Fall 12-2012

Volatile Memory Message Carving: A "per process basis" Approach

Aisha Ibrahim Ali-Gombe
University of New Orleans, aaligomb@uno.edu

Volatile Memory Message Carving: A “per process basis” Approach

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Information Assurance

By

Aisha Ali-Gombe

B.S University of Abuja, 2005
MBA Bayero Univerity Kano, 2011

December, 2012

Table of Contents

List of Figures

Volatile Memory Message Carving: A "Per Process Basis" Approach

Ali-Gombe Aisha

Department of Computer Science, University of New Orleans

ABSTRACT

The pace at which data and information transfer and storage has shifted from PCs to mobile devices is of great concern to the digital forensics community. Android is fast becoming the operating system of choice for these hand-held devices, hence the need to develop better forensic techniques for data recovery cannot be over-emphasized. This thesis analyzes the volatile memory for Motorola Android devices with a shift from traditional physical memory extraction to carving residues of data on a "per process basis". Each Android application runs in a separate process within its own Dalvik Virtual Machine (JVM) instance, thus, the proposed "per process basis" approach. To extract messages, we first extract the runtime memory of the MotoBlur application, then carve and reconstruct both deleted and undeleted messages (emails and chat messages). An experimental study covering two Android phones is also presented.

1. Introduction

Android has the largest smartphone user base, constituting about 61% of the total market in the US and Canada. Motorola, which was recently acquired by Google, has 29% of the total Android market making it second next to HTC.

Messaging today plays a vital role in our lives ranging from emails, SMS to social networking. The ability to bring these messaging flavors on our mobile devices for fast, easy and accessible data and information transfer is one of the key revolutionary changes smartphones brought to the mobile world. Thus the need to design new optimized techniques for the recovery of these messages is of great importance to forensic experts.

A lot of forensic work has been done on Android systems ranging from file system recovery to volatile memory acquisition. Unlike traditional computers systems, mobile devices have a different implementation of OS and file system internals across devices, making generic tool development much more difficult.

Android uses SQLite databases to store user data; hence recovery of the file system is guaranteed to recover undeleted messages. Deleted messages often times can be recovered through string search of the NAND dump, but more often than not, these messages might not laid out sequentially in the actual memory, but could be fragmented and very difficult to rearrange and comprehend. It might also be in a language the investigator doesn't

understand, making the string search technique not too effective.

Volatile memory acquisition and processing also poses the same issues with string searches, thus our approach is to develop a tool to correctly extract both deleted and undeleted messages. This involves first extracting the memory of the messaging process using **memfetch** (Zalewski, 2002), then running our **mcarve** application to carve out individual messages.

The reminder of this work is organized as follows: Section 2 describes related work in detail. Section 3 presents the methodology of our approach. Results and sample runs are shown in Section 4 and lastly section 5 presents conclusions.

2. Related Work

Both volatile and non-volatile storage media are a good source of digital evidence today. Like traditional computer systems, mobile devices utilize both volatile and non-volatile memory. Some limited work has been done to date on extraction of both volatile and non-volatile memory on Android devices. We briefly review these prior efforts in the remainder of this section.

2.1 Non-Volatile Memory Dump

Using the command line tool dd is the most widely used method of obtaining a non-volatile memory image. Most Android mobile devices use NAND flash memory for persistent storage and dd is used to produce a copy of this storage for forensic analysis.

The YAFFS2 filesystem is commonly used over NAND Flash memory instead of other more common filesystems. Due to its robustness and efficiency, it's used across a wide range of mobile devices today. In other to perform forensics examination, (Kessler, 2010) obtained a dd image of an Android mobile device and examined it with FTK. Though a lot of other important items were recovered, "FTK was unable to recover emails and chats". A search tool was then used, but even with that an examiner needs to have a an idea of what to search for, e.g., the email address of the suspect. But the question is how is it possible to know the email address or the social networking user id of a suspect without any additional information?

Hence we need some other techinques that will give us a wealth of digital evidence, no matter how little we know about a suspect.

2.2 Acquiring Volatile Memory

There are two ways of acquiring volatile data: hardware-based and software-based approaches. The hardware-based approach uses Direct Memory Access (DMA) to copy a memory image, beginning with halting the system's processor. The software approach, on the other hand, uses system tools like memdump or dd on Unix systems (Amari, 2009) and works by enumerating all physical pages and writing them out to a storage device while the system continues to operate.

Volatile memory on Unix systems was traditionally read via the /dev/mem or /proc/kcore devices, though not all ma-

chines support /proc/kcore. Because /dev/mem allowed both read and write access to kernel memory, it has been disabled on a lot of modern Linux distributions. The fmem tool, developed by Ivor Kollar (Kollar, 2010), creates a loadable kernel module that creates a /dev/fmem device supporting memory capture, but due to differences in the ARM and Intel x86 architecture, severe instability or even phone crashes might happen during acquisition (Sylve et al, 2011). Furthermore, it is essential to remember that although Android is built on a Linux kernel, it is not Linux (Brady, 2008).

2.3 Acquiring Android Volatile Memory

Lime, developed by Joe Sylve (Sylve et al, 2011), provides a sound forensic tool for volatile memory acquisition. It captures the memory image by:

i. Parsing the kernel iomem structure to get the physical address range of system RAM.
ii. Perform virtual to physical address translation for each memory area
iii. Reading all pages from RAM

As excellent as this method is, an investigator will be left with a large binary file with size of the system RAM (in most case ranging from 512-1GB). String search remains the only way to do application-level memory analysis on the file and is unlikely to be very effective.

As stated early, string search is not an effective way of digging through binary files, particularly when it comes to mobile forensics. Items to be searched, especially with respect to chats, might not necessarily be stored as plain text. More often than not, people chat in short phrases or native language making string search very complex.

Analysis of the memory dump acquired using the Lime forensic tool showed that messages are not arranged sequentially in physical memory. Allocation of physical memory is done on availability of free blocks, thus contiguous allocation is not guaranteed. This can lead to fragmentation within messages stored.

Since messages stored in physical memory might be fragmented and this makes understanding their meaning complicated, there is a need to rearrange broken parts to reconstruct full messages. At times, getting one message alone will not be sufficient; having the message thread will be needed to verify the evidence. Thus recreation of entire message threads is important.

Due to these issues outlined above, there is need for further research to create improved forensic tools that will ease the complicated nature of mobile device examination especially with regards to application-level message carving and reconstruction. Our approach was to design and develop **mcarve**, a forensic tool for extraction of messages and message threads from Android devices.

3. The "Per Process" Approach

This chapter discusses the methodology of this approach in detail.

3.1 Android Applications

Most Android applications are developed in Java, with some support for native C/C++ programs. Android also provides some libraries and native code that can be embedded or built with the help of the Android NDK. Each application in an Android system runs in a separate process in its own Dalvik instance; this ensures the security of processes. The Android Heap is the runtime memory of each Dalvik VM and contains all the objects created by the application.

A typical application has the extension "apk", and is basically a compressed file containing the following:

classes.dex: The Java program is compiled into in to dalvik executable file called "classes.dex". Android doesn't use Java bytecode.

Resource Files: This contains independent items like images and strings used within the app.

AndroidManifest.xml: Information about permissions (e.g., access to contacts, etc.) is set in this file. Every Android application must have this file.

3.2 Android Processes and Memory Management

Android memory management involves freeing objects from memory when they are no longer needed and assigning memory to processes that require it.

Android manages its physical memory on the basis of process priority. Android processes can be categorized into 5 priority levels:

i. Active Process: These are processes that run in the foreground and respond to user events or are at the present interacting with a user. Android reclaims memory from other process with lower priorities whenever active processes are in need of more RAM. Active process are last to be killed on the process priority queue.

ii. Visible Process: A process of this type is not in the foreground but can affect what the user is seeing on the screen. It does not directly interact with the user. Visible processes are only killed when those lower on priority are not available.

iii. Started Services process: This process started some non-active services that do not interact directly with the user.

iv. Background Process: This process is non-visible and at the same time has not started any service.

v. Empty Process: This process does not have any active component and the reason it is kept in memory is for caching purposes, to improve startup time. Empty processes have the lowest priority.

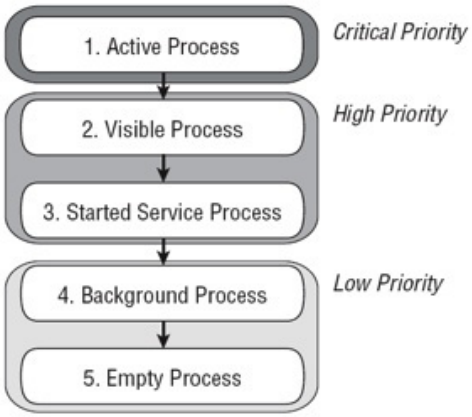

Figure 1. Android Memory Management

3.3 Motoblur

Motoblur is an Android UI replacement and push-based service focused on messaging (emails, SMS and social networking), developed by Motorola (Wikipedia). It creates a parent directory for messages and subsequent subdirectories for variety of widgets it supports. Older versions of Motorola Android phones require that a Motoblur account be set up before phone activation. Newer phones don't mandate the need for a blur account before phone activation but by default, they all come with pre-installed Motoblur services. Whenever a user opens the messaging folder, **"com.motorola.blur.service.main"** receives a signal to start up the EmailEngine and the SocialNetworking Engine. Upon startup, these engines load all the contents of snmessaging.db and email.db (which are sqlite databases) into memory. This is a reasonable decision because users typically access the messages often and loading the messages all the time from non-volatile memory would slow down the system. This content stays in memory until garbage collection, but because Android performs garbage collection on application-by-application basis and is based on the priority of a process, the data may stay in memory for an extended period of time. In general, because the Blur service has components that interact directly with user, it will have a high priority. Chat and emails are delivered through push operations and the Blur application calls the onReceive handler to get incoming messages. Even though the Blur service is not interacting with the user directly, activities such as getting a network connection will cause the onReceive handler to push new messages, which immediately notifies the user. The system will want to maintain the message data in memory because of the tendency for users to check messages upon notification.

3.4 Memfetch

Memfetch is an open source tool used on Linux-based systems to dump the memory region of a user-space process without disrupting its execution, either immediately or at the nearest fault condition (Zalewski, 2002). Memfetch reads the memory boundaries of a running process using /proc/pd/maps, then attaches to that process to PTRACE and copies the memory layout from /proc/pid/mem. In order not to disrupt the running process, memfetch raises the wait signal before reading process memory. Reading from /proc/pid/mem returns exactly how data is stored in the process' memory, thus increasing the accuracy of this approach, as we are guaranteed to output what is in memory at a particular time.

Memfetch outputs mfetch.lst and a number of mem and map files which repre-

sent anonymous maps (mostly shared libraries) and anonymous blocks respectively (stack, heap, code, etc).

For the purpose of this research, we are only interested in the heap of the running application, which by default is the second memory region (mem002.bin).

To enable memfetch to run on Android, we need to make some changes to the code:

i. Remove the include asm/page.h: This is because Android does not support page swapping by default. Whenever it needs to free memory, the whole process is swapped out of memory. Thus we need to declare some integer to hold the page size.
ii. We need to cross compile the code for the ARM architecture: Memfetch was originally created for Linux based Intel x86 systems. To use it on ARM systems, we need to statically cross compile it. This was done using the Codesourcery cross compilation toolkit (Sourcery CodeBench).

3.5 Message Carving

mcarve is our tool, developed to carve out the messages from the process heap. It is intended to further ease the forensic examination of Motorola Android phones. Rather than conducting examinations using string search, investigators can use mcarve to provide a simplified way of rapidly viewing complete messages and message threads. mcarve is designed to accept the heap dump file as an input then outputs two lists for emails and chats respectively.

In other to develop a standardized tool, we need to find out how the Motoblur app works. Since the Motoblur application is closed source, the only option for our purposes is to reverse engineer the application file (apk).

The Android Debug Bridge (adb) comes as a part of the standard Android SDK and provides a command line-based interface for interacting with the Android phone's file system.

To reverse engineer the apk, we first download and install the Android SDK, then execute the adb command to get access to the phone's shell. Then we do the following:

i. Pull the BlurSNMessagingEngine.apk and BlurEmailEngine.apk from /data/app.
ii. Rename the apk to zip file extension, enabling analysis of the classes.dex file.
iii. Download and run dex-2-jar, which will extract the classes.dex file and convert it to a jar file.
iv. Download jd-gui and execute with classes.jar as its argument. The output will be java source code.

We also downloaded and installed smali, which is an assembler/disassembler for the dex format. We run it against the classes.dex to get the assembly files output.

Using the Java code, the assembly code and the memory dump, we can now discover how messages are arranged in the heap.

3.5.1 SocialNetworking Carving and Reconstruction

The SNMessaging engine controls all the social networking widgets of the Motoblur application. At the receipt of intent from the user, it calls the SNMessagingSyncHandler. This is a class that implements SNMailAction. The Sync handler classes loads all messages from the snmessaging.db into the process' memory by calling the routine addSNMessage. It also loads all message threads in memory.

A social networking message thread indicates that communication has occurred at some time between two or more users; in this case the phone owner and other user(s). For every message thread, there is a unique remote id. Whenever a new message arrives from the same participant id, the system uses their remote id to continue the chat thread.

When the sync handler calls addSNThread, it causes the remote id, participant id, username and the thread status to be loaded in memory, in this order, for every chat participant in the database.

3.5.1.1 Algorithm

In order to carve out the chat messages, we need to first create a list of all participants. We use Python regular expressions to extract the thread information. This returns a list of strings all starting with some digits and then the user name. Because the digits of the remote id and participant id are concatenated and the lengths of these digits vary for all users, we decided to strip this string of the beginning digits and be left with only the usernames.

As stated earlier, addSNMessage loads the message table in memory, which includes the remote id, participant id, username and the message body. It distinguishes sent from received messages by concatenating the participant id and the phone owner's id, followed by the owner's username, then the message body.

Using the above list of usernames as the handle, we now extract all chats beginning with the names. We also trail back and extract the participant id. Now we end up with a list of messages beginning with ids, followed by username then message body. But all messages belonging to the phone owner will now have much longer ids (recall that for sent messages, the receiver's id is concatenated with the phone owner's id). Therefore in order to recreate message threads, we now compare the participant id at the beginning of each message; if that id exists then we put those messages in the same list. A collection holds a different list for each message thread.

3.5.2 Email Carving and Reconstruction

Like the SNMessaging engine, the email engine starts up after the Blur App receives intent from the user. The init() routine loads the content of message table email.db onto the heap. It lays down messages in a hash map, with each entry corresponding to a message. The

key represents message id; senders address and message body is concatenated into one string to form the value.

3.5.2.1 Algorithm

Figuring out email headers is not as complex as the social networking chats. The format of the mails starts with the mail ID which is the primary key in the message table of email.db. For both the two phones examined, the ID is 5-digit integer. Between the ID and the beginning of any email address, there is a single non-printable character, followed by a left angular bracket to denote the start of an email address (<).
With the above pattern, we develop a regular expression using Python to get the beginning index of each email starting with the ID, then the email address. This forms the mailList. Having the mailList as a handle, we now extract starting from each index of the mailList until we reach some non-printable characters of length at least 10 because there is no specific pattern to show the end of an email in memory. We now end up with a Mails List with each entry representing an email.

3.6 Behavior of Messages

Users with activated Motoblur accounts have their messages loaded unto memory as described above whenever the phone is switched on. New messages get appended to the hashmap as they come in, thus increasing the size of the process heap. When an email gets deleted, the email is moved from inbox to trash folder, the folder entry in email.db is changed from inbox to trash, but the memory is not affected.
Deleted emails get trapped in the trash forever until the trash is emptied. If the phone is rebooted, these deleted messages will be loaded into memory again. This is because their entries in the database have not been removed.
If the trash is emptied, the entries will be deleted from the database, but because the Android system doesn't support page swapping, the entry of that message in memory potentially remains unaffected until the phone is rebooted, when the new state of the database is loaded.

4. Experiments and Results

4.1 Experimental Setup

Two phones were used for this research; the Motorola Droid and Motorola flip-out.

The Droid was running Android 2.2 Froyo Model number A955, while the Flip-Out was running on Android 2.1 Update Éclair model number MB511. The Droid was rooted before messages were deleted, while for the flip-out phone, messages were deleted before rooting. This is to ascertain that the rooting did not change the state of the Motoblur app.

4.2 Testing

In this section we outline the steps to take for investigating a phone.

First the phone has to be rooted, Universal Androot (Shaka, 2010) is a good rooting kit and it doesn't disrupt most of the user processes. Also it doesn't require

you to reboot the phone so we are sure the memory isn't going to be wiped away. Unlike some rooting software like Z4Root, Universal Androot includes the su binary (superuser.apk), and it works well with Motorola phones.

The phone should then be connected via USB, ensuring that USB debugging is turned on and the phone is set to charge only.

A compiled memfetch is copied into the phone and data is saved in /data/local.

We edited memfetch to call grep blur.service.main on ps output and return the necessary pid. The pid is provided to memfetch as an argument. Execution of memfetch will return the mfetch.lst, and some *.mem and *.map files.

The file mem.002.bin is then copied from the phone and mcarve is run on it. This will output mail and chat collections from the phone. .

Email Sample Output

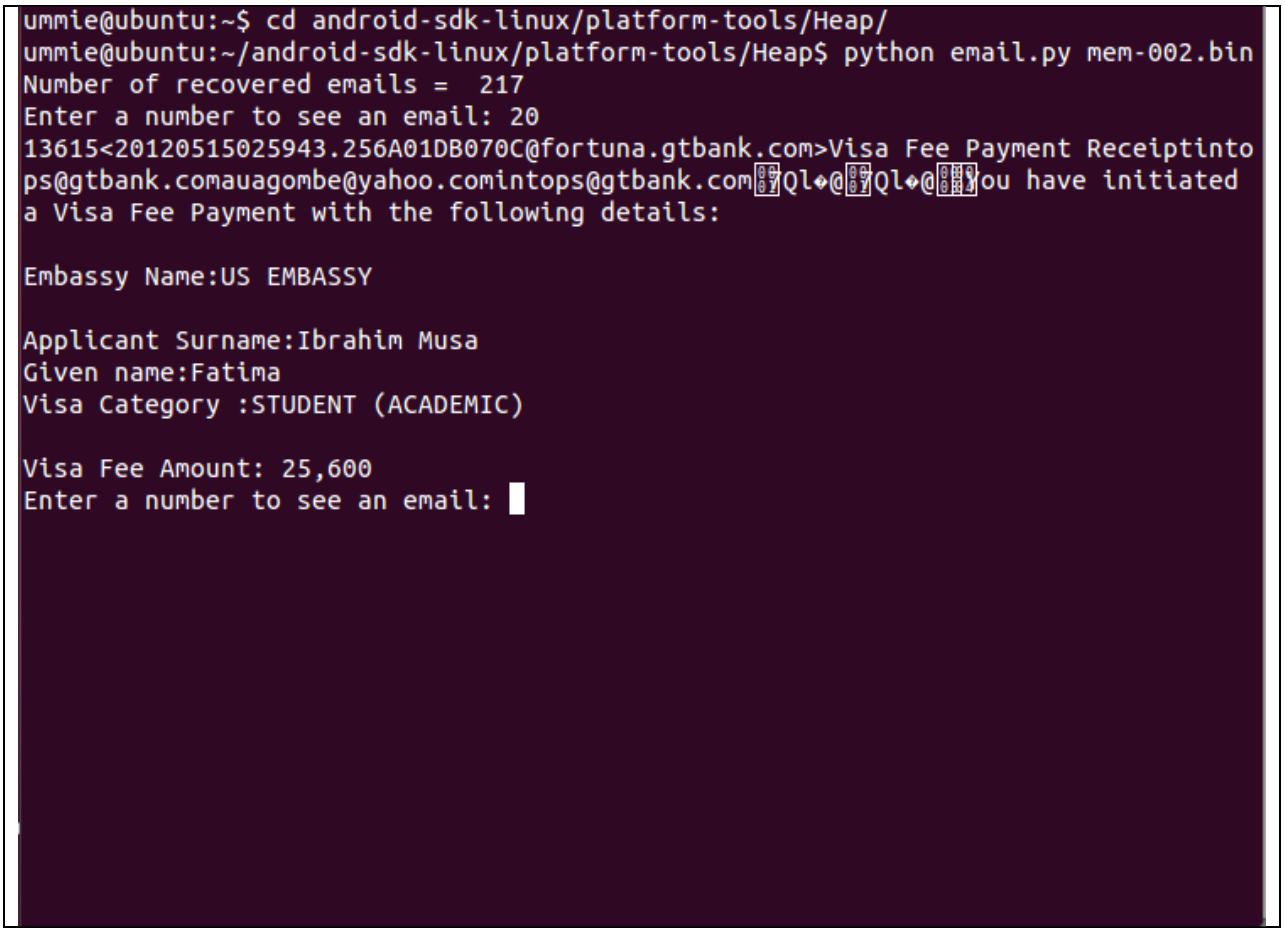

Figure 2. Sample output of email.py.

Chat Sample Output

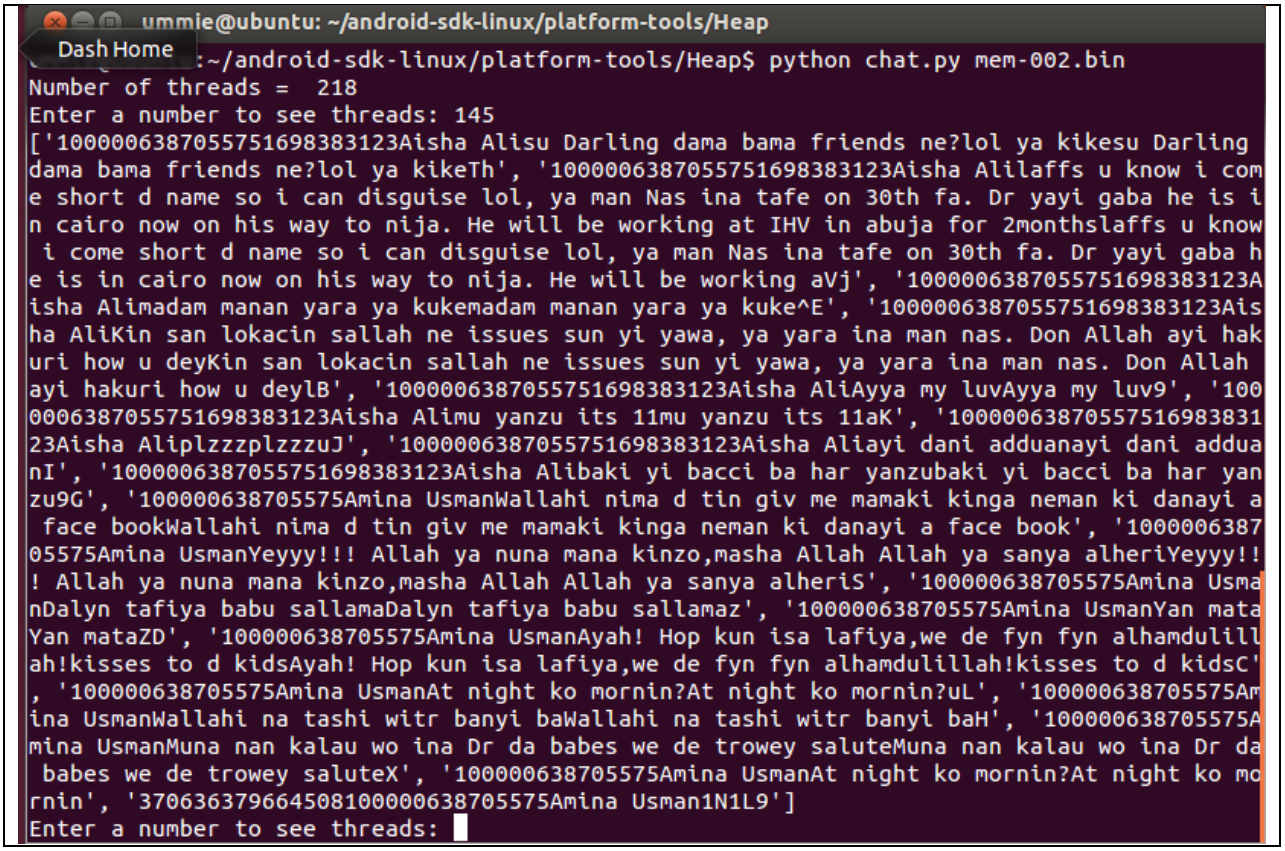

Figure 3: Sample Output of chat.py.

4.3 Sample Run Analysis

	Droid	Flip-Out
Total Emails	198	184
Total Chats	703	694
Total Threads	180	172
Deleted Emails	6	9
Deleted Chats Threads	5	5
Total Recovered Emails	217	202
Total Recovered Threads	218	211
Total Recovered Chats	912	901

Figure 4. Sample run result.

Out of 192 undeleted emails on the Droid phone, all were recovered.

Out of 6 deleted emails on the Droid phone, all were recovered.

Out of 175 undeleted Emails on the flip-out phone, all were recovered.

Out of 9 deleted emails on the flip-out phone, all were recovered.

4.4 Research Findings

Data recovery from the process heap is not only possible, but recovers information in a format as laid out by a particular application source code in its runtime memory. This is especially useful in for processes that load data from databases.

Due to the process priority's role in Android's garbage collection scheme, active processes tend to stay longer in memory. As long as the phone has not been powered down, data in the process heap for the Blur application is maintained and both deleted and undeleted messages can be recovered.

5. Conclusion

This thesis presented a "per process" approach for application-level live forensic analysis, a method we used in capturing the volatile memory of one of the most important applications on Motorola Android phones. It detailed how the runtime memory can be extracted while the process is still running without disrupting its execution, as well as providing the mcarve tool which extracts valuable information (mails and chats) stored and transferred using the Motoblur app.

The MotoBlur app not only controls mails and chats but also transmits and stores friend's feeds and blur contacts. Future research can help in improving this tool by embedding patterns to extract the friends status updates as well as all blur contacts, not limiting to those that have established message threads.

Because the Blur app is limited to only Motorola phones, future research is needed to determine how other Android phones on the market handle email and chats and how best we can find ways of easing forensic investigation on them.

References

Memory Management in Android. (2010). Retrieved from http://mobworld.wordpress.com/2010/07/05/memory-management-in-android/

Amari, K. (2009). *Techniques and Tools for Recovering and Analyzing Data from Volatile Memory.*

Brady, P. (2008). *Anatomy and Physiology of an Android.* Retrieved from Google I/O Session Videos abd Slides: https://sites.google.com/site/io/anatomy-physiology-of-an-android

Canon, T. (2010). *Android Reverse Engineering*. Retrieved from http://thomascannon.net/projects/android-reversing/

Case A, et. al. (2010). Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–40.

Freke, J. *smali/baksmali* . Retrieved 2012, from Google Code: http://code.google.com/p/smali/

Sylve J, et al. Acquisition and analysis of volatile memory from android devices. *DFRWS*, 2011.

Lesser J. & Kessler G. C. (2010). Android Forensics: Simplifying Cell Phone Examinations. *Small Scale Digital Device Forensics*, ISSN# 1941-6164.

Kollar, I. Retrieved 2010, from http://hysteria.sk/wniekt0/foriana/fmem_current.tgz

Kollar, I. (2010a). *Forensic RAM dump image analyser.* Prague: Departmemt of Software Engineering, Charles University.

Panxiaobo. Retrieved 2012, from http://code.google.com/p/dex2jar/

Shaka, H. (2010, August 30). *Universal Androot 1.6.2 beta 5*. Retrieved Sepetember 2012, from http://blog.23corner.com/tag/universalandroot/

Sourcery CodeBench. Retrieved June 01, 2012, from Mentor Graphics: http://www.mentor.com/embedded-software/sourcery-tools/sourcerycodebench/editions/lite-edition/;

T. Vidas, et. al. (2011). Toward a general collection methodology for Android devices. *Digital Investigation*, S14-24.

Thinkfeed. Retrieved 2011, from http://code.google.com/p/innlab/

VLL Thing, et. al. (2010). Live memory forensics of mobile phones. *DFRWS*.

Zalewski, M. (2002) Retrieved 2012, from http://lcamtuf.coredump.cx/soft/memfetch. tgz

VITA

The author Aisha Ali-Gombe was born in Zaria, Kaduna State Nigeria. She obtained her Bachelor's degree in computer science from University of Abuja, Nigeria in 2005 and an MBA from Bayero University Kano in 2011. She then joined the University of New Orleans computer science graduate program to pursue an MS degree in the same field with concentration in information assurance. This research work was done under the supervision of Professor Golden Richard III in 2012.