Smartphone Volatile Memory Acquisition for Security Analysis and Forensics Investigation

Vrizlynn L.L. Thing and Zheng-Leong Chua

Cybercrime & Security Intelligence (CSI) Department
Institute for Infocomm Research, Singapore
vriz@i2r.a-star.edu.sg

Abstract. In this paper, we first identify the need to be equipped with the capability to perform raw volatile memory data acquisition from live smartphones. We then investigate and discuss the potential of different approaches to achieve this task on Symbian smartphones. Based on our initial analysis, we propose a simple, flexible and portable approach which can have a full-coverage view of the memory space, to acquire the raw volatile memory data from commercial Symbian smartphones. We develop the tool to conduct the proof-of-concept experiments on the phones, and are able to acquire the volatile memory data successfully. A discussion on the problems we have encountered, the solutions we have proposed and the observations we have made in this research is provided. With the acquired data, we conduct an analysis on the memory images of the identified memory regions of interest, and propose a methodology for the purpose of in-depth malware security and forensics analysis.

Keywords: Symbian, mobile devices, smartphones, volatile memory data acquisition, malware security and forensics analysis.

1 Introduction

Mobile phones are becoming increasingly prevalent and sophisticated. They are continuously evolving into “smarter” devices (i.e. smartphones with higher processing power and enhanced features) to cater to the needs of users to stay connected anytime, anywhere, with information readily available. Due to the connection and processing capability of smartphones, illegal access to a wealth of information (for example, contacts list, emails, messages, downloaded confidential documents from email attachments) belonging to the users can be acquired from their smartphones with the appropriate technologies (for example, information theft malwares [1], or mobile forensics tools). Therefore, the capabilities to perform in-depth security analysis to prevent and detect attacks, and forensics investigation to acquire evidence from these devices are essential.

Current mobile phone forensics tools are restricted to the acquisition and analysis of active files and data (i.e. logical data acquisition) from the Subscriber Identity Module (SIM), memory cards and the internal flash memory [2–8]. There exists research work focusing on the low-level physical accquisition

L.J. Janczewski, H.B. Wolfe, and S. Shenoi (Eds.): SEC 2013, IFIP AICT 405, pp. 217–230, 2013.
© IFIP International Federation for Information Processing 2013

of raw data from the mobile phones' non-volatile memories [9–12], to support in-depth forensics investigations and evidence analysis, but did not take the volatile memories into consideration.

On the other hand, to support smartphones volatile application data acquisition, loaded malware driver detection, malware behaviour and analysis, it is necessary to have the capability to perform raw volatile memory data acquisition from mobile devices. The ability to acquire the raw volatile memory data from a live device provides security analysts and forensics investigators with a complete picture and insight of the operational states of the live device. However, current anti-virus and anti-malware tools [1, 13–16] for smartphones are limited to the scanning of programs and files in the non-volatile storage space to carry out signature based virus detection.

In this paper, we propose a method to acquire raw volatile memory data from live Symbian smartphones and the methodology to analyse the acquired data to facilitate security analysis and forensics investigations. We develop the tools to conduct the proof-of-concept experiments on commercial Symbian smartphones. There are two main reasons for the choice on the Symbian OS in this research:

1. Even as Android and iOS is rising fast to become the most popular mobile OSes, with Android holding a market share of 52.5%, according to a mobile OS market share survey by Gartner [17, 18], Symbian still holds 16.9% of the market share and is the second most widely used OS in mobile phones. In addition, based on the statistics provided by StatCounter [19], Symbian is observed to be the top smartphones used for the purpose of mobile web browsing.
2. Nonetheless, little has been done on the research of Symbian smartphones live memory security and forensics analysis yet. The reason is that most modern mobile OSes (including Symbian), like generic computer system OSes, use a layer of abstraction such as the virtual memory instead of operating directly on the physical memory. This abstraction layer provides the ability to sandbox each process into its own memory space for security protection. Therefore, with this memory protection in place, a raw volatile memory acquisition tool would have to reside in the kernel space to gain access to the entire memory space [20, 21]. However, in Android, a process memory is exposed to the user-side through the procfs filesystem and this mechanism can be utilized to achieve the live volatile memory acquisition from smartphones running Android OS [22]. On the other hand, Symbian OS does not provide any such mechanism to be leveraged on. As such, it is necessary to investigate and devise a method for conducting a live volatile memory data acquisition pertaining to Symbian smartphones.

The rest of the paper is organised as follow. In Section 2, we present an overview of the existing work on mobile phone forensics research. In Section 3, we present the historical account of mobile forensics research specific to the Symbian smartphones. In Section 4, we present our investigation on the potential approaches to achieve a raw volatile memory data acquisition from commercial Symbian smartphones. We describe the design and the implementation of our live Symbian volatile

memory acquisition tool, the problems we encountered and the solutions we devised, in Section 5. The experiments are described in Section 6. We also proposed the analysis methodology in Section 6. Conclusions follow in Section 7.

2 Mobile Phone Forensics Research

In this section, we present an overview of the existing mobile forensics work in general.

In an early work in 2003, Willassen [2] researched on the forensic investigation of GSM phones. The author presented the types of data of forensic relevance, which can exist on the phones, the SIM and the core network, and emphasized the need for more sound mobile forensics procedures and tools.

In 2006, Willassen [9] proposed extracting the physical image of the mobile phone's internal flash memory by desoldering the memory chip and reading it from a device programmer. However, this method is too invasive and brings with it a high risk of chip damage if the extraction is not performed with high precision and care. Another proposed method was to read the memory through the boundary-scan (JTAG) test pins. The extracted memory was examined to detect the presence of deleted file contents. However, the test pins are usually not prominently shown and labelled. In this case, attempting to identify them may be very challenging and time consuming. There is also the possibility that these pins on the commercial smartphones are not accessible to users.

In the same year (2006), Casadei et al. [3] presented their SIMbrush tool developed for both the Linux and Windows platforms. The tool relied on the PCSC library and supported the acquisition of the entire file system, including the non standard files, on the SIM. However, files with restricted read access conditions could not be extracted.

In 2007, Kim et al. [4] presented a tool to acquire the data from a Korea CDMA mobile phone's internal flash memory. The tool communicated with the phone through the RS-232C serial interface and was able to acquire the existing files on the phone using the underlying Qualcomm Mobile Station Modem diagnostic mode protocol.

In the same year (2007), Al-Zarouni [10] studied the mobile phone flasher devices and considered their applicability in mobile phone forensics. Flasher devices were originally used to perform SIM unlocking and firmware flashing. Therefore, they offered access to the phone's flash memory. As they did not need installation on the phone, they were deemed to be forensically sound. However, their operations were not well-documented and since they were designed to write to the memory, the effect of evidence altering while performing a read was also unknown. Their reading capability and memory access range also varied for phones of different brands and models.

In 2008, Jansen et al. [7] proposed a phone manager protocol filtering technique by intercepting the data between the phone and the phone manager. The objective was to address the latency in the coverage of newly available phone models by existing forensic tools. The authors also proposed an identity module

programming technique, to populate the phone's SIM with reference test data, so as to provide a baseline for the validation of SIM forensic tools.

In 2008, Zdziarski [23] published a book on iPhone forensics which contains information on how to conduct forensic analysis of iPhone, iPhone 3G, and iPod Touch. The book covers information on the type of data that can be stored on an iPhone, the procedure to build a custom recovery toolkit, the recovery of the raw user disk partition, and the application of data carving techniques to recover deleted voicemail, images, emails, etc. from the phone.

In 2009, Hoog [8] presented the existing forensic evidence acquisition tools for Android phones. The Android Debug Bridge (ADB) enabled interaction with the phone over the USB connection. Therefore, active files on the phone can be retrieved through the "adb pull" command. Other tools such as the Nandroid backup and Paraben Device Seizure also supported the extraction of files residing on the phone.

In 2010, Thing et al. [22] proposed a method to acquire live volatile memory data from Android smartphones. In Android, the process memory is exposed to the user-side through the procfs filesystem. The authors proposed utilizing the process tracing (ptrace) system call to suspend the target process, acquire a snapshot of its memory, and then resume its execution. A study and analysis of the dynamic characteristics of volatile data in the process memory was then carried out.

In 2011, Hoog and Strzempka [24] published a book on iPhone and iOS forensics. The book covers information on the techniques to acquire evidentiary data from the iPhone, iPad and other iOS devices. It also provides practical advise on the securing of the devices, and the data and applications residing on them.

3 History of Symbian Smartphone Forensics

In 2007, Mokhonoana and Olivier [5] proposed an on-phone forensic tool to acquire the active files from a Symbian OS version 7 phone and store it on the removable media. Instead of interfacing with the PC connectivity services, the tool interacts with the operating system to perform a logical copy of the files. Experiments were conducted on the Sony Ericcson P800 phone. The main limitation of the tool is that files in use could not be copied (e.g. call logs, contacts).

In 2008, Distefano et al. [6] proposed a mobile phone internal acquisition technique on the Symbian OS version 8 phones. The mobile phone data is acquired using a tool residing on the removable media, instead of the PC/mobile phone USB connection based approach. The tool utilizes the Symbian S60 File Server API in the read-only mode. The authors carried out experiments comparing the tool with Paraben Device Seizure (USB connection to phone) [25] and P3nfs (Remote access through Bluetooth) [26]. The tool took a longer time to perform the acquisition but managed to acquire more data compared to the P3nfs. When compared with the Paraben Device Seizure, lesser data was acquired. However, the authors observed that the larger data size from Paraben was due to the additional information from its acquired data management.

In 2012, Thing and Tan [11] proposed a method to acquire privacy-protected data from Symbian OS version 9.3 and version 9.4 phones. The authors bypass the Symbian platform security (introduced from Symbian OS version 9.1) to obtain an unrestricted read access to the entire filesystem on the phone. Based on the obtained privilege, the authors retrieve the files relevant to SMS messages from the Nokia E72 and N97 phones (running Symbian OS version 9.3 and version 9.4, respectively). Reverse-engineering work is then carried out to derive the various SMS storage formats on the files and to recover both active and deleted SMSes previously stored on the phones' flash memory.

In the same year (2012), Thing and Chua [12] proposed a low-level linear bitwise data acquisition technique for the Symbian OS version 9.4 phones to support evidentiary file carving. A study and analysis of how files are stored and fragmented on the Symbian smartphone flash memory was also carried out.

However, there is no existing work on the live volatile memory data acquisition pertaining to Symbian smartphones. To the best of our knowledge, this is the first work that aims to investigate and devise a method for conducting a live volatile memory data acquisition from Symbian smartphones.

4 Investigation of Potential Live Symbian Volatile Memory Data Acquisition Approaches

In Symbian, debugging APIs are provided for its programmers. They are typically used during Symbian application development to debug and investigate program efficiency problems. We investigate the potential usefulness of these APIs in live raw volatile memory data acquisition from the commercial Symbian smartphones.

4.1 Run-Mode Debugging

The run-mode debugging APIs are applicable in accessing the process memory of a running application. This type of debuggers is target-resident based and focus primarily in debugging applications and middleware. A process or thread identifier is required in order to access a relevant target specific memory region. However, they are restricted in capability and do not have a deep insight in the kernel-mode software and device drivers. For malwares and rootkits to obtain privileged functionalities, they must be able to execute in the supervisor mode and the way to achieve this is through the use of drivers or kernel modules. Therefore, the acquisition tool must be able to obtain a deep insight into the kernel space modules. In addition, memory not committed to any process at the acquisition point in time will not be accessible.

4.2 Stop-Mode Debugging

The stop-mode debugging APIs are able to "freeze" the processes on the device and acquire a snapshot of its current memory state. However, they are required

to be hardware-assisted by utilizing (or tapping onto) the JTAG boundary-scan test pins [9, 27, 28] on the device. This hardware based approach works by accessing the debug ports used by the embedded device processors. JTAG is then switched between the extest or debug mode so as to produce an image dump of the memory. However, the test pins are usually not prominently shown and labelled. In this case, attempting to identify them is known to be very challenging and time consuming. On the commercial smartphones, these pins are commonly removed to prevent access by users.

4.3 Kernel Module

Instead of relying on the existing Symbian drivers or APIs, we can instead write our own kernel space driver for the purpose of memory acquisition. The driver can be a logical device driver that utilizes a logical channel to interact with the user-side application. The driver can then receive specific arguments such as the address of the memory to acquire, and send the acquired data to the user-side application to be stored on the non-volatile memory or to be transmitted out from the phone through a network connection such as 3G or Wifi. Upon receiving the memory address, we intend to perform direct de-referencing of the address to obtain the corresponding memory data and pass it back to the user-side calling application. The idea of the direct de-referencing of the address method is simple. It also provides a flexible way of accessing the memory on the device. Since it utilizes the basic de-reference operator, it should theoretically work across Symbian smartphones with different CPU architectures and/or memory models since the kernel shields us from the effect of these different implementations.

From the absence of relevant prior art, we observed that the live volatile memory data acquisition from Symbian smartphones remains a challenging task. However, we have identified a potential Symbian kernel space driver approach which could achieve a successful acquisition. In this paper, we propose a methodology and discuss the challenges we faced when devising the tool to perform the live volatile memory data acquisition. We also look into the security analysis and forensics investigation of Symbian smartphones based on the acquired data and discuss the observations we have made.

5 The Proposed Acquisition Approach

The live volatile memory data acquisition tool that we have designed, composes of two parts. It consists of a user-side component and a kernel-side component. The user-side component is responsible for loading the kernel-side component, initializing the client side of the logical channel (to support the subsequent communication with the kernel-side component during operation), passing in the addresses to be de-referenced and storing the returned data onto the non-volatile memory. The kernel-side component is a logical device driver which is responsible for setting up the logical channel with the user-side component to support communication during operation, and to return the acquired data from the de-referenced address (which is passed in by the user-side component). We named

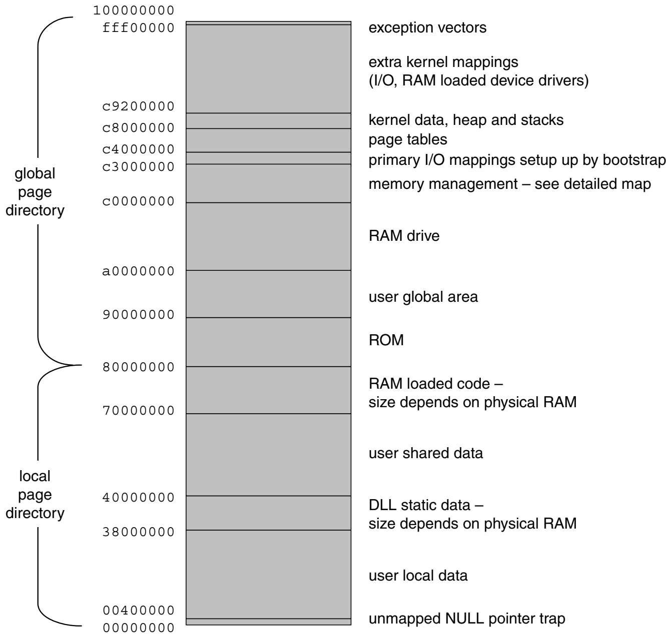

Fig. 1. Symbian Multiple Model Memory Map

the kernel-side component the "Live Volatile Memory Data Acquisiton" driver or "Lamda", and referred to the user-side component as the "LamdaLoader".

Figure 1 shows the Symbian multiple model memory map [20]. With the multiple memory model, Symbian has the concept of a local page directory and a global page directory. The memory region from 0x00000000 to 0x7FFFFFFF is translated via the local page directory while the memory region from 0x80000000 to 0xFFFFFFFF is translated via the global page directory. The global page directory memory regions is accessible by any process while the local page directory memory region is restricted to the current process.

The regions of memory that we are interested in is the global page directory region. Specifically, we are interested in the memory address region from 0xC9200000 to 0xFFEFFFFF (that is, the extra kernel mappings for the I/O and RAM loaded device drivers) and from 0xC8000000 to 0xC91FFFFF (that is, the kernel data, heap and stacks, which contain the kernel objects for the loaded drivers).

Even though we have identified the regions of the memory to perform acquisition, we tested our tool by trying to acquire the entire memory region, starting from the address 0x00000000, from a commercial Nokia N97 smartphone running Symbian OS version 9.4, S60 5th Edition. Upon execution, the smartphone returns a KERN-EXEC 3 panic result and causes it to reboot. Referring to the memory map, we noticed that the address that we sent in is in the region of unmapped NULL pointer traps. The KERN-EXEC 3 panic is caused by an

untrapped page fault and since the offending process is a kernel thread, it causes the smartphone to reboot.

The encountered page fault during this initial execution matches the documented behaviour that the kernel does not support on-demand paging [20]. However, upon investigation of the memory management unit (MMU) code in the Symbian product development kit, we realized that demand paging is actually implemented. This information leads us to believe that the KERN-EXEC 3 panic was not caused solely by a page fault but by other underlying mechanisms. We refer to the documentation provided by the product development kit which describes the impacts of demand page on kernel-side code. The document reveals that demand paging is not implemented for kernel code and data. Therefore, any page fault experienced in the kernel code and data will result in an unhandled fault. We decide to modify the tool to handle the page fault gracefully.

In Symbian, the exception handling and trapping mechanism is provided in the form of the XTRAP/XTRAPD macros (to enable the exception trapping). The difference between the two macros is that the XTRAPD macro declares the result variable whereas the XTRAP macro uses a pre-existing variable. They behave in a similar way as the user-side TRAP/leave(), but instead, they can catch hardware exceptions such as those generated by a faulty memory access. In addition, we utilize the TPhysAddr Epoc::LinearToPhysical(TLinAddr aLinAddr) function in Lamda to return the physical address corresponding to the virtual address passed in by the LamdaLoader. We then check the returned value before de-referencing the virtual address as an additional safety net. If KPhysAddrInvalid (defined in kernel/kern_priv.h) is returned, it indicates that the specific virtual address is unmapped. If the returned value is not KPhysAddrInvalid, we proceed to carry out address de-referencing. It is also important to note that even when a virtual address is mapped to a valid physical address, there are instances when the CPU is not permitted to access a page as it does not satisfy the access policy for the page currently. This could also result in a fault. Hence, the exception trapping is useful in this scenario even with the valid mapping check function in place.

6 Experiments and Analysis

In this section, we use our tool to perform the acquisition of the ROM shadow region (0x80000000 to 0x8FFFFFFF) for verification purpose, and the extra kernel mapping (I/O, RAM loaded device drivers) region (0xC9200000 to 0xFFEFFFFF) and the kernel data, heap and stacks region (0xC8000000 to 0xC91FFFFF) of the N97 smartphone's volatile memory for further analysis.

6.1 ROM Shadow Image

We acquired the ROM shadow images from three different experiments; one from a N97 smartphone, another from the same smartphone after a hard-reset was performed, and the third from another N97 smartphone with the same version

of the ROM flash image. We performed a bitwise comparison and verified that the acquired memory images are identical and therefore, proved that they indeed contain the same version of the ROM image. Next, we analyse the image to identify an approach to facilitate integrity checks of the drivers provided in the ROM flash image on the smartphones originating from the manufactors.

Symbian drivers are binary files and should contain certain specific header information. In Symbian, its E32Image files contain a 12-byte UID (unique identifier and composes of a set of UID1, UID2 and UID3) data that indicate the file type and identify the particular file object. A 4-byte UID1 value of 0x1000007A indicates that the file is of an executable type. Therefore, a logical driver has a UID1 value of 0x1000007A as it is a DLL executable file. In addition, it has a 4-byte UID2 value of 0x100000AF to indicate that it is a logical device driver (LDD). The 4-byte UID3 value is used to identify a particular object (for example, a particular executable file). As the data is stored in little-endian format, the data pattern for searching and extracting the device driver binaries from the ROM memory image is (79 00 00 10 AF 00 00 10). On further analysis, we also discovered that the binary header structure of the detected driver within this memory image is that of the TRomImageHeader structure, as shown in Fig. 2.

```
mem.img  mem.img  mem.img
Offset     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
000673616  9C 46 0A 80 44 3B 0A 80 68 3B 0A 80 00 00 00 00 .F..D;..h;......
000673632  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000673648  00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ................
000673664  00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 ................
000673680  88 EF 04 80 FF FF FF FF FF FF FF FF FF FF FF FF ................
000673696  79 00 00 10 AF 00 00 10 B9 23 28 10 19 32 BD 06 y........#(..2..
000673712  18 48 0A 80 18 48 0A 80 00 00 00 00 6C 03 00 00 .H...H......l...
000673728  6C 03 00 00 00 00 00 00 00 00 00 00 00 10 00 00 l...............
000673744  00 00 10 00 00 20 00 00 84 4B 0A 80 03 00 00 00 ..... ...K......
000673760  DC 49 0A 80 B9 23 28 10 01 00 00 70 FF FF 0F 00 .I...#(....p....
000673776  00 00 00 00 02 01 54 02 2B 01 80 13 5E 01 00 00 ......T.+...^...
000673792  E0 70 09 C8 00 00 00 00 04 00 08 09 00 00 00 00 .p..............
000673808  00 00 0A 00 00 00 00 00 00 00 1F E3 04 C0 9F E5 ................
000673824  1C FF 2F E1 00 00 00 00 88 49 0A 80 38 40 2D E9 ../......I..8@-.
000673840  20 40 9F E5 20 50 9F E5 05 00 54 E1 38 80 BD 08  @.. P....T.8...
000673856  00 00 94 E5 30 FF 2F E1 04 40 84 E2 05 00 54 E1 ....0./..@....T.
000673872  FA FF FF 1A 38 80 BD E8 00 00 00 00 00 00 00 00 ....8...........
000673888  00 00 A0 E3 1E FF 2F E1 10 40 2D E9 73 00 00 EB ....../..@-.s...
000673904  F0 10 9F E5 00 10 80 E5 00 10 A0 E3 78 10 80 E5 ............x...
000673920  24 10 90 E5 05 10 81 E3 24 10 80 E5 10 80 BD E8 $.......$.......
000673936  10 40 2D E9 80 00 A0 E3 56 00 00 EB 00 00 50 E3 .@-.....V.....P.
000673952  00 00 00 0A EF FF FF EB 00 40 B0 E1 05 00 00 1A .........@......
000673968  B4 00 9F E5 90 10 A0 E3 4C 00 00 EB 02 10 A0 E3 ........L.......
```

Fig. 2. TRomImageHeader

6.2 Kernel Data, Heap, Stacks

In this region of the memory, we are able to detect the presence of the loaded executables and libraries as their paths are clearly shown in this memory region. Therefore, known malware executables, loaded dynamic libraries and driver names can be easily detected by performing a simple search. As shown in Fig. 3, we observed that the path for the loaded sisadriver is at offset 0x855890.

```
mem.img   mem.img
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
00855840  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00855856  00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 ................
00855872  00 00 00 00 00 00 00 00 00 00 00 00 28 00 00 00 ............(...
00855888  19 00 00 30 19 00 00 00 5A 3A 5C 73 79 73 5C 62 ...0....Z:\sys\b
00855904  69 6E 5C 73 69 73 61 64 72 69 76 65 72 2E 6C 64 in\sisadriver.ld
00855920  64 00 00 00 20 00 00 00 0D 00 00 30 0D 00 00 00 d... ......0....
00855936  53 74 61 72 74 65 72 53 65 72 76 65 72 00 00 00 StarterServer...
00855952  00 00 00 00 28 00 00 00 0D 00 00 30 0D 00 00 00 ....(......0....
00855968  53 74 61 72 74 65 72 53 65 72 76 65 72 00 00 00 StarterServer...
00855984  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00 ............@...
00856000  3C C1 08 80 00 00 00 00 01 00 00 00 F8 CB 13 C8 <...............
00856016  05 00 00 00 00 00 00 00 00 00 E4 07 00 00 00 00 ................
00856032  00 00 00 00 00 00 00 00 E8 0F 0D C8 E8 0F 0D C8 ................
00856048  00 00 00 00 00 00 00 00 00 00 00 00 38 00 00 00 ............8...
00856064  B6 75 1F 10 B5 4B 20 10 00 00 00 00 00 00 00 00 .u...K .........
00856080  00 00 00 40 00 00 00 00 20 00 00 40 04 2C 27 10 ...@.... ..@.,'.
00856096  01 00 00 00 18 D6 18 C8 00 00 00 00 2C 10 0D C8 ............,...
00856112  2C 10 0D C8 18 00 00 00 0B 00 00 30 0B 00 00 00 ,..........0....
00856128  72 61 6E 64 73 76 72 2E 65 78 65 00 30 00 00 00 randsvr.exe.0...
00856144  30 CB 08 80 00 00 00 00 03 00 00 00 08 00 00 00 0...............
00856160  02 00 00 00 02 00 00 00 28 3E 1D C8 78 2F 1D C8 ........(>..x/..
00856176  02 00 00 00 04 00 00 00 00 00 00 00 18 00 00 00 ................
00856192  05 00 00 30 05 00 00 00 24 48 45 41 50 00 00 00 ...0....$HEAP...
```

Fig. 3. LDD Path in Kernel Data

We also found the path of the LamdaLoader at offset 0x1903456, the path to AknIconSrv (which is a system application loaded from the ROM) at 0x1920464 and the path of the nokiaiscdriver at 0x856592.

6.3 Extra Kernel Mappings

The extra kernel mappings region is the memory region where the RAM loaded drivers are residing in. Therefore, it would be a very important region of interest for conducting malware security analysis and forensics investigations. We designed an experiment to ensure that no other RAM loaded driver is present. After which, we load an additional driver other than Lamda and acquire this memory region. Therefore, we expect that the resultant memory image would contain these two drivers only. We use the similar technique in the ROM memory image analysis to search for presence of loaded drivers based on the UID string pattern (79 00 00 10 AF 00 00 10). Consistent with our expectation, only the two drivers were detected to be present.

We observed that the RAM loaded drivers have the E32ImageHeader (Fig. 4) rather than the TRomImageHeader header structure found in the ROM drivers. The difference is due to the way these two types of drivers were compiled differently in Symbian. Unlike the previous case of ROM drivers integrity check, where simple matching can be performed, the actual extraction of the RAM drivers has to be conducted to facilitate further security and forensics analysis. Therefore, to do so, we have to first determine the size of the driver.

Our first attempt is to use the TUint32 iUncompressedSize field in the E32ImageHeaderComp. However, this field could not be utilized as the loaded driver found in the RAM is in a compressed form and the size is (expectedly) different from its provided uncompressed driver size. We attempted to calculate the

```
mem.img
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF
124788928 8D 00 00 00 01 00 00 00 00 00 00 00 6E 00 00 00 ............n...
124788944 07 00 00 00 44 72 69 76 65 72 31 00 00 00 00 00 ....Driver1.....
124788960 07 00 00 00 44 72 69 76 65 72 31 00 14 DF FF 7F ....Driver1.....
124788976 01 00 00 00 01 00 00 00 78 00 90 D0 F8 00 60 00 ........x.....`.
124788992 0C 00 00 00 00 00 00 00 03 03 03 03 B0 08 00 00 ................
124789008 A3 08 00 00 79 00 00 10 AF 00 00 10 AF 17 8F E5 ....y...........
124789024 C1 5E 56 70 45 50 4F 43 ED EC B7 38 00 00 0A 00 .^VpEPOC...8....
124789040 FC 7A 1F 10 02 00 02 02 40 E5 2B DB 10 95 E1 00 .z......@.+.....
124789056 2B 00 00 12 60 14 00 00 00 00 00 00 00 10 00 00 +...`...........
124789072 00 00 10 00 00 20 00 00 00 00 00 00 0C 0A 00 00 ..... ..........
124789088 00 80 00 00 00 00 00 00 01 00 00 00 F8 14 00 00 ................
124789104 01 00 00 00 60 14 00 00 9C 00 00 00 00 00 00 00 ....`...........
124789120 FC 14 00 00 E0 15 00 00 00 00 00 00 5E 01 01 20 ............^.. 
124789136 9C 15 00 00 AF 17 8F E5 00 00 00 00 FF FF 0F 00 ................
124789152 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
124789168 DE BB 9E BE 37 E6 9F 39 97 39 57 BE 1D 6E F3 30 ....7..9.9W..n.0
124789184 CC D9 78 5B 7A 6F 33 79 94 5E 1C 1B CA 5E 93 37 ..x[zo3y.^...^.7
124789200 98 29 6C CD 4B 97 5D A5 CB 1D E3 46 B9 5B AD 5B .)l.K.]....F.[.[
124789216 45 AD 2D 72 F3 4B CE 63 39 B5 4C CB 4B 2A 83 A3 E.-r.K.c9.L.K*..
124789232 5D BA D5 CC 30 C7 04 2D 33 12 B6 A4 CB 5C B5 D4 ]...0..-3....\..
124789248 CD B5 AA 15 CE 2E 75 B5 9B 41 B1 42 A8 DC F7 E5 ......u..A.B....
124789264 F9 6E F7 B6 F4 D5 7B F4 3E AC 1E E9 F3 BE 66 AF .n....{.>.....f.
124789280 7E 79 89 4A 85 6A B1 54 56 5E 73 65 33 47 CB 29 ~y.J.j.TV^se3G.)
```

Fig. 4. UID String with E32ImageHeader Signature 'EPOC'

size by utilizing the fields such as the iCodeSize and iTextSize, but as expected, these attempts proved futile as these offsets are based on the uncompressed code.

We hypothesize that the compressed driver size information must exist within the header or within range of the located driver as the Loader Server has to be able to determine the size of the driver in order to decompress or load it. Through further analysis on a few examples of RAM loaded drivers in the memory image, we observe that the 4 bytes preceding the UID value hold the actual file size of the driver. With that information, we are able to extract the exact RAM loaded drivers from the memory image programmatically. The extracted compressed drivers can then be uncompressed and analysed offline.

As mentioned, the extra kernel mappings not only hold the RAM loaded device drivers but also the I/O information. While searching for the RAM loaded binary drivers, we came across an entire region of memory which corresponds to the content of the address map at offset 0x34525696. Offset 0x34525696 points to the user local data region in the local page directory. Therefore, our initial hypothesis is that this is the region which the kernel mapped for use as the I/O buffer. Since most I/O operations are buffered for performance benefits, this would imply that when an application requests for an input from the user (for example, a password), the input will first be buffered before being read by the application. The reverse is also true such as when an application needs to write data to a file. Therefore, further work can be done to research on the feasibility of monitoring the I/O buffer in order to intercept the information.

As such, since the framebuffer might be on the RAM, there is also a possibility that the current active screen of the device can be found in the image. Furthermore, for processes that have ended, traces of them may still be residing in the RAM since a deallocation of the memory does not result in the RAM

being cleared. The reverse may also be true. However, for Symbian, when an unit of memory is allocated, it is initialized to zero.

Considering these characteristics of memory allocations and deallocations, we may be able to extract information such as the phone numbers, call logs, SMSes, recently opened files and even the plaintext version of the encrypted files; since in order to process an encrypted file, it must first be decrypted into the RAM.

6.4 Further Experiment: Page Tables Error

We have attempted to acquired the entire memory space from 0x00000000 to 0xFFFFFFFF without any unexpected error except during the acquisition of the page tables region from 0xC4000000 to 0xC8000000. A smooth process of non-disrupted successful acquisition of the page tables memory region is only possible from 0xC4000000 to 0xC6024599. When the tool tries to resolve the address at 0xC6024600, the smartphone was rebooted. We attempted the experiment a few times and the faulting point is consistent. This is an anomoly as we know that we already have the exception handling mechanism in place and this reboot must be caused by something else.

Unfortunately, we are unable to trace the system as tracing is disabled on commercial phones. Therefore, we are unable to identify the exact cause of this error. Our implemented solution is to skip this unaccessible memory and go to the next address. We then observed that after this faulting address at 0xC6024600, there are several interleaved regions of unaccessible memory.

A hypothesis that requires further investigation to verify in our future work is that this error could be caused by implementation differences on commercial phones. For example, Nokia, which is the manufacturer of N97, could have chosen to map a smaller memory range instead of that indicated by Symbian. It is therefore necessary to investigate where the manufacturer actually output the debug trace to. Possible investigative approaches could target the JTAG, Serial, or other ports on the phone.

7 Conclusions

With the prevalence of smartphones and the increasing amount of important information they are holding and storing, it is necessary to be equipped with the capability to conduct an in-depth security analysis and forensics investigations of smartphone information theft malwares. Therefore, in this paper, we identified the need for a memory acquisition technology or tool to conduct raw volatile memory acquisition from live Symbian smartphones. We investigated the different potential approaches to achieve this task and concluded that the kernel space driver approach to perform address de-referencing is the simplest, and most flexible and portable way to achieve the acquisition of raw volatile memory data from the live Symbian smartphones. This approach is also able to obtain a full-coverage view of the entire memory space. We designed and developed the tool for the purpose of performing the acquisition. Along the way, we also

solved the problems relating to exception handling and detection of unmapped addresses by enhancing the tool.

In addition, we identified the relevant memory regions of interest to facilitate further in-depth security analysis and forensics investigations of malwares. Subsequently, we analysed the ROM memory image, the kernel data, heap and stacks memory image, and the extra kernel mappings (I/O, RAM loaded device drivers) to identify ways and devise methods to detect and extract useful information (for examples, the identification and matching of the ROM drivers, the identification and extraction of the RAM loaded device drivers, and the identification of the pathnames of the loaded executables) to support further analysis. Some interesting observations with regard to the I/O buffer region and corresponding information in the user local data memory region in the local page directory, were made.

With this research work, we hope that it provides more insights into the Symbian operational environment and an understanding of how more in-depth anti-malware tools and forensics acquisition and analysis tools can be designed and developed.

References

1. Thing, V.L.L., Subramaniam, P., Tsai, F., Chua, T.-W.: Mobile phone anomalous behaviour detection for real-time information theft tracking. In: International Conference on Technical and Legal Aspects of the e-Society (February 2011)
2. Willassen, S.: Forensics and the GSM mobile telephone system. International Journal of Digital Evidence 2(1), 1–17 (2003)
3. Casadei, F., Savoldi, A., Gubian, P.: Forensics and SIM cards: an overview. International Journal of Digital Evidence 5(1), 1–21 (2006)
4. Kim, K., Hong, D., Chung, K., Ryou, J.-C.: Data acquisition from cell phone using logical approach. In: Proceedings of World Academy of Science, Engineering and Technology, vol. 26 (December 2007)
5. Mokhonoana, P.M., Olivier, M.S.: Acquisition of a Symbian smart phone's content with an on-phone forensic tool. Department of Computer Science, University of Pretoria (2007)
6. Distefano, A., Me, G.: An overall assessment of mobile internal acquisition tool. In: Proceedings of the 8th Digital Forensics Research Conference (DFRWS), Digital Investigation, vol. 5(1), pp. S121–S127 (September 2008)
7. Jansen, W., Delaitre, A., Moenner, L.: Overcoming impediments to cell phone forensics. In: Proceedings of the 41st Hawaii International Conference on System Sciences (2008)
8. Hoog, A.: Android forensics. Presented at Mobile Forensics World 2009 (May 2009)
9. Willassen, S.: Forensic analysis of mobile phone internal memory. In: Pollitt, M., Shenoi, S. (eds.) Advances in Digital Forensics. IFIP, vol. 194, pp. 191–204. Springer, Boston (2006)
10. Al-Zarouni, M.: Introduction to mobile phone flasher devices and considerations for their use in mobile phone forensics. In: Proceedings of the 5th Australian Digital Forensics Conference (December 2007)
11. Thing, V.L.L., Tan, D.J.J.: Symbian smartphone forensics and security: Recovery of privacy-protected deleted data. In: Chim, T.W., Yuen, T.H. (eds.) ICICS 2012. LNCS, vol. 7618, pp. 240–251. Springer, Heidelberg (2012)

12. Thing, V.L.L., Chua, T.-W.: Symbian smartphone forensics: Linear bitwise data acquisition and fragmentation analysis. In: International Conference on Security Technology (November 2012)
13. AVG, Mobilation (May 2012), http://www.avg.com
14. Robota, Anti-virus scanner for symbian mobile phones (May 2012), http://www.robota.nl
15. Dr.Web, Mobile security suite (May 2012), http://www.drweb.com
16. Lookout, Mobile security (May 2012), https://www.mylookout.com
17. Gartner: Market Share: mobile communication devices by region and country, 3q11 (November 2011), http://www.gartner.com
18. Gartner: Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent, (November 2011), http://www.gartner.com/it/page.jsp?id=1848514
19. Statcounter, Top 8 mobile operating systems (February 2012), http://gs.statcounter.com/#mobile_os-ww-monthly-201202-201202-bar
20. Sales, J.: Symbian os internals: Real-time kernel programming (January 2006)
21. Vomel, S., Freiling, F.C.: A survey of main memory acquisition and analysis techniques for the windows operating system. Digital Investigation 8(1), 3–22 (2011)
22. Thing, V.L.L., Ng, K.-Y., Chang, E.-C.: Live memory forensics of mobile phones. Digital Investigation 7, S114–S120 (2010)
23. Zdziarski, J.: iPhone forensics. OI'Reilly Media (September 2008)
24. Hoog, A., Strzempka, K.: iPhone and iOS forensics. Syngress (June 2011)
25. Paraben: Device seizure, http://www.paraben.com/
26. Sourceforge, P3nfs, http://sourceforge.net/projects/p3nfs.berlios/
27. Breeuwsma, I.M.F.: Forensic imaging of embedded systems using jtag (boundary-scan). Digital Investigation 3(1), 32–42 (2006)
28. Savoldi, A., Gubian, P.: Symbian forensics: An overview. In: IEEE International Conference on Intelligent Information Hiding and Multimedia Signal Processing, pp. 529–533 (August 2008)