-commerce) and of our identities (social networks accounts). Therefore mobile applications have the responsibility of handling such sensitive and personal information in a proper, secure way. This paper present the second phase of the MobiLeak project, analysing how mobile applications manage users data when these are loaded in the volatile memory of the device. Scope of this work is to raise the awareness of the research and development communities on the poor attention that is generally paid in the secure development of mobile applications.**

Keywords*—*mobile privacy; security; memory analysis; mobileak project.

I. Introduction

Mobile devices continue to spread rapidly and have heavily modified people's everyday behavior. A great part of the success of smartphones is due to the increasing availability of mobile applications conceived to answer to every possible need of the user. Many of this 'apps' are used daily, influencing and assisting users routines. As they are becoming part of one's life, it is of great importance to ensure the user's security and privacy when handling personal and sensitive information.

The importance of security is highlighted by OWASP (Open Web Application Security Project) and ENISA (European Network and Information Security Agency), who have released secure guidelines practices for developing mobile applications [1]. Unfortunately, the demanding need for innovative mobile applications and the urge of the companies to be the first to deliver a new product, to win the majority of the clients, have mostly left security and privacy outside of the development cycle.

A. The MobiLeak Project

Scope of this paper is to present the results of the second phase of the MobiLeak research project. The goal of MobiLeak is to identify and analyse the privacy threats to which end-users are exposed while using smartphones and mobile applications.

The first phase of MobiLeak [2] was devoted to the privacy assessment on data found in the "Data at Rest" state. In other words, during this phase, the data available in the internal storage of the phone or the SD card were analysed. Data in this state, to be considered secure, should be encrypted with a reasonable long key, moreover, the encryption key should not be available on the same media of storage of the data and should be difficult to generate through an automated procedure.

The results obtained showed that many of the mobile applications did not follow the secure coding best practices [1] [3]. Amongst the well known applications analysed, the 83% revealed personal information such as name of the user, address, last digits of credit card, etc., 6% revealed activity info such as log events, people the user interacted with, duration of sessions etc.; the 25% revealed the password of the login credentials in clear text, while all the applications revealed the username. To conclude, half of the analysed applications revealed in clear text documents that were received during the applications runtime (e.g. pictures exchanged during a Skype call, etc.). The results of the Mobileak phase 1 made evident how the world of mobile applications is far from being considered secure and for that reason merits further investigations and tests.

B. Motivation

In this paper are presented the results of the second phase of the MobiLeak project, dealing with the analysis of the *"Data in Use"* state. Data in this state is data found in the memory of the phone, i.e. data that is currently or has recently been manipulated. Sensitive information found in the *data in use* state can be exploited by malware in various ways. For example, the most direct exploration would be to impersonate a user's identity by finding his/her account's credentials, i.e. username and password.

We chose to target Android as the underlying operating system. The main motivation behind this decision was to exploit the open nature of Android and have direct access to the kernel and the source files that were needed for our research. Moreover, we wanted to target the most used mobile operating system in order for our research to be applicable for the largest amount of mobile users and Android is at the moment the most sold operating system, gaining a 75% market share in the third quarter of 2012 [4]. Finally, recent numbers show that Android is increasingly targeted by malware [5], making it even more important to identify potential misuse of personal and sensitive information that malware may have access to.

The analysis focused mainly on two different groups of applications: mobile banking applications and a subgroup of some of the most popular Android applications. The first category is a very sensitive one: being able to recover the credential for online banking represents a high risk for the user. Potential leakage of information from a bank transaction will have a direct economical impact on the user that could be much higher than having a credit card lost or stolen. The reasons behind the choice of the second category were mainly two: on one hand we wanted to give a continuity to the first phase of MobiLeak [2], testing the same applications analysed in phase-1 also against Data in Use threats, on the other hand this group of applications represents some among the most popular mobile applications, therefore privacy issues affecting them will automatically affect millions of users.

The rest of this paper is structured as follows: Section II introduces research works related to this paper while in Section III are described the basic characteristics of the memory in general, and the memory in Android more specifically. In Section IV we describe the experiment setup and the methodology we used to implement our tests. In Section V we provide details about the memory acquisition phase of the tests while in Section VI the memory analysis phase is described. In Section VII we present the results of our research and in Section VIII we draw our conclusions proposing countermeasures as well as ideas for future works.

II. Related Work

Most of the work related to live memory analysis has been done in the context of digital forensics that is, as defined at the first Digital Forensics Research Workshop (DFRWS) in 2001, *the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations* [6]. Forensics investigations targeting volatile data that can be found in a systems main memory (RAM) is also known as *live forensics* [7] or *RAM forensics* [8]. In this context, *live forensics* tries to express that the focus lies on a systems current state that can be obtained by freezing the scene at a specific point in time. This needs to be performed while the system is live and operational. If that is not the case (e.g. because the main power has been removed) the volatile data is lost. The main memory contains the whole state of an operating system, including running and historical processes, open network connections, management data, and personal data. Being Android based on Linux, most of the relevant works for this research have been carried on both Android and Linux memory analysis areas.

A. Linux Memory Analysis

Traditionally on Linux it was possible to perform memory captures by accessing the `/dev/mem` device. Such device contained a map of the first gigabyte of RAM and allowed acquisition only of the first 896 MB of physical memory, without the need to load code into the kernel. However, due to security concerns the `/dev/mem` device has recently been disabled on all major Linux distributions, as it allowed reading and writing of kernel memory. To overcome these problems, Kollar created *fmem* [9], a loadable kernel module that creates a `/dev/fmem` device supporting memory capture. However, this solution appears to be unsuitable for Android, since it makes use of some kernel functions that are not available on ARM [10].

In [8], Urrea describes the case of a specific Linux distribution by outlining kernel structures relevant for memory management that can be used to retrieve corresponding data. In his solution he uses the `dd` tool to read the physical memory from `/proc/mem`.

Andrew Case has done an extensive amount of work in the field of memory forensics, making lots of efforts to extract forensically relevant information from memory captures [11], and perform deep analysis of Linux kernel data structures as well as userland information [12] [13]. Although all these works have been able to gather numerous objects and data structures from memory, a shortcoming is their inability to deal with the vast number of Linux kernel versions and the large number of widely used Linux distributions.

B. Android Memory Analysis

In [14], Thing, Ng and Chang focus on capturing a specific, running processes, using the ptrace functionality of the kernel to dump specific memory regions of a process, instead of capturing the whole physical memory of Android. The virtual memory captures are then analysed to discover evidence. This approach requires, obviously, memory to be extracted separately for each process of interest.

In [10], Sylve, Case, Marziale and Richard present methods that obtain complete captures of volatile memory from Android devices, along with subsequent analysis of that data in both userland and the kernel. They developed kernel module, DMD, to perform full dump of the device memory, as well as kernel analysis support for the Volatility framework [15][16], implementing ARM-specific support. However, this solution doesn't solve one common problem all modules have, which is the security mechanism of the kernel called module verification. This mechanism is intended to prevent the kernel from loading incompatible or possibly malicious code into the operating system. Therefore, since it is not possible to load a module in a kernel-agnostic way, an alternative solution is to create a pool of precompiled modules against a specific kernel, which basically would mean one module for each device and Android version. This is feasible only for those devices for which the corresponding vendor releases the kernel source code together with its build configuration. As explained in more details later, our acquisition of the memory dump is based on this work.

In [17], Leppert discusses several methods to generate heap dumps from the first Android version till 2.2 (Froyo), and from Android 2.3 (Gingerbread) till version 4.0 (Ice Cream Sandwich). These methods are based on old Android versions and rely on tools no longer available. Moreover, his solution it is not applicable to commercial applications bought or downloaded from an application market and therefore are not scalable. In fact acquiring a heap dump is only possible for applications prepared for debugging. When developing Android applications, there is a flag called `android:debuggable` in the applications configuration file, the `AndroidManifest.xml`. If that option is set to *true*, it causes the application to open a debug port that can be used by DDMS [18] to acquire a heap dump from the application running on a device, which has to be physically connected to a computer. This usually means that the application is still in testing phase, since from the moment the application is available online the `android:debuggable` flag is supposed to be set to *false*.

In [19], Macht takes Android Live Memory Forensics to the next level by performing a deep analysis of the Linux kernel, the Dalvik Virtual Machine and a chosen set of applications. The result is a set of plugins that extend the Volatility Framework, to read data such as user names, passwords, chat messages, and emails chosen from a set of target applications.

III. Memory Management in Android

We can generalize the memory organization as a structure divided into three main areas, also known as segments: the text segment, the stack segment, and the heap segment. The **text segment**, or code segment, is where the compiled code of the program itself resides. The **stack segment** is where local variables are stored, and it is also used for passing arguments to functions along with the return address of the instruction which is going to be executed after the function call is over. When a new function is called and a new stack frame needs to be added, the stack grows downward. Finally, the **heap segment**, or data segment, is the area of memory that gets allocated at runtime and therefore contains the variables that are defined during the program execution and their value. When more memory needs to be allocated, the heap grows upward.

Android is an open source, Linux-based operating system primarily designed for mobile devices such as smartphones and tablets. Previously based on the Linux kernel version 2.6, it uses version 3.x starting from Android 4.0 Ice Cream Sandwich onwards. Android uses the Dalvik virtual machine with just-in-time compilation to run Dalvik 'dex-code' (Dalvik Executable), which is usually translated from Java bytecode. DalvikVM is an interpreter for the Java programming language. It is similar to the Java Virtual Machine (JVM), but it has been specifically designed to operate in embedded environments.

In Android every application runs within a separate process, which has its own instance of the DalvikVM. Being Android a multi-user Linux system, in which each application is a different user, the system by default assigns to each application a unique Linux user ID (the ID is used only by the system and is unknown to the application). The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them. Android has at its core the "Zygote" process, which starts up at *init*. When you start an application, the Zygote is forked, and the Dalvik heap is preloaded with classes and data by Zygote. Dalvik, like virtual machines for many other languages, does garbage collection (GC) on the heap [20]. Garbage collection is a form of automatic memory management that attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the process.

Android memory management involves freeing objects from memory when they are no longer needed and assigning memory to processes that require it. As stated in the Android Developers portal [21], to determine which processes to keep and which to kill, the system places each process into an "importance hierarchy" based on the components running in the process and their state. Processes with the lowest importance are eliminated first, then those with the next lowest importance, and so on, as necessary to recover system resources. There are five levels in the importance hierarchy. The following list presents the different types of processes in order of importance (the first process is most important and is killed last):

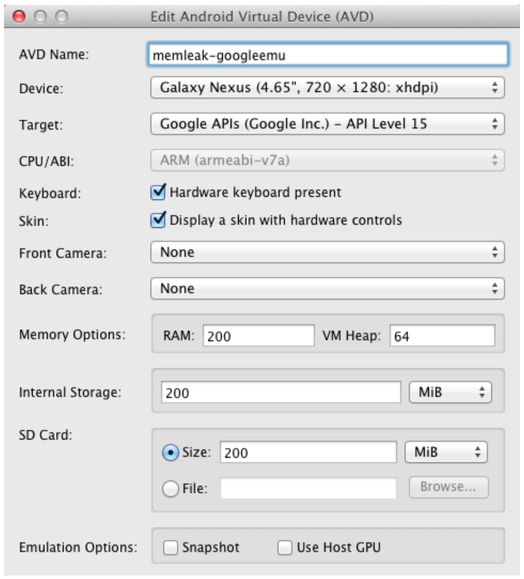
1) Foreground process. A process that is required for what the user is currently doing. Generally, only a few foreground processes exist at any given time. They are killed only as a last resort, if memory is so low that they cannot all continue to run.
2) Visible process. A process that doesn't have any foreground components, but still can affect what the user sees on screen. A visible process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.
3) Service process. This process started some nonactive services that do not interact directly with the user.
4) Background process. A process holding an activity that is not currently visible to the user (the activity's onStop() method has been called). These processes have no direct impact on the user experience, and the system can kill them at any time to reclaim memory for a foreground, visible, or service process. Usually there are many background processes running.
5) Empty process. A process that does not hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it.

As one could imagine, the management of the memory plays a key role under a privacy perspective, as all the user's data needed by an application to fulfill its duties will be stored somewhere in the smartphone memory.

IV. Experiment Setup and Methodology Used

As already anticipated in Section I, scope of the presented experimental campaign was to analyse the memory content of a smartphone (a) while using a certain application and (b) just after closing the target application. The focus of the analysis was on two different groups of applications: mobile banking applications and a subgroup of some of the most popular Android applications. For each application, two types of tests have been performed. In the first one we took a dump of the memory immediately after hitting the "login" button, while the application was still running. In the second test, the memory dump was taken right after quitting from the application. While it might be relatively "normal" to find sensitive data in memory during the application execution, still being able to retrieve the same data after the application has been terminated would highlight a bad memory management procedure either by the application developer or by the Android team, or both. Table I summarizes the steps taken for each of the two types of tests.

As testing environment we used the Android emulator [22] version 21.1.0, which requires Android SDK [23] and Android NDK [24] to be properly installed and initialized. For a complete documentation on how to do so, the reader should refer to the official webpages. We then used the LiME kernel module cross-compiled against the device kernel to dump the



Edit Android Virtual Device (AVD)

AVD Name: memleak-googleemu
Device: Galaxy Nexus (4.65", 720 × 1280: xhdpi)
Target: Google APIs (Google Inc.) - API Level 15
CPU/ABI: ARM (armeabi-v7a)
Keyboard: [✓ Hardware keyboard present]
Skin: [✓ Display a skin with hardware controls]
Front Camera: None
Back Camera: None
Memory Options: RAM: 200 VM Heap: 64
Internal Storage: 200 MiB
SD Card: [● Size: 200 MiB] File: Browse...
Emulation Options: Snapshot Use Host GPU

Fig. 1. Parameters used to generate Android Virtual Device

memory of the emulated device, and the Volatility Framework to analyse the dumps. We will explain later what LiME and Volatility exactly are.

After setting up the environment, in order to create the virtual device we used Android Virtual Device Manager (AVD), choosing as device to be emulated a Galaxy Nexus (see Fig. 1). As target platform we chose the Google API version 15, corresponding to Android 4.0.3 *Ice Cream Sandwich*, because it's the most used of the Android versions based on the new kernel and the second most deployed version on the market in general [25]. Once the virtual device has been defined, we need to get the kernel source code from the device manufacturer website. In our case, since we are using the virtual device, we can use the Android emulator source code, code name *Goldfish*. We picked the kernel version 2.6.29 with the following steps:

```
$ git clone https://android.googlesource.com/kernel/
        goldfish.git ~/android-goldfish
$ cd ~/android-goldfish/
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.4
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master
$ git checkout -t remotes/origin/android-goldfish-2.6.29
        -b goldfish
```

As last step of this preliminary setup phase, we need to cross compile the kernel source code for our system, which runs on an ARM architecture. To do so we first need to set the following environment variables:

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
```

and then we can complete the compilation with the following commands:

```
$ make goldfish_armv7_defconfig
$ make
```

If everything went well, we should be able to run the virtual device created before with the kernel we just compiled

```
$ cd <path-to-android-sdk>/sdk/tools/
$ ./emulator -avd <virtual device name> -kernel ~/android-
        goldfish/arch/arm/boot/zImage -show-kernel -verbose
```

It has to be noticed that an original .config file is needed in case we want to compile the kernel for a real Android device. In such case we first need to export the config.gz file from the device, which unfortunately is not always possible, and then decompress it and place the .config file exported in the kernel source folder. The config.gz file can be exported using the following command:

```
$ adb pull /proc/config.gz
```

TABLE I. STEP-BY-STEP PROCEDURE FOR THE TWO TYPES OF TESTS

Step	Test 1	Test 2
1	Reboot phone	Reboot phone
2	Launch the app	Launch the app
3	Login	Login
4	Memory dump	App is running
5	Finish	Quit app
6	//	Memory dump
7	//	Finish

V. MEMORY ACQUISITION

As mentioned in [17], until Android version 2.2 (Froyo) one of the methods to obtain the heap dump was to send a SIGUSR1-signal to the process. In the DavlikVM the SIGUSR1-signal is defined to force the Garbage Collector and hprof heap dumping. The easiest and most common way to do that was to send the SIGUSR1-signal to the process using the *kill* command:

```
# kill -10 <PID>
```

This kill command will trigger the system to force GC and dump the process heap. Accessing the system log via logcat, it would show something like the following:

```
I/dalvikvm(\emph{PID}): SIGUSR1 forcing GC and HPROF dump
I/dalvikvm(\emph{PID}): hprof: dumping VM heap to
"/data/misc/heap-dump-tmYYYYYYYYYY-pidXXX.hprof-hptemp".
I/dalvikvm(\emph{PID}): hprof: dumping heap strings to
"/data/misc/heap-dump-tmYYYYYYYYYY-pidXXX.hprof".
```

The above described action was possible until Android version 2.2 [26]. As anticipated in Section II, to acquire the memory dumps for further analysis we used Linux Memory Extractor, a.k.a. LiME [27] [28]. As explained by Sylve in the project's documentation, LiME (formerly DMD [10], see Section II) is a Loadable Kernel Module (LKM), which allows the acquisition of volatile memory from Linux and Linux-based devices, therefore Android systems too. To our knowledge, at the time of writing, LiME is the first and currently only tool that allows full memory captures from Android devices. Thanks to its nature of Loadable Kernel Module, LiME also minimizes its interaction between user and kernel space processes during acquisition. In order to acquire the physical memory from the operating system, the LiME module a) parses the kernel structure called *iomem_resource* to get the physical memory address ranges, b) performs virtual to physical address translation for each memory area, and c) reads all pages in each range from RAM. To complete the LiME module preparation for our dump, we need to point the makefile variables *KDIR* and *CCPATH* to the kernel directory and android cross-compiler for ARM in the NDK directory respectively. Once the module has been compiled, we first load it into the device:

```
$ adb push lime.ko /sdcard/lime.ko
$ adb forward tcp:4444 tcp:4444
$ adb shell
$ su
# insmod /sdcard/lime.ko "path=tcp:4444 format=lime"
```

and then we setup *netcat* on listening mode in the host computer:

```
$ nc localhost 4444 > ram-dump.lime
```

LiME requires two compulsory parameters that are *path*, which takes either a filename to write on the local system (SD Card) or tcp:<port>, and *format*, which can have one of the following values:

- *raw*, simply concatenates all System RAM ranges;
- *padded*, pads all non-System RAM ranges with 0s, starting from physical address 0;
- *lime*, each range is prepended with a fixed-sized header which contains address space information.

We chose *lime* as dump format since Volatility address space has been developed to support this format, and we will use Volatility Framework to analyse the dumps later.

VI. Memory Analysis

To analyse the memory dumps, we used the Volatility Framework [15] [16], an open source collection of tools implemented in Python, for the extraction of digital artifacts from volatile memory (RAM) samples. The first step in order to use Volatility, is to create a device profile, as explained in the project's webpage documentation. Basically a device profile is a zip file with information on the kernel's data structures and debug symbols, which is used by Volatility in order to locate and parse critical information. To create kernel's data structures (vtypes) we need to compile *'module.c'*, present in the volatility source code under *'tools/linux'*, against the kernel we want to analyse. We first customize the Makefile in the folder related to the *'module.c'* file:

```
obj-m += module.o
KDIR := /Volumes/android-fs/android-sources/goldfish/
CCPATH := ~/android-ndk-r8d/toolchains/arm-linux-
     androideabi-4.7/prebuilt/darwin-x86/bin
-include version.mk
all: dwarf
dwarf: module.c
        $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-
        androideabi- -C $(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD)
        modules /Tools/dwarf-20130207/dwarfdump/dwarfdump -di
        module.ko > module.dwarf
```

Then we run the make command, which will produce the file *module.dwarf*:

```
$ cd volatility/tools/linux
$ make
$ head module.dwarf

.debug_info

<0><0x0+0xb><DW_TAG_compile_unit> DW_AT_producer<GNU C 4.7>
DW_AT_language<DW_LANG_C89> DW_AT_name </volatility/tools/
linux/module.c> DW_AT_comp_dir</Volumes/android-fs/android-
sources/goldfish> DW_AT_stmt_list<0x00000000>
<1><0x1d><DW_TAG_base_type> DW_AT_byte_size<0x00000004>
DW_AT_encoding<DW_ATE_unsigned> DW_AT_name<long unsigned int>
<1><0x24><DW_TAG_pointer_type> DW_AT_byte_size<0x00000004>
DW_AT_type<<0x0000002a>>
...
```

After that, to get the symbol list we need to grab the System.map file for the kernel we want to analyse. This can be found in the main directory of the compiled kernel source. Regarding this file, while for Linux systems it is also possible to find the System.map file under the /boot directory, for mobile device this is kind of equivalent to the */proc/kallsyms* file. However, based on our experience it is strongly recommended to use the System.map file generated by compiling the kernel source code, since the */proc/kallsyms* file is actually missing many symbols and may drive Volatility to raise errors (i.e. ValueError). Only if everything else fails, */proc/kallsyms* should be used. Finally, to create the profile we need to place both the *module.dwarf* and the *System.map* file into a zip file, and then move this zip file under *volatility/plugins/overlays/linux/*:

```
$ zip /volatility/volatility-read-only/volatility/plugins/
        overlays/linux/Golfish-2.6.29.zip module.dwarf
        /Volumes/android-fs/android-sources/goldfish/
        System.map
  adding: module.dwarf (deflated 90\%)
  adding: Volumes/android-fs/android-sources/goldfish/
          System.map (deflated 73\%)
```

Once the correct profile has been created, since Android is based on Linux we can use most of the Linux commands/plugins to analyse the memory dump. In order to find the data related to the user inputs information (i.e. usernames, passwords, etc.) for a specific application, we first need to identify the process id (PID) of the target application, then we map the process in the memory, in order to find the offsets of the *heap*, which is the area of the memory we are interested in, and lastly we dump the *heap*. To do that, we used the following three Volatility plugins:

- *linux_pslist*, which gathers active tasks by walking the task_struct->task list
- *linux_proc_maps*, which gathers process maps for linux
- *linux_dump_map*, which writes selected memory mappings to disk

Table II presents the case of the *eBay* application, as one example out of the 26 total applications analysed for this research work. As described above, once we find the PID of the eBay process we use Volatility to identify the offsets and then dump the process' heap and the DalvikVM's heap. The reason why we find two heaps is because, as explained in Section III, when an application is launched it runs in its own process, which contains its own instance of the DalvikVM. Therefore one heap is the native heap of the process itself, and the other is the internal heap of the virtual machine. Since the application runs inside the DalvikVM, we expect to find input data inserted by the user (i.e. usernames, passwords, pin codes, etc.) eventually in the dalvik-heap if not in both.

Memory dumps are binary files, to look into them we used the command line utility `string` and an hex-editor. In the analysis we looked, for the same value, for two different types of encoding: ASCII and Unicode. For example, if the password used in the test was *my-password* we looked for both:

- *my-password*

TABLE II. OUTPUT AND STEP-BY-STEP OF THE VOLATILITY COMMANDS USED FOR THE ANALYSIS

```
$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_pslist
Offset     Name                 Pid             Uid             Gid    DTB        Start Time
---------- -------------------- --------------- --------------- ------ ---------- ----------
0xca969400 com.ebay.mobile      379             10067           10067  0x0aec8000 2013-03-29 09:22:08 UTC+0000


$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_proc_maps -p 379 | grep heap
Pid      Start      End        Flags       Pgoff Major  Minor  Inode      File Path
-------- ---------- ---------- ------ ---------- ------ ------ ---------- ------------------------------
     379 0x0000b000 0x003d1000 rw-           0x0      0      0          0 [heap]
     379 0x409b2000 0x42124000 rw-           0x0      0      7        353 /dev/ashmem/dalvik-heap
     379 0x42124000 0x449b2000 ---     0x1772000      0      7        353 /dev/ashmem/dalvik-heap
     379 0x46e02000 0x46e03000 r--           0x0      0      7        368 /dev/ashmem/SurfaceFlinger read-only heap


$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_dump_map -s 0x0000b000 --dump-dir ~/memdump/ebay-heap/
Task       VM Start   VM End         Length Path
---------- ---------- ---------- ---------- ----
       379 0x0000b000 0x003d1000   0x3c6000 /memdump/ebay-heap/task.379.0xb000.vma


$ python vol.py --profile=LinuxGolfish-2_6_29x86 -f ebay.lime linux_dump_map -s 0x409b2000 --dump-dir ~/memdump/ebay-heap/
Task       VM Start   VM End         Length Path
---------- ---------- ---------- ---------- ----
       379 0x409b2000 0x42124000  0x1772000 /memdump/ebay-heap/task.379.0x409b2000.vma
```

- 6D 00 79 00 2D 00 70 00 61 00 73 00 73 00 77 00 6F 00 72 00 64 00, which is the hexadecimal equivalent of the ASCII value *"m.y.-.p.a.s.s.w.o.r.d."* and of the Unicode value *"my-password"*

We also looked for the eventual presence of identification keywords such as "username", "password", "pin", etc., because the potential danger of the evidence found increases if preceded by something that uniquely identifies it. In fact, even if leaving a password in cleartext is wrong, if this password is randomly placed with other information it may not be easily identifiable as the password, if at all. Therefore, if a malicious user/application does not know the password itself, placing the password value in cleartext that follows its identification label in a form like *'password=my-password'* makes it easily recoverable for anyone who looks for it.

VII. RESULTS

As claimed in Section I we took under consideration for our test campaign 26 different applications, which means 52 memory dumps analysed. We looked for user credentials, namely username, password, pin code for the bank, etc., either in the process heap and the dalvik-heap. The results confirmed that the heap of reference is mainly the dalvik-heap, which is the container where the application is executed in. In the following we describe the results of the testing campaign for both the banking applications and general purposes applications sets.

A. Banking Apps Results

Banking applications are a quite sensitive field of analysis especially considering the direct impact of their vulnerabilities on the citizen life. For this analysis campaign we considered 15 different mobile banking applications. As the scope of this work is not to perform dedicated "penetration tests" of the banking environment but rather to raise the attention of the security and development communities on the possible implications of un-careful development of these applications, we will not disclose here the name of the banking applications analysed.

Table III summarises the results of our investigation with regards to the banking applications. More in details, the table reports our findings in the native heap and in the Dalvik heap during the application execution and after its termination. In an average of 75% of the applications we found username and password following their respective identification label, in both types of dumps (i.e. during and after the application execution). In 16% of the applications analysed we found username and password in clear (ASCII and/or Unicode), but without any identification that could help identify them. Only in 7% of the cases we did not find any trace of the user credentials in the memory.

TABLE III. STATISTICS RELATED TO THE BANK APPLICATIONS GROUP

DUMP	TYPE	HEAP		DALVIK-HEAP	
		USERNAME	PASSWORD	USERNAME	PASSWORD
during execution	with id	26.7	20.0	80.0	73.3
	without id	6.7	6.7	20.0	13.3
	nothing found	66.7	73.3	0.0	13.3
after execution	with id	26.7	20.0	73.3	80.0
	without id	6.7	6.7	20.0	13.3
	nothing found	66.7	73.3	6.7	6.7

B. General Purposes Apps Results

In the second group of applications, where we analysed 11 of the most popular Android applications already studied in the first research work related to MobiLeak [2] namely *Box, Dropbox, eBay, Evernote, Facebook, Groupon, Linkedin, Monster, Skype, Tweetdeck* and *Twitter*, the situation reported is slightly better. Against what expected in the process heap, we found the username in an average of 73% of the applications, 55% were following their identification label. On the other hand in 90% of the cases we did not find the password at all while the application was still running, reaching 100% if we consider the dumps taken after the application has been terminated. The improvement compared to the previous group regards the dalvik-heap. Also here there is a significant difference between usernames and passwords occurrences found, while in the first group the situation was a bit more homogeneous. In 90% of the cases we found the username while the application was running, 45% of which following their identification label. In both types of dumps, while the application was running and after its execution, only in the 9% of the cases we did not find any occurrences of username. We found passwords occurrences in about 80% of the cases while the application was running, just 20% of which following their identification label. Instead, from the dumps taken after the application has been terminated we found passwords occurrences in about 45% of the cases, 0% of them linked to an identification label, and 55% of the time we did not find any occurrence. Table IV summarises the described results.

TABLE IV. STATISTICS RELATED TO THE MOBILEAK1 APPLICATIONS GROUP

DUMP	TYPE	HEAP		DALVIK-HEAP	
		USERNAME	PASSWORD	USERNAME	PASSWORD
during execution	with id	63.6	9.1	45.5	18.2
	without id	18.2	0.0	45.5	63.6
	nothing found	18.2	90.9	9.1	18.2
after execution	with id	54.5	0.0	63.6	0.0
	without id	18.2	0.0	27.3	45.5
	nothing found	27.3	100.0	9.1	54.5

VIII. CONCLUSION AND FUTURE WORK

The results of the presented testing campaign can be considered in some way unexpected. In fact, the tests clearly show that a huge amount of personal and critical data are stored without the due care in the mobile-device memory. This fact is surprising, considering that the first group of tests were related to banking applications (i.e. an area where cyber-security is taken in high consideration) and the second group of tests were related to companies with huge amount of investments in cyber-security. From the results of our analysis campaign we can clearly state that there is an issue concerning both privacy and security as, in average, applications keep sensitive data in cleartext in their memory space. However, to evaluate correctly the impact of our findings and to propose a set of solutions, it is indeed needed to make a distinction between the results related to the memory analysis while the application is in use and the results related to the memory analysis after closing the application.

While the application is running, the presence of cleartext data in the memory at a certain point in time is unavoidable. In this case the first and most immediate thing to do would be **to avoid keeping such data coupled with its clear identification label**. If a malicious user/application is looking for a password without knowing it, storing a password which follows the keyword *'password'* would make the research straightforward. It is also true that this would not completely solve the issue of keeping data in cleartext. In some cases a determined attacker could reverse engineer the data structures allocated in the memory by the application, being able to locate the password or other specific data anyway. To mitigate the impact of this attack-approach, the OS manufacturer should provide APIs that would help the developer to **clear that specific sensitive data from the memory after it has been used**, since after a successful login there is no need to keep the credentials in the memory waiting for someone to grab them.

Another *"bad practice"* we identified is that **most of the applications analysed do not have a 'quit' button** or they have it hidden within a series of menus, making it hard to find. The only way to quit these applications is often by pushing the phone home button, which leaves the process still running in the background for a while and with it all the data allocated in the memory. This also explains why we found the **processes still running after exiting from the applications for 25 out of 26 applications analysed**. Only one process immediately terminated when quitting the application, and which had a proper "quit" button implemented. Although it may be also due to the underlying OS when the process remains running still for a while after the application has been terminated, **developers should make sure that the quit button implements procedures to clean the memory before shutting down the application**.

Unfortunately the results of our campaign magnify one more time how the cyber-security is a matter of secure-development and secure-design. Cyber-security by design is a costly approach in term of development time and resources and the *App* world is by nature based on a completely different paradigm (rapid development and low costs). Nevertheless as mobile applications are becoming the daily companions of everybody, the *App* model need to be reviewed as soon as possible to protect the citizen's privacy and security. With regards to our research activity, to raise the attention on these new cyber-threats, we plan to continue in the development of the MobiLeak project, concentrating our attention on the third state in which data can be found on mobile-devices, i.e. *data in transit*.

ACKNOWLEDGMENT

The authors would like to thank Andrew Case, Michael Hale Ligh and the Volatility community for their support and assistance in the testing of the Volatility Framework with their new born support for the Android platform.

REFERENCES

[1] ENISA and OWASP, "Smartphone secure development guidelines," November 2011. [Online]. Available: http://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1/smartphone-secure-development-guidelines (Access date: 19/04/2013)

[2] P. Stirparo and I. Kounelis, "The mobileak project: Forensics methodology for mobile application privacy assessment," in *Internet Technology And Secured Transactions, 2012 International Conferece For*. IEEE, Dec. 2012, pp. 297–303.

[3] viaForensics, "Secure mobile development best practices," 2012. [Online]. Available: https://viaforensics.com/resources/reports/best-practices-ios-android-secure-mobile-development/

[4] TechCrunch, "Idc: Android market share reached 75% worldwide in q3 2012," November 2012. [Online]. Available: http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012/ (Access date: 08/04/2013)

[5] TheNextWeb, "In one year, android malware up 580%, 23 of the top 500 apps on google play deemed high risk." Oct. 2012. [Online]. Available: http://thenextweb.com/google/2012/10/25/in-one-year-android-malware-up-580-23-of-the-top-500-on-google-play-deemed-high-risk/ (Access date: 08/04/2013)

[6] DFRWS, *A Road Map for Digital Forensic Research*. Report from the 1st Digital Forensic Research Workshop (DFRWS), 2001, p. 16.

[7] F. Adelstein, "Live forensics: diagnosing your system without killing it first," *Communications of the ACM*, vol. 49, no. 2, pp. 63–66, 2006.

[8] J. M. Urrea, "An analysis of linux ram forensics," Ph.D. dissertation, Monterey, California. Naval Postgraduate School, 2006.

[9] I. Kollár, "Forensic ram dump image analyser," Master's thesis, Department of Software Engineering, Charles University, Prague, 2010.

[10] 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.

[11] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev, “Face: Automated digital evidence discovery and correlation,” *Digital Investigation*, vol. 5, Supplement, no. 0, pp. S65 – S75, 2008, the Proceedings of the Eighth Annual DFRWS Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287608000340

[12] A. Case, L. Marziale, C. Neckar, and G. G. Richard III, “Treasure and tragedy in kmem_cache mining for live forensics investigation,” *Digital Investigation*, vol. 7, Supplement, no. 0, pp. S41 – S47, 2010, ¡ce:title¿The Proceedings of the Tenth Annual {DFRWS} Conference¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287610000332

[13] A. Case, L. Marziale, and G. G. Richard III, “Dynamic recreation of kernel data structures for live forensics,” *Digital Investigation*, vol. 7, Supplement, no. 0, pp. S32 – S40, 2010, ¡ce:title¿The Proceedings of the Tenth Annual DFRWS Conference¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287610000320

[14] V. L. Thing, K.-Y. Ng, and E.-C. Chang, “Live memory forensics of mobile phones,” *digital investigation*, vol. 7, pp. S74–S82, 2010.

[15] Volatile System, “The volatility framework: Volatile memory artifact extraction utility framework.” [Online]. Available: https://www.volatilesystems.com/default/volatility (Access date: 08/04/2013)

[16] “The volatility framework.” [Online]. Available: https://code.google.com/p/volatility/ (Access date: 08/04/2013)

[17] S. Leppert, “Android memory dump analysis,” *Student Research Paper, Chair of Computer Science*, vol. 1, 2012.

[18] Google, “Using ddms.” [Online]. Available: http://developer.android.com/tools/debugging/ddms.html (Access date: 19/04/2013)

[19] H. Macht, “Live memory forensics on android with volatility,” Master’s thesis, Friedrich-Alexander University Erlangen-Nuremberg, January 2013.

[20] EmbeddedLinux, “Android memory usage.” [Online]. Available: http://elinux.org/Android_Memory_Usage (Access date: 08/04/2013)

[21] “Processes and threads.” [Online]. Available: http://developer.android.com/guide/components/processes-and-threads.html (Access date: 08/04/2013)

[22] “Android emulator.” [Online]. Available: http://developer.android.com/tools/help/emulator.html (Access date: 08/04/2013)

[23] “Android sdk.” [Online]. Available: http://developer.android.com/sdk/index.html (Access date: 08/04/2013)

[24] “Android ndk.” [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html (Access date: 08/04/2013)

[25] “Dashboard - platform versions.” [Online]. Available: http://developer.android.com/about/dashboards/index.html (Access date: 08/04/2013)

[26] A. McFadden, “Don’t do heap dump on sigusr1.” [Online]. Available: https://github.com/android/platform_dalvik/commit/b037a464512c0721 bdca969ae19cce3d4b17b083#vm/SignalCatcher.c (Access date: 08/04/2013)

[27] J. Sylve, “Android mind reading,” in *ShmooCon Security Conference*, 2012.

[28] J.Sylve, “LiME - Linux Memory Extractor.” [Online]. Available: https://code.google.com/p/lime-forensics/ (Access date: 08/04/2013)